ReDroid: Prioritizing Data Flows and Sinks for App Security Transformation

Ke Tian[1], Gang Tan[2], Danfeng (Daphne) Yao[1], Barbara G. Ryder[1]
[1]Department of Computer Science, Virginia Tech
[2]Department of Computer Science and Engineering, Penn State University
ketian@cs.vt.edu,gtan@cse.psu.edu,{danfeng,ryder}@cs.vt.edu

ABSTRACT

Security transformation is to transfer applications to meet security guarantees. How to prioritize Android apps and find suitable transformation options is a challenging problem. 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. We present an efficient graph-algorithm based risk metric for prioritizing risky flows and sinks in Android grayware apps. Our risk prioritization produces orderings that are consistent with published security reports.

We demonstrate a new automatic app transformation framework that utilizes the above prioritization technique to improve app security. The framework provides more rewriting options than the state-of-art solutions by supporting flow- and sink-based security checks. Our prototype ReDroid is designed for security analysts who manage organizational app repositories and customize third-party apps to satisfy organization imposed security requirements. Our framework enables application transformation for both benchmark apps and real-world grayware to strengthen their security guarantees.

CCS CONCEPTS

• **Security and privacy → Software security engineering**; *Software security engineering*; *Software reverse engineering*;

KEYWORDS

Android Rewriting, Security Transformation, Sink Prioritizing

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., [20, 23–25]). In addition, the use of machine-learning methods enables automatic malware recognition based on multiple data-flow features (e.g., [9, 28, 29, 32]).

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. 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. For example, the DoD (Department of Defense) has its own private app store for DoD employees [4]. 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)* **[Transformation]** to modify the code and repackage the app. In addition, post-rewrite monitoring may be needed, if the access or 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.

Existing app rewriting solutions for Android are limited in several aspects. These solutions are specific to certain code issues and are not designed for general-purpose security customization. For example, Davis and Chen performed an HTTP-specific rewriting that ensures the use of HTTPS in the retrofitted apps [18]. Rewriting for the Internet permission check has also been performed [22]. AppSealer [34] proposed a rewriting solution to mitigate component hijacking vulnerabilities. Due to the specific rewriting needs, the target locations to be rewritten are relatively straightforward to

FEAST'17, November 3, 2017, Dallas, TX, USA
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5395-3/17/11…$15.00
https://doi.org/10.1145/3141235.3141239

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. We found that it is not uncommon for real-world apps to have more than 100 sensitive flows (tainted flows based on SuSi labeling). If additional post-rewrite monitoring is required at runtime, then modifying *every single* sensitive flow or sink may substantially slow down the performance. Because 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.

In this paper, we demonstrate a new security customization technique for Android apps. 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.

In addition to sink prioritization, we also demonstrate a new and automatic code rewriting technique that can verify and terminate the riskiest sink *at runtime*. For the Android-specific 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 new sink-ranking algorithm that is useful for prioritizing sensitive data flows in Android apps. Our algorithm 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 new app transformation framework **ReDroid** [1]. We use ReDroid to demonstrate the usage of rewriting in defending ICC hijacking and privacy leak vulnerabilities. Our rewriting shows high flexibility, e.g., *all* the benchmark apps are successfully rewritten for log monitoring. Our rewriting supports automatic flow-based and sink-based rewriting.
(3) We performed an extensive experimental evaluation on the validity of permission risks and sink rankings. Our inspection indicates that top risky sinks found by ReDroid are consistent with external security reports. We demonstrate the effectiveness of both inter-app ICC relay and logging-based rewriting techniques. Our rewriting is practical by successfully rewriting benchmark apps and real-world grayware. The transformed app enables one to monitor runtime activities involving Java reflection, dynamic code loading, and URL strings.

We automate the sink prioritization and app transformation via rewriting. Our ranking algorithm supports both *sink ranking* and *flow ranking*. 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 more risky than a single flow. Thus, we focus on rewriting sinks.

