Computers & Security 92 (2020) 101750

Contents lists available at ScienceDirect

Computers & Security

journal homepage: www.elsevier.com/locate/cose

Prioritizing data flows and sinks for app security transformation☆

Ke Tian[a,*], Gang Tan[b], Barbara G. Ryder[a], Danfeng (Daphne) Yao[a,1]

[a] Department of Computer Science, Virginia Tech, Blacksburg, VA, 24060 United States
[b] Department of Computer Science and Engineering, Penn State University, University Park, PA 16802 United States

ARTICLE INFO

Article history:
Received 5 June 2018
Revised 1 September 2019
Accepted 6 February 2020
Available online 7 February 2020

Keywords:
Android security
Application rewriting
Sink prioritization
Program analysis
Machine learning
Data-flow analysis

ABSTRACT

There have been extensive investigations on identifying sensitive data flows in Android apps for detecting malicious behaviors. Typical real world apps have a large number of sensitive flows and sinks. Thus, security analysts need to prioritize these flows and data sinks according to their risks, i.e., flow ranking and sink ranking. In this paper, we present an efficient graph-algorithm based risk metric for prioritizing risky flows and sinks in Android grayware apps. The new risk metric is quantitative and can differentiate the sensitivities of flows and sinks in an app. In the experiments, our risk prioritization produces orderings that are highly consistent with manual inspection. To enable post-detection security enforcement of sensitive sinks, we also present an automatic rewriting framework that utilizes the above prioritization technique. Our rewriting strategies are more feasible than the state-of-art solutions by supporting flow- and sink-based rewriting. We implement our prototype as ReDroid. ReDroid is designed for security analysts who manage organizational app repositories and customize third-party apps to satisfy organization imposed security requirements. We use ReDroid to rewrite both benchmark apps and real world grayware.

1. Introduction

The research on mobile app security has been consistently focused on the problem of how to differentiate malicious apps from benign apps. Static data-flow analysis has been widely used for screening Android apps for malicious code or behavioral patterns (e.g., Elish et al., 2015; Gibler et al., 2012; Gordon et al., 2015; Lu et al., 2012). In addition, the use of machine-learning methods enables automatic malware recognition based on multiple data-flow features (e.g., Arp et al., 2014; Tian et al., 2016).

These solutions are useful for security analysts who manage public app marketplaces or organizational app repositories. An *organizational app repository* is a private app sharing platform within an organization, the security of apps on which is regulated and approved by the organization based on its security policies and restrictions. For example, the organization may be a government agency where employees with certain security clearance levels are required to install apps from the specified repository to their work phones. The organization may also be a company, where employees possessing highly sensitive proprietary information and trade secrets are required to install apps compliant with the company's IT security policies.

In these scenarios, a security analyst is often faced with a *new* type of apps, besides malware and benign apps. These apps are mostly benign, but with undesirable behaviors that are incompatible with the organization's policies. Such apps or app libraries may be from trustworthy companies or developers, and may have passed standard conventional screenings. However, the app contains potentially sensitive data flows that are incompatible with the organization's policies. As requesting developers to change their code is oftentimes infeasible, current practices are to either reject the app or reluctantly accept it, despite its undesirable security behaviors. A similar dilemma is faced by individual users as well. For example, a privacy-conscious user may wish to dynamically restrict an app's location sharing at runtime according to her specific preferences.

Our work is motivated by this new need of security customization of apps. A general-purpose framework for customizing the security of off-the-shelf apps would be extremely useful and timely. Such a framework involves several key operations: *(1)* **[Prioritization]** to identify problematic code regions in the original app, *(2)* **[Modification]** to modify the code and repackage the app. In addition, post-rewrite monitoring may be needed, if the access or

☆ A preliminary version of the work appeared in the proceedings of workshop on Forming an Ecosystem Around Software Transformation (FEAST), collocated with the ACM Conference on Computer and Communications Security (CCS). Dallas, TX. Nov. 2017. (Tian et al., 2017).

* Corresponding author.
E-mail addresses: ketian@cs.vt.edu (K. Tian), gtan@cse.psu.edu (G. Tan), ryder@cs.vt.edu (B.G. Ryder), danfeng@cs.vt.edu (D. (Daphne) Yao).
[1] Member, IEEE.

https://doi.org/10.1016/j.cose.2020.101750
0167-4048/

sharing of sensitive data is determined dynamically. We have made substantial progress towards these goals. We report several new techniques, including quantitative risk metrics for ranking sensitive data flows and sinks in Android apps.

Our major contribution is to propose a new prioritization algorithm to rank sensitive sinks for apps. Our approach is capable to capture internal data dependencies among sensitive sinks and provide sinks with evidence-based quantification of risks. The algorithm consists of two major components: a taint-flow based sensitivity aggregation and a machine learning based sensitivity quantification. Comparing with existing solutions that only aim to detect taint flows (Arzt et al., 2014) of sensitive sinks, our goal is to enable both sensitivity aggregation and quantitative ranking of sensitive sinks.

To achieve our goal, we first define a quantitative risk metric for sensitive flows and sinks in a taint-flow. For sensitive sinks, the metric summarizes all the sensitive flows that a sink is involved in. We design an efficient graph algorithm that computes the risks of all sensitive sinks in time linear to the size of a directed taint-flow graph G, i.e., $O(|E|)$, where $|E|$ is the number of edges in G. (A taint-flow graph is a specialized data-flow graph that only contains data flows originated from predefined sensitive sources and leading to predefined sensitive sinks.) The risk value of a sink is calculated based on *all* the sensitive API calls made on the sensitive data flows leading to a sink. A sink may be associated with multiple such sensitive flows.

In order to rank risky sinks, we map sensitive API calls to quantitative risk values, using a maximum likelihood estimation approach through parameterizing machine-learning classifiers. These classifiers are trained with permission-based features and a labeled dataset. Then, we use the risk metric to identify and rewrite the sinks associated with the riskiest data flows without reducing the app's functionality.

Our work also moves a step towards sink-specific rewriting by extending app rewriting solutions for Android. Rewriting is regarded as post-detection mitigation to enforce security policies. Existing solutions are specific to certain code issues and are not designed for our security customization scenarios (Davis and Chen, 2013; Fratantonio et al., 2015). Due to the specific rewriting needs, the target locations to be rewritten are relatively straightforward to identify. Most of the existing solutions use direct parsing for code-region identification. Yet, oftentimes it is unclear which regions of the code need to be modified in order to achieve the best risk reduction. If additional post-rewrite monitoring is required at runtime, then modifying *every single* sensitive flow or sink may substantially slow down the performance. Since the rewriting process at the binary or bytecode level is error-prone, minimizing the impact of rewriting on the original code structure is also important.

We demonstrate a practical Jimple-level code rewriting technique that can verify and terminate the riskiest sink *at runtime*. For the Android-specific inter-app inter-component communication (ICC) mechanism, we propose *ICC relay* to redirect an intent. We replace the original intent with a relay intent; the relay intent then redirects the potentially dangerous data flow to an external trusted app for runtime security policy enforcement. The communication between the modified app and the trusted app is via explicit-intent based ICC. The trusted app is where data owner may implement customized security policies.

The technical contributions of our work are summarized as follows.

1) We present a general sink-ranking approach that is useful for prioritizing sensitive data flows in Android apps. Specifically, our approach relies on two main technical enablers. The one enabler is a quantitative risk metric for sensitive flows and sinks in taint-flow graphs that is based on machine learning techniques.
   The other enabler is an efficient $O(|E|)$-time taint-graph based risk-propagation algorithm that ensures the maximum coverage of all sensitive sources and internal nodes of a sink.
2) We implement a proof-of-concept prototype called **ReDroid**[2]. We use ReDroid to demonstrate the usage of rewriting in defending ICC hijacking and privacy leak vulnerabilities. Our rewriting supports flow-based and sink-based rewriting, which is more feasible beyond the state-of-art rewriting solutions.
3) We have performed an extensive experimental evaluation on the validity of permission risks and sink rankings. Our manual inspection indicates that top risky sinks found by ReDroid are consistent with external reports. We compare various permission-based and non-permission-based risk metrics, in terms of their abilities to identify top risky sinks.
4) We demonstrate the feasibility and effectiveness of both inter-app ICC relay and logging-based rewriting techniques in testing DroidBench and ICC-bench apps. We also successfully customized recently released grayware. The customized app enables one to monitor runtime activities involving Java reflection, dynamic code loading, and URL strings.

Our ranking algorithm supports both *sink ranking* and *flow ranking*[3]. However, due to the interdependencies of flows, cutting a flow in the middle may cause much more runtime errors than removing the flow's end-point sink. In addition, a sink aggregates multiple flows, making them riskier than a single flow. Thus, we focus on rewriting sinks.

Comparing with the previous conference version (Tian et al., 2017), we substantially extended the paper from 7 pages to 13 pages by adding new content, and providing details of our approach and evaluation.

We summarize the differences in three aspects: (1) we conducted new experiments that evaluated and compared the effectiveness of our ranking and rewriting approach (Section 4.1-4.9). The experimental results validate the efficiency of our approach (Section 4.5 and 4.6). We also provided case studies on the ability to prioritizing sensitive flows (Section 4.6) and machine learning accuracy for transforming permission strings into risk values (Section 4.8). (2) We incorporated the pseudocode to describe our algorithms on computing risk scores of sinks from the risk propagation (Algorithm 1). We added new definitions (Definition 2 and 3 for aggregation notations) to elaborate our algorithm in details. (3) We substantially extended most part of the paper to improve its readability, added new related work, and provided details of our approach and implementation.

2. Overview

Before we give the overview of our approach in Section 2.2, we first show a few examples to motivate the needs for ranking sensitive data flows and rewriting apps for security.

We target data leaks in our current threat model, specifically data flows in an app that may result in the disclosure and exfiltration of sensitive data. With proper source-sink definitions, the proposed sink-ranking and rewriting-based monitoring framework can be extended to support other security applications, which is discussed in Section 3.9.

