Learning-based Cyber Security Analysis and Binary Customization for Security

Ke Tian

Dissertation submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

Danfeng(Daphne) Yao, Chair

Naren Ramakrishnan

Barbara G. Ryder

Na Meng

Gang Tan

June 9, 2018

Blacksburg, Virginia

Keywords: mobile security, malware detection, web security

Learning-based Cyber Security Analysis and Binary Customization for Security

Ke Tian

(ABSTRACT)

This thesis presents machine-learning based malware detection and post-detection rewriting techniques for mobile and web security problems. In mobile malware detection, we focus on detecting repackaged mobile malware. We design and demonstrate an Android repackaged malware detection technique based on code heterogeneity analysis. In post-detection rewriting, we aim at enhancing app security with bytecode rewriting. We describe how flow- and sink-based risk prioritization improves the rewriting scalability. We build an interface prototype with natural language processing, in order to customize apps according to natural language inputs. In web malware detection for Iframe injection, we present a tag-level detection system that aims to detect the injection of malicious Iframes for both online and offline cases. Our system detects malicious iframe by combining selective multi-execution and machine learning algorithms. We design multiple contextual features, considering Iframe style, destination and context properties.

Learning-based Cyber Security Analysis and Binary Customization for Security

Ke Tian

(GENERAL AUDIENCE ABSTRACT)

Our computing systems are vulnerable to different kinds of attacks. Cyber security analysis has been a problem ever since the appearance of telecommunication and electronic computers. In the recent years, researchers have developed various tools to protect the confidentiality, integrity, and availability of data and programs. However, new challenges are emerging as for the mobile security and web security. Mobile malware is on the rise and threatens both data and system integrity in Android. Furthermore, web-based iframe attack is also extensively used by web hackers to distribute malicious content after compromising vulnerable sites.

This thesis presents on malware detection and post-detection rewriting for both mobile and web security. In mobile malware detection, we focus on detecting repackaged mobile malware. We propose a new Android repackaged malware detection technique based on code heterogeneity analysis. In post-detection rewriting, we aim at enhancing app security with bytecode rewriting. Our rewriting is based on the flow and sink risk prioritization. To increase the feasibility of rewriting, our work showcases a new application of app customization with a more friendly user interface. In web malware detection for Iframe injection, we developed a *tag-level* detection system which aims to detect injection of malicious Iframes for both online and offline cases. Our system detects malicious iframe by combining selective multi-execution and machine learning. We design multiple contextual features, considering Iframe style, destination and context properties.

Acknowledgments

I would like to express my gratitude to my mentor Dr. Danfeng (Daphne) Yao. She introduced me to cyber security research and had since guided me through the most interesting and exciting research topics in the past five years. I also greatly appreciate Dr. Yao's kindness and generosity. I would not have survived the past five challenging years of graduate school without her encouragement and inspiration.

I would also like to express my sincere gratitude to Dr. Gang Tan, Dr. Na Meng, Dr. Naren Ramakrishnan and Dr. Babara Ryder who generously serve as my committee member. I have benefited a lot from their diverse perspectives and insightful comments which help deeper understanding in my research. I also want to thank Dr. Gang Wang for providing guidance on the web security. I would like to thank my friends and peers who I worked with over the past five years: Dr. Zhou Li, Dr. Kui Xu, Dr. Xiaokui Shu, Dr. Hao Zhang, Dr. Fang Liu, Dr. Long Cheng, Xiaodong Yu, Yin Liu, Zheng Song, Tong Zhang, Qingrui Liu, Xinwei Fu, Sazzadur Rahaman and Ya Xiao.

Finally, I would like to express my gratitude to my family who is always having faith in me. Their support helps me go through the hardest time during my PhD study.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Malware for cyber security has become a prominent problem for researchers and developers. In mobile security, Android is attackers' premium target for malware. Mobile malware is on the rise and threatens both data and system integrity in Android. In web security, Iframe is a web primitive frequently used by web developers to integrate content from third parties. It is also extensively used by web hackers to distribute malicious content after compromising vulnerable sites.