2 OVERVIEW

We first show a few examples to motivate the needs for ranking sensitive data flows and rewriting apps for security.

Security Transformation via App Rewriting. Table 1 summarizes the security applications with our rewriting. Our rewriting identifies multiple vulnerabilities such as ICC hijacking and privacy leak. We rewrite apps to enforce 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.

Automation of Prioritization and Rewriting. The prioritization and rewriting of an app are both automatic. During the prioritization analysis, our approach reports sensitive sinks, along with their risk values and taint flows. Based on the analysis results, security analysts could customize rewriting strategies for their interesting sinks, e.g., selecting top-k sinks. The selected sinks are automatically rewritten by our framework. The app is recompiled with the security policy enforced for post-rewrite monitoring.

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 Genome Database [35]. We found a single app can contain more than 20 distinct sinks. On average, an app contains more than 100 sensitive flows. 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.

A Toy Example. In Figure 1, 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

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

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

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.

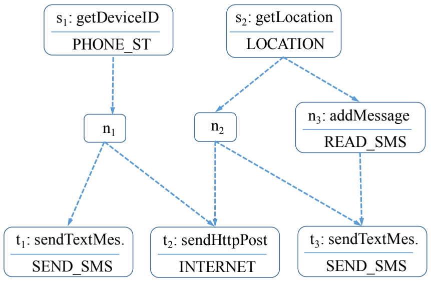

Figure 1: An example of a taint-flow graph. Nodes represent function calls or instructions. Permissions (shown at the bottom of a node) associated with the functions (shown at the top of a node) are shown. Directed edges represent data dependence relations.

t_3) and several internal nodes, one of which involves a sensitive function. 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 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 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 Figure 1 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.

2.1 Workflow

We briefly describe our workflow with key 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 risks, 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 3.2.
(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.

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.

Next, we describe technical details of our operations. We present risk propagation in Section 3.1, permission mapping in Section 3.2 and rewriting in Section 3.3.

3.1 Risk Propagation

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

Graph Construction We use Android-specific static program analysis 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 $G(V, E, S, T)$. We then apply transitive reduction transforms $G(V, E, S, T)$ into $G'(V, E', S, T)$.

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.

Rewriting Granularity	RetroSkeleton [18] and [10][19]	ReDroid **(Ours)**
Package-level (Repackage)	✓	✓
Class-level (Class Inject)	✓	✓
Method-level (Method Invoc.)	✓	✓
ICC-level (Intent Redirect)	–	✓
Prioritization-based Rewriting	–	✓

Table 2: Comparison of ReDroid with existing Android bytecode rewriting frameworks. Method invoc. is short for method invocation to invoke a customized method instead of an original method. RetroSkeleton is the standard rewriting framework for Android apps. Our rewriting supports more Android-specific rewriting strategies than RetroSkeleton.

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.

3.2 Permission-Risk Mapping

We follow a maximum likelihood estimation approach [31, 33], 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 machine learning 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).

We use the feature-importance value of a permission as a security measurement for the permission sensitivity. An important permission (e.g., READ_SMS) is an indicator of a sensitive app from empirical studies [9]. A permission (e.g., INTERNET) with low sensitivity has a low importance value. Our method automatically maps a permission string into a quantitative risk value.

3.3 Application Security Transformation

The innovation of our bytecode rewriting is to support Android-specific ICC and sink prioritization analysis. Our rewriting addresses Android vulnerabilities and static analysis limitations to aid security analysts. Existing rewriting solutions do not support our specific rewriting requirements. Table 2 presents the comparison of ReDroid with existing Android rewriting frameworks [2]. Unlike previous rewriting demonstrations on Smali (such as [18, 30]), our inter-app ICC relay rewriting approach requires more substantial technical efforts.

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.

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 inspect the data before it is consumed (e.g., sent out). Our redirection mechanism leverages Android-specific inter-component communication (ICC). Android ICC mechanism enables the communication among different apps [15, 17].

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.

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.