[2] ReDroid is short for *Re*writing An*Droid* apps.

[3] Flow ranking is a special case of sink ranking in our Algorithm 1.

Algorithm 1 Pseudocode for computing risk scores of sinks from the risk propagation. The function getSelfPermission() is used to compute the self risk. The function getRiskValue() is used to compute the risk value for each permission with machine learning.

Input: The sensitive taint-flow graph G a program.
Output: The sink set T with risk scores.
function PerAggre(Graph $G(V, E, S, T)$, aggregation function agg_func)
  V_{sort} = Topological_Sort(G)
  $G'(V, E', S, T)$ = Transitive_Reduction(G, V_{sort})
  /*P is a hashmap representing the aggregate permissions for all nodes in G, and r is the hashmap representing the risk score for all nodes in G */
  for each $v \in V$ **do**
    $P[v] = \emptyset, r[v] = 0$
  end for
  if $agg\_func == E2E$ **then**
    /* In case of E2E aggregation */
    for each $v \in V$ **do**
      $P[v]$ = getSelfPermission(v)
    end for
  else $agg\_func == SS$
    /* In case of SS aggregation */
    for each $v \in S \cup T$ **do**
      $P[v]$ = getSelfPermission(v)
    end for
  end if
  /* Propagation of sensitive permissions */
  for $(i = 1 \rightarrow V_{sort}.size())$ **do**
    $v = V_{sort}[i]$
    for $e = \{v_1 \rightarrow v\} \in G'$ **do**
      $P[v] = P[v] \cup P[v_1]$
    end for
    if $v \in T$ **then**
      for each $p \in P[v]$ **do**
        /* Map permissions to risk values */
        $r[v]+$ = getRiskValue(p)
      end for
    end if
  end for
  return r
end function

2.1. Motivation and design choices

2.1.1. Security usage of App rewriting

Table 1 summarizes the security applications with our rewriting. Our rewriting can identify multiple vulnerabilities such as ICC hijacking and privacy leak. We rewrite apps to enforce different security policies, these security policies help a security analyst efficiently detect vulnerable activities and offer security mitigations. Our rewriting framework can prevent vulnerabilities in stand-alone apps and vulnerabilities in app communication channels. We elaborate our rewriting feasibility with more details in Section 4.1.

We envision two types of use scenarios for app rewriting tools as follows. Both scenarios are possible. However, before rewriting tools can be made fully reliable, automated, and usable, the second use scenario is unlikely.

1) *Used by security analysts who manage app repositories.* Security analysts retrofit off-the-shelf apps for organizational app repositories to make them comply with organizational security policies. Employees download retrofitted apps into their regulated work phones.
2) *Used by individuals to customize privacy*. Users have specific data-access preferences that cannot be satisfied by an off-the-shelf app and choose to retrofit the app.

Table 1
The vulnerabilities that can be identified by our rewriting framework. Our rewriting framework can identify vulnerabilities in stand-alone apps and vulnerabilities in app communication channels.

Type	Vulnerability	Our framework addresses
Inter-app Com. (IAC)	ICC hijacking	✓
	Collusion	✓
Stand-alone App	Privacy Leak	✓
	Reflection	✓
	String Obfuscation	✓
	Dynamic Code Loading	✓

2.1.2. Flow and sink prioritizing

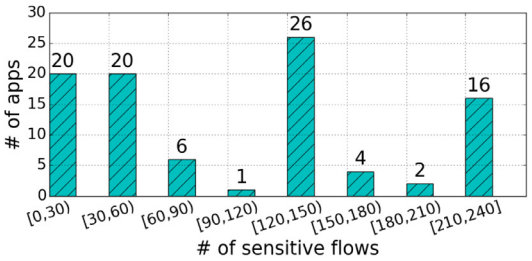
Apps typically have a large number of sensitive flows. In order to show the importance of ranking these flows, we conduct an experiment on 100 apps that are randomly selected from Android Malware Genome Database (Zhou and Jiang, 2012). We use FlowDroid Arzt et al. (2014) for static program analysis and SuSi Rasthofer et al. (2014) for labeling sensitive sources and sinks. Our sensitive source and sink definitions follow SuSi, where sources are calls to read sensitive data and sinks are calls that can leak sensitive data. Fig. 1a shows the distribution of the number of sensitive flows. Fig. 1b presents the distribution of the number of source and sink nodes. A single app can contain more than 20 distinct sinks. A data flow is sensitive, if any node on its path is labeled sensitive. These statistics indicate the complexity of sensitive flows and sinks in a single app. An appropriate prioritizing mechanism would help a security analyst to facilitate the app monitoring, e.g., identifying most sensitive flows and sinks. The motivating experiment indicates the need for prioritizing sensitive flows and sensitive sinks according to systematic quantitative metrics.

2.1.3. Flow-based sink ranking vs. flow ranking

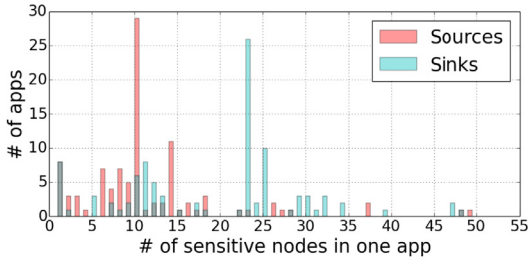
The risk of a sink should be associated with *all* the sensitive paths flowing into that sink, which usually involves many nodes besides the sink itself. A sink may be reachable by multiple sensitive flows. Therefore, the risk factors from all these flows need to be *aggregated* in order to completely reflect the risk of a sink. Our sink ranking is computed on flows, i.e., *flow-based sink ranking*. In comparison, computing the risk of a single flow is simpler. It can serve as a basic building block for computing the risk of a sink. Flow ranking is a special case of our sink ranking algorithm. However, flow ranking should not be used to guide the rewriting, as it may provide an incomplete risk profile of the code.

2.1.4. Sink rewriting vs. flow rewriting

Once the most sensitive sink is identified, rewriting that endpoint region likely produces a minimal impact on the app's functionality. Revising a flow (e.g., cutting an internal edge) requires substantial more engineering efforts, due to the interdependency of flows. However, in some scenarios, flow rewriting may be more fine-grained than sink rewriting. For example, a sink may be associated with n flows, only one of which is sensitive and needs to be modified. The other $n-1$ flows do not involve sensitive data or operations and can be left intact. Our logging based rewriting supports both flow- and sink-based rewriting. This strategy can log and inspect each node along a data flow. In contrast, the ICC relay is more focusing on sinks (e.g., startActivity) with ICC vulnerabilities.



(a) The distribution of the sensitive taint flows distinguished by source and sink pairs.



(b) The distribution of # of sensitive sources and sinks in the app dataset.

Fig. 1. The example for the distribution of sensitive flows and sources and sinks.

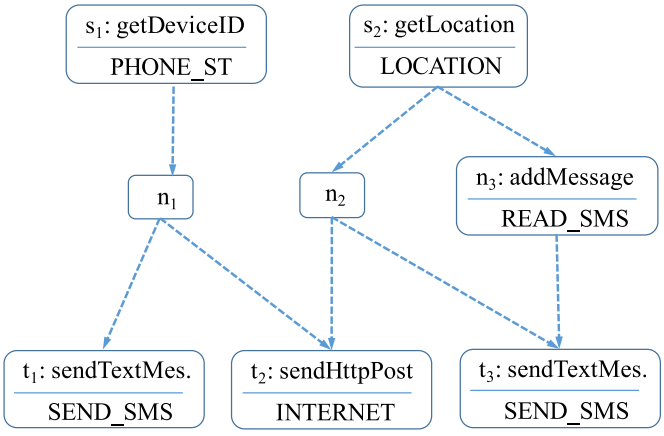


Fig. 2. An example of a taint-flow graph. Nodes represent function calls or instructions. Permissions (shown at the bottom of a node) associated with the functions (shown at the top of a node) are shown. Directed edges represent data dependence relations.

2.1.5. Sensitive API-based risk vs. permission-based risk

These two risk metrics are *equivalent* in our model. We map the sensitive API calls of a data-flow path into their corresponding Android permissions, as shown in Fig. 2. For example, getLocation API call is mapped to LOCATION permission. We then quantify permissions' risks through statistical methods. Our risk-computation approach can be extended to support other types of risk definitions (e.g., by leveraging data-flow features in Android malware classifiers such as (Arp et al., 2014; Elish et al., 2015)).

2.1.6. A toy example

In Fig. 2, we use a toy taint-flow graph (simplified from GoldDream) to illustrate several possible sink-ranking methods and how they impact security. The figure contains two sensitive source (s_1 and s_2), three sensitive sinks (t_1, t_2, and t_3) and several internal nodes, one of which involves a sensitive function. Android permissions associated with the functions are shown at the bottom of nodes. Consider two approaches for ranking the risks of sensitive sinks: a sink-only approach and a source-sink approach. In the straightforward sink-only approach, the risk level of a sink is determined only by the sink's function name and the permission it requires. This approach clearly cannot distinguish two different sinks sharing the same function name, e.g., t_1 and t_3. It is also unclear how to compare the risk level of t_1's permission and t_2's permission.

In a more complex source-sink approach, the risk of a sink is determined not only by the sink itself, but also by all of its sensitive sources. For example, in Fig. 2 the risk of sink t_2 is associated with the permission set (PHONE_ST, RECEIVE_SMS, and INTERNET), where the first two permissions are from the two sources s_1 and s_2, and the last permission is from the sink itself. Although this source-sink approach also needs a method to quantify the risks of permissions, it is more desirable than the sink-only method. The reason is that the source-sink approach more accurately reflects sensitive flow properties.

This example indicates that a reasonable sink-ranking algorithm needs (1) to capture internal data dependences; (2) evidence-based quantification of risk. In ReDroid, we evaluate and compare several sink ranking mechanisms in terms of how they impact app rewriting.