In mobile security, we focus on the Android repackaged malware detection and post-detection rewriting. One popular category of Android malware is repackaged malware. During repackaging, malware writers statically inject malcode and modify the control flow to ensure its execution. Researchers found 80.6% of malware are repackaged malware, which demonstrates the popularity and severity of repackaged malware [152]. Repackaged malware is difficult to detect by existing classification techniques, partly because of their behavioral similarities to benign apps.

We propose a new Android repackaged malware detection technique based on code heterogeneity analysis. Our solution strategically partitions the code structure of an app into multiple dependence-based regions (subsets of the code). Each region is independently classified on its behavioral features. We examine Android programs for code regions that seem unrelated in terms of data/control dependences. Regions are formed through data/control dependence analysis and their behavior is examined with respect to security properties (e.g., calling sensitive APIs). We refer to code in different regions as heterogeneous code if regions of the program exhibit distinguishable security behaviors. Our experimental results show a 30-fold improvement in repackaged malware classification. The average false negative rate for our partition- and machine-learning-based approach is 30 times lower than the conventional machine-learning-based approach (non-partitioned equivalent). Overall, we achieve a low false negative rate of 0.35% when evaluating malicious apps, and a false positive rate of 2.96% when evaluating benign apps.

After repackaged malware detection, we aim at improving app security with rewriting. App rewriting significantly enhances app security by enforcing security policies. Our rewriting is based on the flow and sink risk prioritization. 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. We demonstrate a practical security customization technique for Android apps. 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 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 also extend our rewriting framework with natural language process. General app customization is limited to analysts with experienced security analyst. We raise a question that is it possible to have a better interface for generating rewriting specifications to programs? To address this problem, we present an approach towards a language-based interface to programs for app customization. We build an interface prototype, named *IronDroid*, to customize apps according to natural language inputs. Our approach reduces the semantic gap from general human languages to specific app customization policies. We provide an end-to-end design from natural languages to automated app customization. We transform language commands into structure and dependence trees. Program inputs are extracted by mining the transformed trees. We generate rewriting specifications via a user-intention-guided taint flow analysis. The app is rewritten by following rewriting specifications using our rewriting framework.

Beyond mobile security, we also take an insight at web security. Iframe is a web primitive frequently used by web developers to integrate content from third parties. It is also extensively used by web hackers to distribute malicious content after compromising vulnerable sites. Previous works focused on page-level detection, which is insufficient for Iframe-specific injection detection. As such, we conducted a comprehensive study on how Iframe is included by websites around

Internet in order to identify the gap between malicious and benign inclusions. By studying the online and offline inclusion patterns from Alexa top 1M sites, we found benign inclusion is usually regulated. Driven by this observation, we further developed a *tag-level* detection system which aims to detect injection of malicious Iframes for both online and offline cases. Different from previous works, our system brings the detection granularity down to the tag-level for the first time without relying on any reference. The evaluation result shows our approach could achieve this goal with high accuracy.

Chapter 2

Literature Review

This chapter presents the related literature and techniques to Android repackaged malware detection and general-purposed android malware detection. I also include the techniques that we used in application rewriting and android vulnerability detection. For the very related work, I discuss the differences and demonstrate the advantages of my work.

2.1 Android Malware Detection Techniques

Repackaged Malware Detection. DroidMOSS [151] applied a fuzzy hashing technique to generate a fingerprint to detect app repackaging, the fingerprint is computed by hashing each subset of the entire opcode sequences. Juxtapp [65] examined code similarity through features of k-grams of opcode sequences. ResDroid [107] combined the activity layout resources with the relationship among activities to detect repackaged malware. Zhou et al. [150] detected the piggybacked code

based on the signature comparison.

However, the code level similarity comparisons are vulnerable to obfuscation technique, which is largely used in app repackaging. To improve obfuscation resilience, Potharaju et al. [101] provided three-level detection of plagiarized apps, which is based on the bytecode-level symbol table and method-level abstract syntax tree (AST). MassVet [41] utilizes UI structures to compute centroid metrics for comparing the code similarity among apps.