4 EXPERIMENTAL EVALUATION

We extend FlowDroid [11] for our static program analysis. FlowDroid is the most advanced and popular tool for static taint flow construction. Our mapping from a statement into the requested permission is based on PSCout [12]. It identifies 98 distinct permissions, and builds a one-to-one projection from 15,099 distinct statements to the corresponding permissions. The new permission risk value is computed based on our machine learning algorithms. The source and sink identifiers come from Susi [26], which categorizes a large set of critical sources and sinks. Unless stated otherwise, we use E2E aggregation to evaluate the properties of analyzed apps.

We build our own rewriting framework to support the prioritization technique. The app is automatically rewritten and recompiled into a new app. To evaluate the rewriting efficiency, we test both benchmark apps and real-word grayware. To explore sink properties, we test on 923 apps with suspicious behaviors from Genome dataset and 683 free popular benign apps from Google Play dataset.

[2] Based on positive feedbacks from security analysts and developers, we aim to release our tool as a free software. We will provide our project site after acceptance.

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 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.

The benign apps are verified via the VirusTotal inspection [8]. These apps cover different categories and contain complex code structures.

We aim to answer the following questions through our preliminary evaluation: **RQ1:** Can ReDroid be used to transform real-world grayware and benchmark apps to defend vulnerabilities? (In Section 4.1). **RQ2:** Are the ranking results reasonable and validate (In Section 4.2) ?

4.1 RQ1: Security Improvement

We present the feasibility of ReDroid to detect and rewrite real-world grayware apps that previously have not been reported. Table 1 summarizes the security applications with our rewriting. We utilize benchmark apps to evaluate the efficiency of our rewriting framework. We also use two grayware examples to demonstrate how to use rewriting for grayware analysis.

Benchmark Suits Evaluation. We evaluate our ICC relay and logging rewriting strategies on DroidBench(IccTA)[6] and ICC-Bench [5]. 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 on a real-world device Nexus 6P. We can detect private data (e.g., intent actions and extra data fields) in the intent by inspecting the logs. 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 rewriting can terminate the sink invocation at runtime to prevent data leakage. 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 are 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.

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). We submitted two grayware APKs to VirusTotal on Aug-10-2016. VirusTotal reports it as benign. 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 a real-world device Nexus 6P. We use Monkey [7] for automatic user interaction with the app. The real-world grayware is more complex than benchmark apps, we also manually interact with the app if Monkey cannot reach the modified code. 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 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.

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(). 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 demonstrate the effectiveness of our approach in rewriting benchmark apps and real-world grayware. Our rewriting enables security analysts to customize and monitor apps for security guarantees.

4.2 RQ2: Validation of Sink Priorities

Because of the lack of benchmarks, validating the quality of sink priorities is challenging. Indeed, we aim to release our dataset as a benchmark. We compare the riskiest sinks from our analysis with the descriptions for known grayware and malware apps, to ensure our outputs are consistent and compatible with the findings in security websites and articles. These findings are generated by security analysts with substantially manual efforts. The validation shows that our automatic sink prioritization results are highly consistent with manual security analysis. 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 [28].

For grayware apps org.ohny.weekend, org.qstar.guardx and uk.or g.battery, our analysis returns a 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 [21].

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 [2]. 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 [3].

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 [1].

These validation provides the initial evidences indicating the quality of our ranking results. Our sink prioritization is recognized by the security analysis reports. We envision our automatic approach would be a helpful tool to aid security analysts, by reducing their manual efforts on finding most sensitive sinks and offering multiple rewriting options.

Summary of Experimental Findings. We summarize our major experimental findings as follows.

(1) We give multiple demonstrations of app security transformation, including inter-app ICC relay and logging. Our rewriting with sink ranking is a practical yet efficient tool for application security transformation.
(2) Manual inspections show that our risk ranking results are consistent with the security analysts, for a small set of malware apps and grayware apps. This consistency indicates the effectiveness of sink prioritization algorithms.