2.2. Definitions

We describe the workflow of our flow-ranking analysis for sink ranking and rewriting. Our new capability is the efficient computation of *end-to-end* flow risks, quantifying risks associated with data-flow dependence. We first give several key definitions used in our model, including self risk, aggregate risk, and the standard taint-flow graph.

Definition 1. *Taint-flow graph* is a directed graph $G(V, E, S, T)$ with source set $S \subseteq V$ and sink set $T \subseteq V$ and $S \cap T = \emptyset$, where for any flow $f = \{v_0, v_1 ... v_n\}$ in G, $v_0 \in S$ and $v_n \in T$ and $e = \{v_i \rightarrow v_j\} \in E$. The flow f represents the taint-flow path from the source v_0 to the sink v_n, which is denoted as $f = \{v_0 \rightsquigarrow v_n\}$.

The taint-flow graph is a subgraph of the data-flow graph. Our model considers two types of risks for each node in the taint-flow graph, *self risk* and *aggregate risk*, which are defined next.

Self Risk. Given a taint-flow graph $G(V, E, S, T)$ and a node $v \in V$, the self risk $P_s[v]$ of v is the risk associated with v's execution. $P_s[v] = \emptyset$, if no risk is involved.

Aggregate risk. Given a sink $t \in T$ in the taint-flow graph G, the aggregate risk $P[t]$ of sink t is a set that represents the risks associated with the taint flows of t under some aggregation function agg_func().

Our instantiation of the risk metric is based on the analysis of risks associated with sensitive APIs on data flows into a sink. Therefore, self risk is also referred to as *self permission*, and aggregate risk is also referred to as *aggregate permission* for the rest of the paper. We compute risk values of permissions through a maximum likelihood estimation approach.

In Section 3, we present two instantiations of the aggregation function agg_func(). One is a straightforward source-sink (SS) aggregation, where the aggregate risk of a sink is the union of self risks of the sink and its source(s). The other is the end-to-end (E2E) aggregation, which outputs all the permissions associated with all the taint flows that the sink is in. Our experiments compare how these two aggregation functions impact the flow-ranking accuracy.

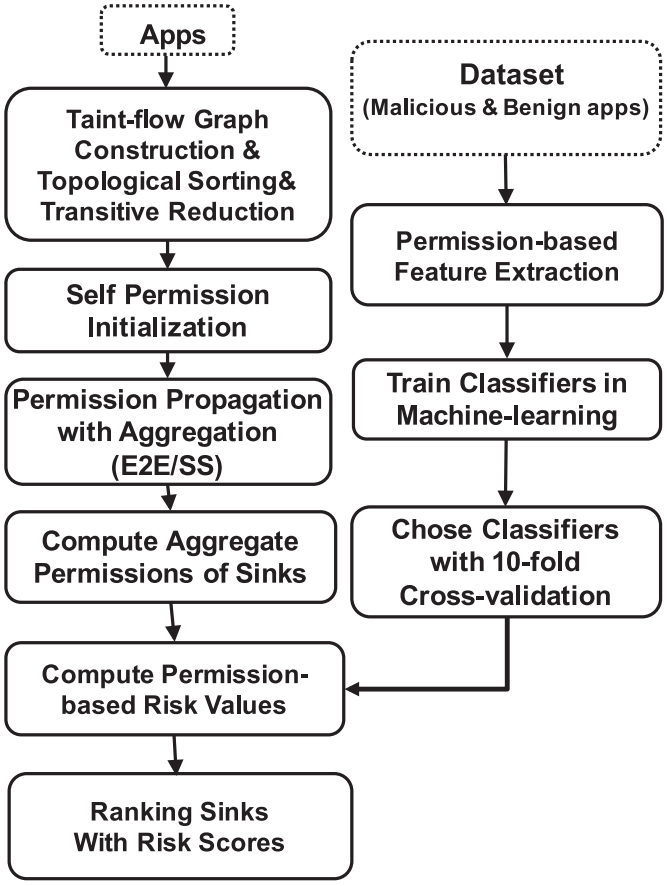


Fig. 3. Our workflow for prioritizing risky sinks.

2.3. Workflow

Fig. 3 shows our workflow for sink ranking with graph propagation. We briefly describe these operations.

1) *Taint-flow construction*. We generate the taint-flow graph that describes sensitive data flows from sources to sinks. Nodes in the taint-flow graph are mapped to their self risk values, as defined above. This mapping process may vary, if different risk aggregation function is used. We demonstrate two such functions, source-sink aggregation and end-to-end aggregation.
2) *Risk propagation to sinks.* The operation outputs the aggregate risk set for each sensitive sink. The propagation needs to efficiently traverse the data-dependence edges from sources to sinks. The key in designing the propagation algorithm is to visit each graph edge a constant number of times, realizing $O(|E|)$ complexity, where $|E|$ is the size of the graph edges. We present our solution in Section 3.1.
3) *Permission-Risk Mapping*. We follow a maximum likelihood estimation approach to produce a risk value for each permission empirically. Intuitively, the risk of a permission is high, if the permission is often requested by malware apps, but rarely by benign apps. With labeled training data and machine learning (ML) classifiers with permission-based features, we automatically map permissions to risk values $r \in [0, 1]$. We present our ML-solution in Section 3.5.[4]
4) *Flow-based Sink Prioritization*. To obtain the risk score of a sink, one needs to quantify the risk associated with the sink's aggregate permission. The risk score of a sink is computed by its correlated permissions with risk values. We rank the sinks according to their risk scores. The risk score of sinks captures its importance and security properties in the app.

Risk ranking guides the app customization for risk reduction. For example, one can choose to intercept the riskiest sink and relay the flow to a trusted runtime monitor. We describe several security customization techniques in Section 3.6. Besides rewriting, the sink-ranking technique is also useful for static analysis based malware detection.

3. Risk metrics and computation

We aim to quantitatively compute and rank risks of sinks in an app. Our approach is to construct the sensitive taint-flow graph and compute the set of permissions associated with each flow through graph propagation algorithms. The aggregation algorithms find the *accumulated* risk factors (naming permissions) of a source-sink path in $O(|E|)$ complexity, where $|E|$ is the number of edges in the graph. Our risk is based on the permissions of sensitive APIs. Our pseudocode is given in Algorithm 1 in the Appendix.

Next, we describe technical details of our operations. We present risk propagation in Section 3.1, permission mapping in Section 3.5 and rewriting in Section 3.6.

3.1. Risk propagation

The purpose of risk propagation is to aggregate all risky flows associated with a sink.

3.2. Graph construction

We use Android-specific static program analysis tools (namely FlowDroid) to obtain the taint-flow graph $G(V, E, S, T)$, which represents the data dependence among code statements in the app from sensitive sources to sinks, where $n \in V$ is the statement in the code and $e = \{n_1 \rightarrow n_2\} \in E$ represents that n_2 is data dependent on n_1, $S \subseteq V$ is the sensitive source set S and $T \subseteq V$ is the sensitive sink set. Loops may occur due to control dependence, e.g., while loops. Our subsequent permission aggregation only computes over distinct permissions. Because each loop execution involves the same set of permissions, we follow each loop only once. This reduction generates a directed acyclic graph $G(V, E, S, T)$.

Security analysts can customize their definitions of sensitive sources and sinks based on organizational security policies. These definitions impact the static taint analysis. For example, smaller sensitive sets usually give fewer sensitive flows required to rewrite.

3.3. Transitive reduction

The purpose of transitive reduction is to maximally remove redundant edges while preserving reachability of the graph (Aho et al., 1972). Transitive reduction helps us to reduce the iteration of edges in our quantitative propagation analysis. It does not affect our final results because it preserves the reachability from a source $s \in S$ to a sink $t \in T$. Specially, the reduced graph has the same nodes, sources, and sinks, but different edges. Transitive reduction transforms $G(V, E, S, T)$ into $G'(V, E', S, T)$.

Specifically, the reduced graph $G'(V', E', S', T')$ is a subgraph of the original taint-flow graph $G(V, E, S, T)$, with $E' \subseteq E$, and the number of nodes keep the same $V' = V$, $S = S'$ and $T = T'$.

Transitive reduction produces a directed acyclic graph (DAG). For each sink t, it has a subgraph reversely rooted by t, i.e., there

[4] Other permission-risk quantification techniques may be used, e.g., Bayesian-Network based Android permission risk analysis (Peng et al., 2012).

exists a subgraph rooted by t, if the directions of edges are reversed.

3.4. Risk propagation to sinks

With the assignment of all the statements, we perform risk propagation analysis algorithm on the graph $G'(V, E', S, T)$. Each node in the graph is initialized with the corresponding self risk and the empty set as its aggregate risks. Specifically, we provide two different aggregation algorithms: SS (source-to-sink) aggregation and E2E (end-to-end) aggregation in Definition 2.

Definition 2. Denote a taint-flow path in a transitive reduced taint-flow graph $G'(V, E', S, T)$ by $f = \{s \rightsquigarrow n_1 \rightsquigarrow ... \rightsquigarrow n_i \rightsquigarrow t\}$, where $s \in S$, $t \in T$ and n_i is an internal node on f. We define source-sink (SS) aggregation and end-to-end (E2E) aggregation methods as follows.

SS aggregation. The aggregate risk set $P[t]$ of a sink $t \in T$ is defined as

$$P[t] = P_s[t] \cup \left\{ \bigcup_{\{s \in S \mid \exists\ f=\{s \rightsquigarrow t\}\}} P_s[s] \right\} \quad (1)$$

E2E aggregation. The aggregate risk set $P[t]$ of a sink $t \in T$ is defined as

$$P[t] = P_s[t] \cup \left\{ \bigcup_{\substack{\{s \in S,\ \{n_1 ... n_k\} \in f \\ \mid\ \exists\ f=\{s \rightsquigarrow t\}\}}} P_s[s] \cup P_s[n_1] ... \cup P_s[n_k] \right\} \quad (2)$$

E2E aggregation for a sink t generates a set that consists of all the distinct permissions corresponding to the taint-flow subgraph that is reversely rooted by t. The difference between the two aggregations is on the sensitive internal nodes. The SS aggregation only considers the sensitive sources and sinks, whereas the E2E aggregation includes the permissions of internal nodes. The E2E aggregation produces all the distinct permissions that are required by the taint-flow subgraph that is reversely rooted by a sink t.

Following a taint flow, the aggregate risk set of a node is non-decreasing (i.e., increasing or stable). If n_j is the successor of n_i on a path, the permission used in n_i is propagated to n_j. Algorithm 1 shows the pseudocode for the permission aggregation and risk computation.

For the example in Fig. 2, the output of E2E and SS aggregations are the same for sinks t_1 and t_2, i.e., $P[t_1]$ ={PHONE_ST, SEND_SMS}, and $P[t_2]$ ={PHONE_ST, RECEIVE_SMS, INTERNET}. However, they are different for sink t_3. Specifically, for SS aggregation $P[t_3]$ = {RECEIVE_SMS, SEND_SMS}, whereas E2E aggregation has a larger aggregate risk set for the sink, which is $P[t_3]$ = {RECEIVE_SMS, READ_SMS, SEND_SMS}. Our experiments in Section 4.5 show how they impact security and rewriting.

The flow-based sink aggregation algorithm can be modified to compute risk scores of flows. For a flow $f = \{v_0 \rightsquigarrow v_n\}$, risk value of node $n \in f$ is computed by *getRiskValue*(n). The risk score of flow f is computed though the propagation from v_0 to v_n without aggregation of other flows.

3.5. Permission-risk mapping with maximum likelihood estimation

The purpose of permission-risk mapping is to quantify the risk values of sensitive permissions. Although research has shown certain permissions are predictive of malware and researchers propose risk-quantification mechanisms for permissions (e.g., rule-based Enck et al. (2009) and Bayesian-based probabilistic models (Peng et al., 2012)), how to use them for prioritizing sinks for rewriting has not been systematically studied.

Definition 3 Sink Risk**.** For a sink t in a taint-flow graph G, we evaluate its risk based on its aggregate permissions $P[t]$. In ReDroid, we compute $r(t)$ as the summation of quantified permission risks:

$$r(t) = \sum_{p \in P[t]} w(p) \quad (3)$$

where $w()$ is a function that maps a permission p to a quantitative risk value $w(p)$.

We follow a maximum likelihood estimation approach, to empirically map a permission p to their quantitative risk value $w(p)$. We parameterize binary classifiers with permission-based features. The task of binary classifiers is to label an unknown app as benign (negative) or malicious (positive). The optimal permission-risk mapping and configuration should *maximize* the accuracy of a binary classifier, i.e., low false positives (false alarms) and low false negatives (missed detection of malware).

We use the feature-importance value of a permission as a security measurement for the permission sensitivity. An important permission is an indicator of malicious apps, because malicious apps request more critical permissions (e.g., READ_SMS) from empirical studies (Arp et al., 2014). A permission (e.g., INTERNET) existing in both benign and malicious apps has a low importance value. Our method automatically maps a permission string into a quantitative risk value.

Our training set is selected from both malicious and benign app dataset. We evaluate several supervised learning techniques (e.g., KNN, SVM, Decision Tree and Random Forest) and compare their accuracy in Section 4.8. The Random Forest classifier achieves the highest accuracy. The evaluation of these classifiers is based on standard measurements, namely 10-fold cross-validation. We use the classifier that maximizes the classification accuracy to compute the risk values of permissions.

3.6. Automatic app rewriting

We rewrite on the app's intermediate representation Jimple, which is based on Java analysis framework Soot. The Soot supports Java-specific function instrumentation. We implement our rewriting framework by supporting Android-specific components, e.g., ICC. Our prioritizing algorithm is regarded as a Soot plug-in to quantitatively compute risk scores for sinks. Table 2 presents the comparison of ReDroid with existing Android rewriting frameworks. Our ReDroid supports more rewriting operations, including intent redirection, than current rewriting solutions. Unlike previous rewriting demonstrations on Smali (such as Davis and Chen (2013); Xu et al. (2012)), our inter-app ICC relay rewriting approach requires more substantial code modification[5].

The target sink can be selected by the sink prioritization. We identify a target sink based on its package, class and method names and the context of the sink (e.g., parameters). Once the target sink is located, code modification is more challenging, as it needs to ensure the successful execution of the modified app. We reuse the registers and parameter fields from the original code. We replace the sink function with a new customized function. We compile the new function separately and extract its Jimple code. The new function's parameters need to be compatible with the API specification and the context.

3.7. Proactive rewriting with inter-app ICC relay.

This ICC-relay strategy *redirects* data flows to the risky sink of an app to a trusted proxy app, so that the trusted proxy app can

[5] Without access to the code of existing solutions, we aim to release our framework to facilitate the reproduction of app rewriting.

Table 2
Comparison of ReDroid with existing Android rewriting frameworks. Method invoc. is short for method invocation to invoke a customized method instead of an original method. RetroSkeleton is implemented based on I-ARM-Droid. ReDroid supports more rewriting strategies than the existing frameworks.

Rewriting Granularity	I-ARM-Droid Davis et al. (2012) RetroSkeleton Davis and Chen (2013)	ReDroid (Ours)
Package-level (Repackage)	✓	✓
Class-level (Class Inject)	✓	✓
Method-level (Method Invoc.)	✓	✓
ICC-level (Intent Redirect)	–	✓
Flow-based Rewriting	–	✓
Sink-based Rewriting	–	✓

inspect the data before it is consumed (e.g., sent out). Our redirection mechanism leverages Android-specific inter-component communication (ICC) and explicit intent. Android ICC mechanism enables the communication among different apps (Chin et al., 2011).

The original intent is replaced by a new explicit intent that invokes methods in the proxy app in order to complete the task. The original intent is cloned and stored in a data field of the new explicit intent. This redirection mechanism gives the proxy an opportunity to inspect the sensitive data of the original intent *at runtime*. Specifically, once the trusted proxy receives a request from the rewritten app via ICC, the execution of the rewritten app is paused (i.e., onPause is invoked). The proxy can choose to log the requests and analyze them offline, or perform online inspections (with respect to pre-defined policies). Upon proxy's completion, the original intent is re-constructed to allow the rewritten app to continue its execution. The execution of the app may be impacted by the invocation of the ICC, especially when the proxy's inspection is performed online.

3.8. Passive rewriting with logging

Passive logging-based rewriting is useful for intercepting dynamically generated data structures that are related to risky sinks, e.g., a URL string in an HTTP request that is manipulated along the taint flow. The static taint-flow analysis can detect the suspicious risky sink with strings as its parameters. However, the exact content of the string usually cannot be resolved through static analysis. Logging them to local storage enables offline inspection.

The advantages of the logging approach are two-fold. (1) It is relatively straightforward to implement at the Smali level, and (2) logging does not impact the execution path of the rewritten app. The rewritten app executes without interruption. However, the analysis in this approach is conducted the offline, whereas the redirect mechanism can actively block data leaks at runtime if needed.

3.9. Discussion and limitations

We discuss limitations of our approach and future directions. This paper is focused on technical aspects of app modifications. Legal issues (e.g., copyright restrictions) are out of the scope of discussion.

3.10. Flow precision

Static analysis cannot estimate exactly dynamic execution paths, our graph analysis is conservative and may over-approximate the permissions related to the sinks. Our prototype is built on the existing framework FlowDroid, for the facility of generating flow-sensitive graphs. Our approach can be also built on other program analysis frameworks, e.g., Gibler et al. (2012); Lu et al. (2012). Our main source of imprecision in sink ranking comes from imprecise data-flow graphs. Current static program analysis over-approximates apps' behaviors by considering all possible paths, including some infeasible paths. The over-approximation in graphs introduces inaccuracy for our quantitative analysis. Thus, the corresponding aggregate permissions and risks of sinks in ReDroid may be overestimated.

3.11. Native code

Native code gains its popularity recently for code obfuscation (Tam et al., 2017). Android supports invoking sensitive APIs in a reflection-like way from native code dynamically (Afonso et al., 2016). Native code introduces missing edges for the static tool FlowDroid to generate graphs in our analysis. A possible mitigation is to introduce hybrid analysis. Hybrid analysis combines static analysis similar to our approach and dynamic analysis to resolve reflected APIs by running the application at runtime. However, hybrid analysis suffers from performance and is not as scalable as static analysis. Therefore, more substantial work is needed for balancing precision and scalability.

3.12. Dynamic permission

Google has recently introduced Android dynamic permission to protect user privacy[6]. Dynamic permission provides an interface for denying the access of reading private data (i.e., sources). However, dynamic permission ignores the data flow dependence. It cannot track data and estimate how the private data is abused. In contrast, our rewriting is based on ranked sinks with the aggregated sensitive data flows. Our approach can estimate the risk score of a dangerous sink and provide customized rewriting operations. In compliment with dynamic permission, our rewriting provides two-factor data verification for both sources and sinks.

3.13. Rewriting challenges

Code rewriting requires substantial technique skills. If not careful, the retrofitted app may not be successfully recompiled or may crash at runtime. Our sink ranking and rewriting is automated. However, the current rewriting demonstration is based on the intermediate representation Jimple via reverse engineering. Current cutting-edge reverse engineering tools (e.g., Soot) cannot extract Jimple IR from native code or encrypted code. Therefore, more substantial work is needed for increasing the rewriting usability.

4. Experimental Eevaluation

We use FlowDroid Arzt et al. (2014) for static program analysis. Our map Tables 1–7 ping from a statement into the re-

[6] https://goo.gl/9FTnEL.

Table 3
Evaluation of ICC relay and logging based rewriting on benchmark apps. The column of Re. means the number of apps that can run without crashing after rewriting. The column of In. means the number of apps that we can successfully invoke the sensitive sink and observe the modified behaviors.