5 RELATED WORK

Android Taint Flow Analysis The vulnerability of apps can be abused by attackers for privilege escalation and privacy leakage attacks [16]. Researchers proposed taint flow analysis to discover sensitive data-flow paths from sources to sinks. CHEX [25] and AndroidLeaks [23] identified sensitive data flows to mitigate apps' vulnerability. Bastani *et al.* described a flow-cutting approach [14]. However, their work only provides theoretical analysis on impacts of a cut, without any implementation. DroidSafe [24] used a point-to graph to identify sensitive data leakage. FlowDroid [11] 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.

Android Rewriting The app-retrofitting demonstration in RetroSkeleton [18] aims at automatically updating HTTP connections to HTTPS. Aurasium [30] instruments low-level libraries for monitoring functions. Reynaud *et al.* [27] rewrote an app's verification function to discovered vulnerabilities in the Android in-app billing mechanism. AppSealer [34] proposed a rewriting solution to mitigate component hijacking vulnerabilities, the rewriting is to generate patches for functions with component hijacking vulnerabilities. Fratantonio *et al.* [22] 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 general than existing rewriting frameworks by supporting both ICC-level and sink-based rewriting with data flow analysis. ARTist [13] is a compiler-base rewriting tool to modify Android runtime virtual machine. However, compiler-based rewriting relies on a particular Android version and introduces high runtime overhead. Our bytecode rewriting modifies an app's bytecode and is compatible for all the Android versions. Bytecode rewriting is more suitable for security analysts to rewrite apps and enforce organization policies. Based on the static code analysis, we also provide rich contexts (e.g., sink prioritization) for generating rewriting options.

6 CONCLUSIONS AND FUTURE WORK

In this paper, we present two new technical contributions, a quantitative risk metric for evaluating sensitive flows and sinks, and a risk propagation algorithm for computing the risks. We also design a new rewriting framework ReDroid, for Android-specific app customization. We demonstrate the feasibility of both ICC-relay and logging-based rewriting techniques in our experiments. Redroid aids security analysts for practical grayware analysis and monitoring. We extensively demonstrated how sink ranking is useful for rewriting grayware to improve security. Rewriting with sink prioritization is a promising solution for application security transformation.

REFERENCES

[1] Android Root privilege. https://blog.lookout.com/droiddream/. Accessed: 2017-05-02.

[2] Android trojan genimi. https://nakedsecurity.sophos.com/2010/12/31/geinimi-android-trojan-horse-discovered/. Accessed: 2017-05-02.

[3] Android trojan plankton. https://www.csc.ncsu.edu/faculty/jiang/Plankton/. Accessed: 2017-05-02.

[4] Department of defense app store. http://mashable.com/2013/10/30/department-of-defense-app-store/#INyfw4BG2aq7. Accessed: 2017-05-02.

[5] ICC Benchmark Apps. https://github.com/fgwei/ICC-Bench. Accessed: 2017-05-01.

[6] IccTA Benchmark Apps. https://github.com/secure-software-engineering/DroidBench/tree/iccta. Accessed: 2017-05-01.

[7] Monkey Automatic input generation. https://developer.android.com/studio/test/monkey.html. Accessed: 2017-05-01.

[8] Virusl Total. https://www.virustotal.com. Accessed: 2017-05-01.

[9] Arp, D., Spreitzenbarth, M., Hübner, M., Gascon, H., Rieck, K., and Siemens, C. Drebin: Effective and explainable detection of Android malware in your pocket. In *Proc. of NDSS* (2014).

[10] Arzt, S., Rasthofer, S., and Bodden, E. Instrumenting Android and java applications as easy as abc. In *International Conference on Runtime Verification* (2013).

[11] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., and McDaniel, P. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Conference on Programming Language Design and Implementation (PLDI)* (2014).

[12] Au, K. W. Y., Zhou, Y. F., Huang, Z., and Lie, D. Pscout: analyzing the Android permission specification. In *Proc. of CCS* (2012).

[13] Backes, M., Bugiel, S., Schranz, O., von Styp-Rekowsky, P., and Weisgerber, S. ARTist: The Android runtime instrumentation and security toolkit.