App Category	#of ICC Exits	Logging Success		ICC Relay Success	
ICCBench		Re.	In.	Re.	In.
icc_implicit_action	1	1	1	1	1
icc_implicit_category	1	1	1	1	1
icc_implicit_data	2	2	2	2	2
Icc_implicit_mix	3	3	3	3	3
icc_implicit_src_sink	2	2	2	2	2
icc_dynregister	2	2	2	2	2
DroidBench(IccTA)					
iac_startActivity	1	1	1	1	1
icc_startActivity	2	2	2	2	0
iac_startService	1	1	1	1	1
iac_broadCast	1	1	1	1	1
Summary	**16**	**16**	**16**	**16**	**14**

Table 4
Percentages of malware and benign apps that exhibit conditions A and B, respectively, where condition A is where the risk of the aggregate permission of the riskiest sink is greater than the risk of the sink's self permission, and condition B is where the risk of the aggregate permission of the riskiest sink is greater than the risk of (aggregated) self permissions of its corresponding sources.

	Condition A	Condition B
Malware	92%	88%
Benign	41%	40%

Table 5
A case study for sink T_1, T_2, T_3 and T_4. T_i represents the sink ID, C represents the class name, M represents the method name, F represents the function name. They have different risk scores with a same function android.util.Log: int e under different classes and methods inside an app *DroidKungFu3–1cf4d**. E2E and SS aggregations identify the same sensitive sink. T_1 is the riskiest sink with more critical taint flows and permissions.

T_i	T_1	T_2	T_3	T_4
C	com.ju6.a	uk.co.lilhermit.android.core.Native	com.adwo.adsdk.L	com.adwo.adsdk.i
M	a()	runcmd_wrapper()	a()	a()
F	Android.util.Log int e()			
$r(T_i)$	0.170	0.156	0.007	0

Table 6
Compare classification performance with two different measurements: 10-fold cross-validation and ROC curve with AUC value. Random Forest achieves highest accuracy in the four different classifiers. The detection achieves 96% accuracy for distinguishing malicious and benign apps.

	10-fold CV		ROC Curve
	F-Score	Accu	AUC value
KNN	0.88	0.88	0.9786
SVM	0.91	0.92	0.9584
D.Tree	0.94	0.94	0.9661
R.Forest	**0.96**	**0.96**	**0.9796**

quested permission is based on PSCout (Au et al., 2012). It identifies 98 distinct permissions, and builds a one-to-one projection from 15,099 distinct statements to the corresponding permissions. Permission risk value is computed based on a machine learning toolkit Sklearn. We use Androguard to extract permissions from a large set of apps. The permission risk is computed by using the

Table 7
Top risky permissions with their normalized risk values. Risk values are computed based on feature importance from Random Forest classifier. Risk values are computed by the maximization of the capability to distinguish benign and malicious apps.

Permission	Risk value
READ_PHONE_STATE	0.149
READ_SMS	0.107
RECEIVE_SMS	0.090
CHANGE_WIFI_STATE	0.080
WRITE_SMS	0.062
SEND_SMS	0.050
WRITE_CONTACTS	0.034
READ_CONTACTS	0.034
REC_BOOT_COMPLETED	0.029

App Genome dataset. It can also be computed based on Androzoo Allix et al. (2016) dataset. We choose the Genome dataset for the demonstration. The source and sink identifiers come from Susi Rasthofer et al. (2014), which categorizes a large set of critical sources and sinks. The graph analysis is based on a standard Java graph library JGraphT. Unless stated otherwise, we use E2E aggregation to evaluate the properties of malicious and benign apps. The rewriting process is based on the assemble and disassemble tool Soot. The app is automatically modified to enforce security properties and recompiled into a new application. Our evaluation is performed on 923 malicious apps from Genome dataset and 683 free popular benign apps from Google Play. The benign apps are verified via the VirusTotal inspection[7]. These apps cover different categories and contain complex code structures. As we show in Figure 1, a single app contains 11 distinct sensitive sources and 19 distinct sensitive sinks on average.

We aim to answer the following questions through our evaluation:

- **RQ1:** Can ReDroid be used to rewrite real world grayware and benchmark apps to defend vulnerabilities? (In Section 4.1).
- **RQ2:** Does the more complex E2E aggregation method provide high accuracy in ranking (In Section 4.5)?
- **RQ3:** Are the ranking results interpretable, e.g., consistent with manual validation (In Section 4.6) ?
- **RQ4:** Is ReDroid flow-aware, i.e., being able to differentiate sinks with identical method names (In Section 4.7)?
- **RQ5:** How efficient is our maximum likelihood estimation for the permission-risk mapping (In Section 4.8)?
- **RQ6:** How much is our analysis overhead (In Section 4.9)?

4.1. RQ1: rewriting apps for security

We present the feasibility of ReDroid to detect and rewrite real world grayware apps that previously have not been reported. We also demonstrate the ICC-relay based rewriting technique. Table 1 summarizes the security applications with our rewriting. We utilize benchmark apps to evaluate the feasibility of our rewriting framework. The benchmark apps are proposed by IccTA (Li et al., 2015) and AmanDroid (Wei et al., 2014) to achieve high coverage of various ICC vulnerabilities. We also use two real world grayware apps to demonstrate the possibility to use rewriting to mitigate static analysis limitations.

[7] https://www.virustotal.com/.

4.2. Benchmark suits evaluation

We evaluate our ICC relay and logging rewriting strategies on DroidBench(IccTA)[8] and ICC-Bench[9]. Apps in the ICC-Bench contain ICC-based data leak vulnerabilities. DroidBench also involves collusion apps through inter-app communications. Logging based rewriting achieves 100% success rate in both rewriting and observing the modified behaviors. The reason why logging based rewriting achieves high accuracy is that the inspection of sensitive sinks does not violate the program control and data dependences. All the rewritten apps keep valid logic (without crashing) when we run these apps with Monkey[10]. We can detect private data in the intent by inspecting the logs at runtime. It is worth to note that the logging based rewriting is easily extended to support dynamic checking. By implementing a sensitivity checking function for the logged data, our logging based rewriting can terminate the sink invocation at runtime. Therefore, the logging based rewriting is more suitable to defend privacy leak vulnerabilities in stand-alone apps.

For ICC relay rewriting, we can successfully rewrite all the apps but fail to redirect the intent in two cases. The failed two cases belong to the icc_startActivity category, where the receiver component InFlowActivity is protected and not exposed to components outside the app. Our ICC relay cannot reinvoke the receiver component from the outsider proxy app. Except the two cases, our rewriting is able to relay and redirect all the intents in the inter-app communications (IAC). Furthermore, implicit intents only specify the properties of receiver components by actions or categories. Adversarial apps can intercept implicit intents by ICC hijacking. Our ICC relay is capable to relay the implicit intent and inspect the receiver components. Therefore, the ICC relay is more suitable to defend IAC-based vulnerabilities.

4.3. Grayware I–reflection and DexClassLoad

The grayware app belongs to the game category targeting Pokemon fans. It is a puzzle game based on the Pokemon-Go app. The package called mobi.rhmjpuj.ghmjvk.sprvropjgtn appears on a third-party market (AppChina Market). VirusTotal reports it as benign[11]. However, we found multiple permissions registered in the app, e.g., WRITE_EXTERNAL_STORAGE, GET_TASKS, PHONE_STATE, SYSTEM, RESTART_PACKAGES and etc. This puzzle app is potentially risky, as it appears to request for more permissions than necessary and has dynamically loaded code (e.g., DexClassLoad) and reflection methods (e.g., Java.lang.reflection).

We use ReDroid to perform the logging-based rewriting, aiming to intercepting reflection and Dexloaded strings. For reflection, we focus on strings related to get class and method names (e.g., Class.forName and Class.getMethod) before reflect.invoke is triggered. For dynamic dex loading, we focus on strings before they are passed into system.DexClassLoader.loadClass to dynamically load classes. The sensitive string parameters are logged by ReDroid. We test the rewritten app on an emulator, using Monkey. During our execution (nearly 100 seconds), the reflection and dynamically loaded classes showed no suspicious activities.

This customization demonstrates the monitoring of Java reflection and dynamic code loading regions through rewriting. The monitoring of activities from rewritten apps can be automated with minimal human interactions with pre-defined rules and filters. App customization provides opportunities to perform dynamic monitoring of apps in production environments.

4.4. Grayware II – URL strings

The grayware app belongs to the wallpaper category targeting Pokemon-Go fans. It is a Pokemon wallpaper app. The package called com.vlocker.theme575c30395* appears on a third-party Android app market (Anzhi Market). The app was released leveraging the world-wide popularity of the Pokemon-Go app. Only 1 out of 55 anti-virus scanners reports this app as potentially risky. However, the wallpaper app contains a large number of sensitive sinks as URL.init(), file.write(), executeHttp(). It requests multiple permissions, including writing settings: WRITE_EXTERNAL_STORAGE, modifying the file system: FILESYSTEMS, intercepting calls: PROCESS_OUTGOING_CALLS, and changing network state: CHANGE_NETWORK_STATE. These permissions enable the wallpaper app to read sensitive information and modify the device state. We rewrote the URL related sink, e.g., net.URL.init(String) to log string type data before calling net.URL.openConnection(). We tested the rewritten app on an emulator, using Monkey. By analyzing the logged events, we found that private data (e.g., phone ID, IMEI) is leaked through a network request, when a user clicks on an image. Similarly as above, the monitoring of activities from rewritten apps can be automated.

The experimental results present the feasibility of the rewriting on both benchmark and real wold apps. We demonstrate two real world use cases to apply ReDroid to mitigation static analysis limitations.

4.5. RQ2: comparison of ranking accuracy

We compare our SS and E2E aggregation with the following sink-ranking metrics in terms of their accuracy in identifying the riskiest sinks. We compare our aggregation-based sink-ranking metrics with 2 basic metrics: the *in-degree* metric and the *sink-only* metric. In the *in-degree* metric, the sensitive sink's risk score is determined by its in-degree on a taint-flow graph. In the *sink-only* metric, the sensitive sink's risk score is determined by the risk of this sink's self permission.

We compare the result of the riskiest sink selection among several risk metrics. The comparison is expressed as the result consistencies, with respect to the E2E aggregation metric. For only 25% of the malware apps, the sink-only approach produces consistent riskiest sink result with E2E. This rate is higher at 47% for benign apps. The in-degree approach clearly has a very low consistency with E2E, i.e, they disagree on most rankings.

Although both SS and E2E agree to most cases, we found they disagree on long taint-flow paths that have sensitive internal nodes. Internal nodes (i.e., non-sink and non-source) on taint flows may also involve sensitive permissions. For example, in app cc.halley.droid.qwiz, a sensitive taint flow as: findVIewbyId() → getActiveNetworkInfo() → outputStream() → Log.e(). Both source findVIewbyId() and sink Log.e() are permission-insensitive, however, the sensitive internal codes on the path increases the sensitivity of the sink. getActiveNetworkInfo() is associated with permission NETWORK and outputStream() is associated with permission EXTERNAL_STORAGE. The path is risky, because the internal nodes involve critical permission. Network state information is propagated and may be potentially leaked along the path. A lack of coverage on the internal sensitive nodes introduces ranking inaccuracy. These results confirm that the comprehensive coverage of permission-requiring nodes in E2E aggregation is useful in practice.

Since E2E captures internal data flow dependences, it would be expected for E2E to achieve a higher accuracy comparing to SS, which is validated by our experiments. However, SS aggregation is still useful to balance accuracy and performance. We found E2E encounters additional 4% overhead in Section 4.9.

[8] https://github.com/secure-software-engineering/DroidBench/tree/iccta.
[9] https://github.com/fgwei/ICC-Bench.
[10] https://developer.android.com/studio/test/monkey.html.
[11] We submitted two grayware APKs to VirusTotal on Aug-10-2016.

We compare the permission propagation properties in malicious and benign apps. We consider two conditions, A and B, which are defined next. Table 4 presents the percentages of apps that exhibit such conditions. The experimental results show a large number of apps, especially malware, involve *multiple* ($\geq$ 2) sensitive permissions on taint flows. They indirectly validate the importance of flow-based permission propagation and aggregation algorithm.

Condition A is where the risk of the aggregate permission of the riskiest sink is greater than the risk of the sink's self permission.

Condition B is where the risk of the aggregate permission of the riskiest sink is greater than the risk of the (aggregated) self permissions of its corresponding sources.

4.6. RQ3: validation of sink priorities

Because of the lack of standard benchmarks[12], validating the quality of sink priorities is challenging. We perform manual inspections by comparing the riskiest sinks with the descriptions for known grayware and malware apps, to ensure our outputs are consistent and compatible with English descriptions found in security websites and articles. Due to the limited reports, we narrow down our analysis in popular ads libraries and typical malware families. These reported apps include varied behaviors, from network communications to root privilege escalations. The in-depth literature on grayware is scant, which increases the difficulty of this validation.

For grayware apps jp.co.jags and android.TigerJumping, our analysis returns the risky method net.URL located in the jp.Adlantis package. This finding is consistent with previous report stating that Adlantis libraries cause binary-classification based malware detection to fail (Tian et al., 2016).

For grayware apps org.ohny.weekend, org.qstar.guardx and uk.org.crampton.battery, our analysis returns the risky sink execute() located in an ad package com.android.Flurry. This ad library was previously reported to demonstrate excessive amounts of unauthorized operations by researchers (Elish et al., 2013).

For malware in the Geinimi family (e.g., Geinimi–037c*.apk), our analysis identifies the risky sink sendTextMessage. This sink is confirmed by a security report[13]. It is identified as a trojan to send critical messages to a premium number.

For malware in Plankton family (e.g., Plankton–5aff*.apk), our analysis returns the risky sink execute(HttpRequst) associated with aggregate permission as READ_PHONE_STATE (from a source getDeviceId()) and INTERNET. Our finding is consistent with the report of this malware, which refers to it as the spyware with background stealthy behaviors involving a remote server[14].

For malware in DroidDream (e.g., DroidDream–fed6*.apk), our analysis returns the risky sink write(byte[]) in package android.root.setting. An external report cites this malware for root privilege escalation[15]. These manual validation efforts provide the initial evidences indicating the quality of our ranking results.

4.7. RQ4: case study on sensitive taintflows

We use a real world app DroidKungFu3–1cf4d* to illustrate the importance of risk propagation. This app has four distinct sinks sharing the same method name. The method name is android.util.Log. This function requires no permission, i.e., self permission is $\emptyset$. Yet, the four sinks have different risk scores computed by our risk aggregation procedure. Table 5 presents the four sinks with their risk scores.

The sink with the highest risk score involves three distinct permissions READ_PHONE_STATE, LOCATION and INTERNET. The sources getLine1Number(), getDeviceId(), getSubscribeId() and getSimSerialNumber() are related to READ_PHONE_STATE permission. The source getLastKnownLocation() and the internal node getLongitude() are related to LOCATION permission. The execute(HttpUriRequest) and openConnection() are related to INTERNET permission. getIntent() requires no permission. Although these sinks share the same function name, the riskiest sink T_1 involves more critical paths than the others.

- T_1: getLastKnownLocation() $\rightarrow$ getLongitude() $\rightarrow$ T_1, getLine1Number() $\rightarrow$ T_1, getDeviceId() $\rightarrow$ T_1, getSubscribeId() $\rightarrow$ T_1, getSimSerialNumber() $\rightarrow$ T_1, execute(Http) $\rightarrow$ T_1.
- T_2: execute(HttpUriRequest) $\rightarrow$ T_2, getLine1Number() $\rightarrow$ T_2, getDeviceId() $\rightarrow$ T_2.
- T_3: openConnection() $\rightarrow$ T_3.
- T_4: getIntent() $\rightarrow$ T_4.

4.8. RQ5: quality of likelihood estimation

Our machine learning techniques enable to compute risk scores of permissions, which maps one particular permission to a quantitative and computable score value. We test four different machine learning approaches: Support Vector Machine (SVM), k-nearest neighbors (KNN), Decision Tree (D.Tree) and Random Forest (R.Forest). The dataset is originally labeled for Android malware classification (Tian et al., 2016). We reuse the dataset for our risk score computation. The benign apps are collected from official app market Google Play. The malicious apps are selected from popular malware database Genome and VirusShare. It is worth to note that our machine learning technique is aimed to compute the risk scores of permissions, not for malware classification. The permissions of an app are transformed into features for each classifier. Each permission corresponds to a certain position in a feature vector, where 1 means the app registers for this permission and 0 means the app does not register for this permission. We apply two standard evaluation measurements: 10-fold cross-validation and ROC curve. 10-fold cross-validation divides the dataset into 10 portions. Each time, the 9 portions of them are used as the training set and the rest of the data is used as the testing set.

We compute the average accuracy rate and F-score to evaluate these classifiers. Receiver operating characteristic (ROC) curve draws a statistic curve and computes an area under curve (AUC) value. A higher AUC value represents a better classification capacity.

The experimental results validate the hypothesis that permissions are useful as the features to distinguish the benign and malicious apps. Random Forest maximizes the accuracy in the classification.

Table 6 presents the detection accuracy of four different classifiers. Random Forest achieves the highest accuracy and AUC value among these four classifiers. In ReDroid, we calculate the risk value for each permission in the random forest classifier. Table 7 presents top risky permissions with their normalized risk values. We focus on the permissions that are related to private data and phone state reading. Specifically, READ_PHONE_STATE achieves highest risk value as 0.149. The reason why READ_PHONE_STATE is most sensitive is because it enables an app to access private phone information, e.g., device Id and current phone state. Malicious apps

[12] We aim to release our dataset as a benchmark.

[13] https://nakedsecurity.sophos.com/2010/12/31/geinimi-android-trojan-horse-discovered/.

[14] https://www.csc.ncsu.edu/faculty/jiang/Plankton/.

[15] https://blog.lookout.com/droiddream/.

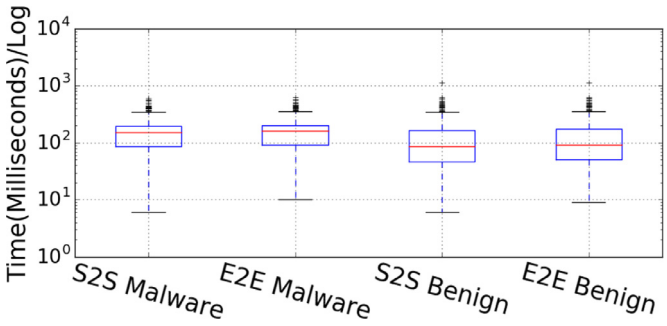


Fig. 4. Runtime of permission propagation in Algorithm 1 on malware and benign apps under SS and E2E aggregation functions, respectively. Both aggregation methods have a low average runtime of around 0.1 second, with E2E aggregation slightly slower than SS.

abuse this permission for collecting privacy information. These sensitive permissions have higher risk values, because they are associated with malicious behaviors. In our quantitative analysis, the risk values of permissions are used as the input for initialization of sensitive nodes.

4.9. RQ6: analysis overhead

We compare the runtime of Algorithm 1 under two SS and E2E aggregations in Fig. 4.[16] Experiments were performed over both benign and malware datasets on a Linux machine with Intel Xeon CPU (@3.50GHz) and 16G memory. Fig. 4 presents the four runtime distributions in log scale. The runtime is focusing on the permission propagation analysis with the input of the transitive reduced graph and the output of sorted sinks. Both E2E and SS aggregations have a similar low overhead of around 0.1 second. E2E has an additional 4% overhead than SS on average. The average runtime of malware is larger than that of benign apps, because malware apps typically have more sensitive sinks and complex graph structures. The performance results confirm the efficiency of our graph algorithm.