[14] Bastani, O., Anand, S., and Aiken, A. Interactively verifying absence of explicit information flows in Android apps. In *Proc. of OOPLSA* (2015).

[15] Bosu, A., Liu, F., Yao, D. D., and Wang, G. Collusive data leak and more: Large-scale threat analysis of inter-app communications. In *Proc. of AisaCCS* (2017).

[16] Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A.-R., and Shastry, B. Towards taming privilege-escalation attacks on Android. In *Proc. of NDSS* (2012).

[17] Chin, E., Felt, A. P., Greenwood, K., and Wagner, D. Analyzing inter-application communication in Android. In *Proc. of MobiSys* (2011).

[18] Davis, B., and Chen, H. RetroSkeleton: Retrofitting Android Apps. In *Proc. of MobiSys* (2013).

[19] Davis, B., Sanders, B., Khodaverdian, A., and Chen, H. I-ARM-Droid: A rewriting framework for in-app reference monitors for Android applications. *Proc. of MoST* (2012).

[20] Elish, K. O., Shu, X., Yao, D. D., Ryder, B. G., and Jiang, X. Profiling user-trigger dependence for Android malware detection. *Computers & Security* (2015), 255–273.

[21] Elish, K. O., Yao, D. D., Ryder, B. G., and Jiang, X. A static assurance analysis of Android applications. In *Technical Report., Department of Computer Science* (2013).

[22] Fratantonio, Y., Bianchi, A., Robertson, W., Egele, M., Kruegel, C., Kirda, E., Vigna, G., Kharraz, A., Robertson, W., Balzarotti, D., et al. On the security and engineering implications of finer-grained access controls for Android developers and users. In *Proc. of DIMVA* (2015).

[23] Gibler, C., Crussell, J., Erickson, J., and Chen, H. AndroidLeaks: Automatically detecting potential privacy leaks in Android applications on a large scale. In *Proc. of Trust and Trustworthy Computing* (2012).

[24] Gordon, M. I., Kim, D., Perkins, J., Gilham, L., Nguyen, N., and Rinard, M. Information-flow analysis of Android applications in DroidSafe. In *Proc. of NDSS* (2015).

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

[26] Rasthofer, S., Arzt, S., and Bodden, E. A machine-learning approach for classifying and categorizing Android sources and sinks. In *Proc. of NDSS* (2014).

[27] Reynaud, D., Song, D. X., Magrino, T. R., Wu, E. X., and Shin, E. C. R. Freemarket: Shopping for free in Android applications. In *Proc. of NDSS* (2012).

[28] Tian, K., Yao, D. D., Ryder, B. G., and Tan, G. Analysis of code heterogeneity for high-precision classification of repackaged malware. In *Proc. of MoST* (2016).

[29] Tian, K., Yao, D. D., Ryder, B. G., Tan, G., and Peng, G. Code-heterogeneity aware detection for repackaged malware. In *Proc. of IEEE Transactions TDSC* (2017).

[30] Xu, R., Saïdi, H., and Anderson, R. Aurasium: Practical policy enforcement for Android applications. In *Proc. of USENIX Security* (2012).

[31] Zhang, H., Yao, D. D., and Ramakrishnan, N. Detection of stealthy malware activities with traffic causality and scalable triggering relation discovery. In *Proc. of AsiaCCS* (2014).

[32] Zhang, H., Yao, D. D., and Ramakrishnan, N. Causality-based sensemaking of network traffic for Android application security. In *Proc. of AISec* (2016).

[33] Zhang, H., Yao, D. D., Ramakrishnan, N., and Zhang, Z. Causality reasoning about network events for detecting stealthy malware activities. *Computers & Security* (2016).

[34] Zhang, M., and Yin, H. AppSealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in Android applications. In *Proc. of NDSS* (2014).

[35] Zhou, Y., and Jiang, X. Dissecting Android malware: Characterization and evolution. In *Proc. of IEEE (S&P)* (2012).