We evaluate rewriting performance based on the file size overhead. The benchmark apps come from DroidBench and ICC-Bench in Section 4.1). On average, both logging and ICC relay based rewriting achieves <1 % size overhead, which is relatively negotiable. Our approach is very efficient in rewriting benchmark apps. We also discuss the sources that introduce size overhead in practical rewriting scenarios. 1) The complexity of rewriting. If the rewriting strategy is very complex, e.g., dynamic checking with multiple conditions, we need to implement more rewriting functions. 2) The number of impacted code in rewriting. If we need to rewrite a large number of sinks in an app, the rewriting overhead increases significantly. Therefore, with the sensitive sink prioritization, we could optimize the number of sinks for rewriting based on the sensitivity ranking.

4.10. Summary of experimental findings

We summarize our major experimental findings as follows.

1) We give multiple demonstrations of app customization for security, including inter-app ICC relay and logging. We successfully detect and rewrite recently released Pokemon-Go related grayware, which enables the monitoring of runtime activities involving Java reflection and dynamic code loading and URL strings.
2) Our risk-ranking algorithm is efficient for real world apps. Given a taint-flow graph, our graph algorithm with E2E aggregation has an additional 4% overhead than the SS aggregation, but both can complete within 0.1 second for most real world apps.
3) Manual inspections show that our risk ranking results are consistent with the English descriptions of apps, for a small set of malware apps and grayware apps. This consistency indicates the effectiveness of sink prioritization algorithms.
4) SS and E2E aggregations are consistent in finding the riskiest sinks on most apps, with E2E being slightly more comprehensive for long tainted flows with sensitive internal nodes. They substantially outperform sink-only and in-degree approaches. Malware sinks have more aggregate permissions and risk scores than those of benign apps. In 92% of malicious apps, the aggregate permission of the riskiest sink is greater than the risk of the sink's self permission, only in 41% of benign apps, the aggregate permission of the riskiest sink is greater than the risk of the sink's self permission.

5. Related work

5.1. Android taint flow analysis

The vulnerability of apps can be abused by attackers for privilege escalation and privacy leakage attacks (Bugiel et al., 2012). Researchers proposed taint flow analysis to discover sensitive dataflow paths from sources to sinks. CHEX (Lu et al., 2012) and AndroidLeaks (Gibler et al., 2012) identified sensitive data flows to mitigate apps' vulnerability. Bastani et al. described a flow-cutting approach (Bastani et al., 2015). However, their work only provides theoretical analysis on impacts of a cut, without any implementation. DroidSafe (Gordon et al., 2015) used a point-to graph to identify sensitive data leakage. FlowDroid (Arzt et al., 2014) proposed a static context- and flow-sensitive program analysis to track sensitive taint flows. These solutions address the privacy leakage by tracking the usage of privacy information. Our sink ranking is based on static analysis and our prototype utilizes FlowDroid.

5.2. Android rewriting

The app-retrofitting demonstration in RetroSkeleton (Davis and Chen, 2013) aims at automatically updating HTTP connections to HTTPS. Aurasium (Xu et al., 2012) instruments low-level libraries for monitoring functions. Reynaud et al. (2012) rewrote an app's verification function to discovered vulnerabilities in the Android in-app billing mechanism. AppSealer Zhang and Yin (2014) proposed a rewriting solution to mitigate component hijacking vulnerabilities, the rewriting is to generate patches for functions with component hijacking vulnerabilities. Fratantonio et al. (2015) used rewriting to enforce secure usage of the Internet permission. Because of the special goal on INTERNET permission, the rewriting option cannot be applied to general scenarios. The rewriting targets and goals in these tools are specific. Furthermore, our rewriting is more feasible than existing rewriting frameworks by supporting both ICC-level and sink-based rewriting with data flow analysis.

5.3. Malware detection with quantitative reasoning

Our work is also related to malware classification with quantitative reasoning. Researchers Wüchner et al. (2014) and Wuchner et al. (2015) regarded the quantitative value among difference processes as the total number of transferred resources based on the OS-level system logs. These numbers are used to better distinguish malicious and benign processes.

[16] Runtime measured excludes FlowDroid and maximum likelihood estimation.

PRIMO (Octeau et al., 2016) used probabilities to estimate the likelihood of implicit ICC communications. The triage of ICC links is based on the true positive likelihood of links. MR-Droid (Liu et al., 2017) measured inter-app communication properties with static analysis. DIAL-Droid (Bosu et al., 2017) performed static analysis on millions of apps to discover suspicious ICC link communications. DroidCat (Cai et al., 2018) utilized app-level profiling to identify malicious behaviors. Peng et al. (2012) used permissions to detect Android malware. The permission risk values are generated from probabilistic Bayesian-Network models. In contrast, we compute permission risk values by maximizing the classifier's capacity of detecting malicious and benign apps. The risk value computation in our approach associates a permission's correlation to malicious apps. These approaches are not compatible with risky-sink-guided rewriting as they are not designed for security customization of off-the-shelf apps. In our model, sensitive sinks are prioritized based on the aggregate risk scores. Our analysis is focused on quantitatively ranking different sensitive sinks. Our results validate the effectiveness of ranking sinks with machine-learning-based risk value computation and graph-based permission propagation.

5.3.1. Defense of vulnerabilities

Grayware or malware with vulnerabilities can result in privacy leakage. Pluto (Demetriou and Merrill, 2016) discovered the vulnerabilities of the abuse in ads libraries. In order to defend vulnerabilities, many approaches have been proposed to track dynamic data transformation or enforce security policy. TaintDroid (Enck et al., 2014) adopted dynamic taint analysis to track the potential misuse of sensitive data in Android apps. CrypoGuard (Rahaman et al., 2019) used static slicing to identify security vulnerabilities. Elish et al. (2018) used static program analysis to approximate suspicious inter-application communication vulnerabilities. Merlin (Banerjee et al., 2009) used path constraints to infer explicit information specifications to identify security violations. AspectDroid (Ali-Gombe et al., 2016) used static instrumentation and automated testing to detect malicious activities. We demonstrate the defense of vulnerabilities by rewriting apps in the experiments. Our quantitative rewriting is operated on application level with rewriting. We rank flow-based sinks by the graph propagation with permission-based risk values. We specialize different rewriting rules to defend vulnerabilities.

5.3.2. Program repairing

Program repairing is related to our work since it provides solutions for generating patches. The patches are used to identify bugs for program repairing. GenProg (Le Goues et al., 2012; Weimer et al., 2009) used genetic programming algorithms to discover patches that lead to bugs. PAR (Kim et al., 2013) used human-defined patch templates to learn patterns for fixing bugs. Prophet (Long and Rinard, 2016) used a probabilistic model to characterize the properties of correct code patches. The trained model is used to detect defects in real world apps. Our approach differs from these approaches in the model design. Our approach is designed for enhancing Android specific security with rewriting. Our approach enforces security properties on the sensitive sinks from the computation of graph-based permission propagation.

6. Conclusions and future work

In this paper, we present two new technical contributions for Android security, a quantitative risk metric for evaluating sensitive flows and sinks, and a risk propagation algorithm for computing the risks. We implement a prototype called ReDroid, and demonstrate the feasibility of both ICC-relay and logging-based rewriting techniques.

ReDroid is a tool for (1) quantitatively ranking sensitive data flows and sinks of Android apps and (2) customizing apps to enhance security. Our work is motivated by apps that appear mostly benign but with some security concerns, e.g., risky flows incompatible with organizational policies, aggressive ad libraries, or dynamic code that cannot be statically reasoned. We extensively evaluated and demonstrated how sink ranking is useful for rewriting grayware to improve security. Our risk metrics are more general and can be applied in multiple security scenarios. For future research, we plan to focus on supporting automatic rewriting with flexible security policy specifications.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

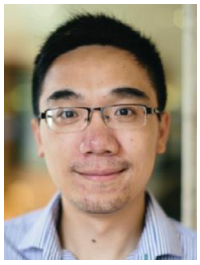
The authors would like to thank the anonymous reviewers for their insightful comments and suggestions on the work. This project was supported in part by NSF grant CNS-1717028.

References

Afonso, V., Bianchi, A., Fratantonio, Y., Doupé, A., Polino, M., de Geus, P., Kruegel, C., Vigna, G., 2016. Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy. In: Proc. of NDSS.
Aho, A.V., Garey, M.R., Ullman, J.D., 1972. The transitive reduction of a directed graph. SIAM J. Comput. 131–137.
Ali-Gombe, A., Ahmed, I., Richard III, G.G., Roussev, V., 2016. AspectDroid: Android app analysis system. In: Proc. of CODASPY.
Allix, K., Bissyandé, T.F., Klein, J., Le Traon, Y., 2016. Androzoo: Collecting millions of android apps for the research community. In: Proc. of MSR.
Arp, D., Spreitzenbarth, M., Hübner, M., Gascon, H., Rieck, K., Siemens, C., 2014. Drebin: Effective and explainable detection of Android malware in your pocket. In: Proc. of NDSS.
Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P., 2014. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In: Conference on Programming Language Design and Implementation (PLDI).
Au, K.W.Y., Zhou, Y.F., Huang, Z., Lie, D., 2012. PScout: analyzing the Android permission specification. In: Proc. of CCS.
Banerjee, A., Livshits, B., Nori, A.V., Rajamani, S.K., 2009. Merlin: Specification inference for explicit information flow problems. In: Proc. of PLDI.
Bastani, O., Anand, S., Aiken, A., 2015. Interactively verifying absence of explicit information flows in Android apps. In: Proc. of OOPLSA.
Bosu, A., Liu, F., Yao, D.D., Wang, G., 2017. Collusive data leak and more: Large-scale threat analysis of inter-app communications. In: Proc. of AisaCCS.
Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A.-R., Shastry, B., 2012. Towards taming privilege-escalation attacks on Android.. In: Proc. of NDSS.
Cai, H., Meng, N., Ryder, B., Yao, D., 2018. DroidCat: Effective android malware detection and categorization via app-level profiling.
Chin, E., Felt, A.P., Greenwood, K., Wagner, D., 2011. Analyzing inter-application communication in Android. In: Proc. of MobiSys.
Davis, B., Chen, H., 2013. RetroSkeleton: Retrofitting Android Apps. In: Proc. of MobiSys.
Davis, B., Sanders, B., Khodaverdian, A., Chen, H., 2012. I-ARM-Droid: A rewriting framework for in-app reference monitors for Android applications. In: Proc. of MoST.
Elish, K., Cai, H., Barton, D., Yao, D., Ryder, B., 2018. Identifying mobile inter-app communication risks.
Elish, K.O., Shu, X., Yao, D.D., Ryder, B.G., Jiang, X., 2015. Profiling user-trigger dependence for Android malware detection. Comput. Secur. 255–273.
Elish, K.O., Yao, D.D., Ryder, B.G., Jiang, X., 2013. A static assurance analysis of Android applications. Technical Report.. Department of Computer Science.
Enck, W., Gilbert, P., Han, S., Tendulkar, V., Chun, B.-G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N., 2014. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. ACM Trans. Comput. Syst. (TOCS).
Enck, W., Ongtang, M., McDaniel, P., 2009. On lightweight mobile phone application certification. In: Proc. of CCS.
Fratantonio, Y., Bianchi, A., Robertson, W., Egele, M., Kruegel, C., Kirda, E., Vigna, G., Kharraz, A., Robertson, W., Balzarotti, D., et al., 2015. On the security and engineering implications of finer-grained access controls for Android developers and users. In: Proc. of DIMVA.
Gibler, C., Crussell, J., Erickson, J., Chen, H., 2012. AndroidLeaks: Automatically detecting potential privacy leaks in Android applications on a large scale. In: Proc. of Trust and Trustworthy Computing.

Gordon, M.I., Kim, D., Perkins, J., Gilham, L., Nguyen, N., Rinard, M., 2015. Information-flow analysis of Android applications in DroidSafe. In: Proc. of NDSS.
Kim, D., Nam, J., Song, J., Kim, S., 2013. Automatic patch generation learned from human-written patches. In: Proc. of ICSE.
Le Goues, C., Dewey-Vogt, M., Forrest, S., Weimer, W., 2012. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In: Proc. of ICSE.
Li, L., Bartel, A., Bissyandé, T.F., Klein, J., Le Traon, Y., Arzt, S., Rasthofer, S., Bodden, E., Octeau, D., McDaniel, P., 2015. IccTA: Detecting inter-component privacy leaks in android apps. In: Proceedings of the 37th International Conference on Software Engineering-Volume 1.
Liu, F., Cai, H., Wang, G., Yao, D., Elish, K.O., Ryder, B.G., 2017. MR-Droid: A scalable and prioritized analysis of inter-app communication risks. In: Proc. of MoST.
Long, F., Rinard, M., 2016. Automatic patch generation by learning correct code. In: Proc. of POPL.
Lu, L., Li, Z., Wu, Z., Lee, W., Jiang, G., 2012. CHEX: statically vetting Android apps for component hijacking vulnerabilities. In: Proc. of CCS.
Octeau, D., Jha, S., Dering, M., McDaniel, P., Bartel, A., Li, L., Klein, J., Le Traon, Y., 2016. Combining static analysis with probabilistic models to enable market-scale Android inter-component analysis. In: Proc. of POPL.
Peng, H., Gates, C., Sarma, B., Li, N., Qi, Y., Potharaju, R., Nita-Rotaru, C., Molloy, I., 2012. Using probabilistic generative models for ranking risks of Android apps. In: Proc. of CCS.
Rahaman, S., Xiao, Y., Afrose, S., Shaon, F., Tian, K., Frantz, M., Kantarcioglu, M., Yao, D.D., 2019. CryptoGuard: High precision detection of cryptographic vulnerabilities in massive-sized java projects.. In: Proc. of CCS.
Rasthofer, S., Arzt, S., Bodden, E., 2014. A machine-learning approach for classifying and categorizing Android sources and sinks. In: Proc. of NDSS.
Reynaud, D., Song, D.X., Magrino, T.R., Wu, E.X., Shin, E.C.R., 2012. FreeMarket: Shopping for free in Android applications.. In: Proc. of NDSS.
Demetriou, S., Merrill, W., Yang, W., Zhang, A., Gunter, C.A., 2016. Free for all! assessing user data exposure to advertising libraries on Android. In: Proc. of NDSS.
Tam, K., Feizollah, A., Anuar, N.B., Salleh, R., Cavallaro, L., 2017. The evolution of android malware and android analysis techniques. In: Proc. of ACM Computing Surveys (CSUR).
Tian, K., Tan, G., Yao, D., Ryder, B., 2017. ReDroid: Prioritizing data flows and sinks for app security transformation.. In: Proc. of FEAST, collocated with the ACM Conference on Computer and Communications Security (CCS).
Tian, K., Yao, D.D., Ryder, B.G., Tan, G., 2016. Analysis of code heterogeneity for high-precision classification of repackaged malware. In: Proc. of MoST.
Wei, F., Roy, S., Ou, X., et al., 2014. AmanDroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In: Proc. of CCS.
Weimer, W., Nguyen, T., Le Goues, C., Forrest, S., 2009. Automatically finding patches using genetic programming. In: Proc. of ICSE.
Wüchner, T., Ochoa, M., Pretschner, A., 2014. Malware detection with quantitative data flow graphs. In: Proc. of AsiaCCS.
Wuchner, T., Ochoa, M., Pretschner, A., 2015. Robust and effective malware detection through quantitative data flow graph metrics. In: Proc. of DIMVA.
Xu, R., Saïdi, H., Anderson, R., 2012. Aurasium: Practical policy enforcement for Android applications. In: Proc. of USENIX Security.
Zhang, M., Yin, H., 2014. AppSealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in Android applications.. In: Proc. of NDSS.
Zhou, Y., Jiang, X., 2012. Dissecting Android malware: Characterization and evolution. In: Proc. of IEEE (S&P).

Ke Tian is a PhD candidate in Department of Computer Science at Virginia Tech, Blacksburg. He received his bachelor degree majoring information security from University of Science and Technology of China in 2013. He received the National Scholarship of China in 2012. His research interests is in Cybersecurity, Mobile security and machine learning.



Dr. Gang Tan is the James F. Will Career Development Associate Professor in the Department of Computer Science and Engineering at the Pennsylvania State University, University Park, PA. He leads the Security of Software (SOS) Lab. His research is at the interface between computer security, programming languages, and formal methods. He received his bachelor's degree in Computer Science with honors from Tsinghua University in 1999 and his Ph.D. degree from Princeton University in 2005. He has received an NSF CAREER award, two Google Research Awards, and a Francis Upton Graduate Fellowship. He is a member of IEEE and ACM.

Dr. Barbara G. Ryder is a emerita faculty member in the Department of Computer Science at Virginia Tech, where she held the J. Byron Maupin Professorship in Engineering. She received her A.B. degree in Applied Mathematics from Brown University (1969), her Masters degree in Computer Science from Stanford University (1971) and her Ph.D. degree in Computer Science at Rutgers University (1982). From 2008-2015 she served as Head of the Department of Computer Science at Virginia Tech, and retired on September 1, 2016. Dr. Ryder served on the faculty of Rutgers from 1982-2008. She also worked in the 1970s at AT&T Bell Laboratories in Murray Hill, NJ. Dr. Ryder's research interests on static/dynamic program analyses for object-oriented and dynamic programming languages and systems, focus on usage in practical software tools for ensuring the quality and security of industrial-strength applications. Dr. Ryder became a Fellow of the ACM in 1998, and received the ACM SIGSOFT Influential Educator Award (2015), the Virginia AAUW Woman of Achievement Award (2014), and the ACM President's Award (2008). She received a Rutgers School of Arts and Sciences Computer Science Distinguished Alumni Award (2016), was named a CRA-W Distinguished Professor (2004), and was given the ACM SIGPLAN Distinguished Service Award (2001). Dr. Ryder led the Department of Computer Science team that tied nationally for 2nd place in the 2016 NCWIT NEXT Awards.She has been an active leader in ACM (e.g., Vice President 2010-2012, Secretary-Treasurer 2008-2010; ACM Council 2000-2008; General Chair, FCRC 2003; Chair ACM SIGPLAN (1995-97)). She serves currently as a Member of the Board of Directors of the Computer Research Association (2014-2020,1998-2001). Dr. Ryder is an editorial board member of ACM Transactions on Software Engineering Methodology and has served as an editorial board member of ACM Transactions on Programming Languages and Systems, IEEE Transactions on Software Engineering, Software: Practice and Experience, and Science of Computer Programming. Dr. Ryder led the Department of Computer Science at Virginia Tech team that tied nationally for 2nd place in the 2016 NCWIT NEXT Awards. She was a founding member of the NCWIT VA/DC Aspirations in Computing Awards. Dr. Ryder has advised 16 Ph.D. and 3 M.S. students to completion of their theses; she has supervised the research of 4 post-docs and more than 30 undergraduate researchers at Rutgers and Virginia Tech.

Daphne Yao is an associate professor of computer science at Virginia Tech. In the past decade, she has been working on designing and developing data-driven anomaly detection techniques for securing networked systems against stealthy exploits and attacks. Her expertise also includes mobile security. Dr. Yao received her Ph.D. in Computer Science from Brown University. Dr. Yao is an Elizabeth and James E. Turner Jr. '56 Faculty Fellow and L-3 Faculty Fellow. She received the NSF CAREER Award in 2010 for her work on human-behavior driven malware detection, and the ARO Young Investigator Award for her semantic reasoning for mission-oriented security work in 2014. She has several Best Paper Awards (e.g., ICNP '12, CollaborateCom '09, and ICICS '06) and Best Poster Awards (e.g., ACM CODASPY '15). She was given the Award for Technological Innovation from Brown University in 2006. She held multiple U.S. patents for her anomaly detection technologies. Dr. Yao is an associate editor of IEEE Transactions on Dependable and Secure Computing (TDSC). She serves as PC members in numerous computer security conferences, including ACM CCS. She has over 75 peer-reviewed publications in major security and privacy conferences and journals.