Solutions have been proposed on the similarity comparison of apps based on graph representations. DNADroid [46] compared the program dependence graphs of apps to examine the code reuse. AnDarwin [47] speeds up DNADroid by deploying semantic blocks in program dependence graphs, and then deployed locality hashing to find code clones. DroidSim [113] used component-based control flow graph to measure the similarity of apps. ViewDroid [143] focused on how apps define and encode user's navigation behaviors by using UI transition graph. DroidLegacy [52] detected a family of apps based on the extracted signature.

Instead of finding pairs of similar apps, our approach explores the code heterogeneity for detecting malicious code and benign code. Our approach avoids the expensive and often error-prone whole-app comparisons. It complements existing similarity-based repackage detection approaches.

Machine-Learning-based Malware Detection. Peng et al. [99] used the requested permissions to construct different probabilistic generative models. Wolfe et al. [124] used the frequencies of *n*-grams decompiled Java bytecode as features. DroidAPIMiner [19] and DroidMiner [132] extracted features from API calls invoked in the app. Drebin [24] gathered as many features including

APIs, permissions, components to represent an app, and then uses the collected information for classification. Gascon et al. [62] transformed the function call graph into features to conduct the classification. STILO [129] and CMarkov [128] applied hidden Markov models to detect anomaly behaviors. AppContext [133] adopted context factors such as events and conditions that lead to a sensitive call as features for classifying malicious and benign method calls. Crowdroid [35] used low-level kernel system call traces as features.

These solutions cannot recognize code heterogeneity in apps, as they do not partition a program into regions. In comparison, features in our approach are extracted from each DRegion to profile both benign and malicious DRegion behaviors.

Malware Detection with Quantitative Reasoning Our work is also related to malware classification with quantitative reasoning. Researchers [125, 126] 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. PRIMO [96] used probabilities to estimate the likelihood of implicit ICC communications. The triage of ICC Links is based on the true positive likelihood of links. Peng et al. [99] 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.

2.2 Android Rewriting and Defense Techniques

Android Taint Flow Analysis The vulnerability of apps can be abused by attackers for privilege escalation and privacy leakage attacks [34]. Researchers proposed taint flow analysis to discover sensitive data-flow paths from sources to sinks. CHEX [90] and AndroidLeaks [63] identified sensitive data flows to mitigate apps' vulnerability. Bastani *et al.* described a flow-cutting approach [30]. However, their work only provides theoretical analysis on impacts of a cut, without any implementation. DroidSafe [64] used a point-to graph to identify sensitive data leakage. FlowDroid [25] 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 [49] aims at automatically updating HTTP connections to HTTPS. Aurasium [130] instruments low-level libraries for monitoring functions. Reynaud *et al*. [104] rewrote an app's verification function to discovered vulnerabilities in the Android in-app billing mechanism. AppSealer [148] proposed a rewriting solution to mitigate component hijacking vulnerabilities, the rewriting is to generate patches for functions with component hijacking vulnerabilities. Fratantonio *et al*. [61] 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.

Defense of Vulnerabilities Grayware or malware with vulnerabilities can result in privacy leakage. Pluto [110] 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 [56] adopted dynamic taint analysis to track the potential misuse of sensitive data in Android apps. Anception [60] proposed a mechanism to defend the privilege escalation attack by deprivileging a portion of the kernel with sensitive operations. DIALDroid [33] and MR-Droid [88] detects inter-component communication vulnerabilities in app communications. Merlin [29] used path constraints to infer explicit information specifications to identify security violations. AspectDroid [21] 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.

2.3 Web-oriented Malware Detection

Content-based detection. Machine learning approach has been widely adopted in several research fields, e.g., anomaly detection and web security [144–146]. Detecting malicious code distributed through websites is a long-lasting yet challenging task. A plethora of research focused on applying static and dynamic analysis to detect such malicious code. .

Differential analysis has shown promising results in identifying compromised websites. By comparing a website snapshot or JavaScript file to their "clean-copies" (i.e., web file known to be untampered), the content injected by attacker can be identified [32, 87]. Though effective, getting "clean-copies" is not always feasible. On the contrary, `FrameHanger` is effective even without such reference. In addition to detection, researchers also investigated how website compromise can be predicted [109] and how signatures (i.e., Indicators of Compromise, or IoC) can be derived [39]. `FrameHanger` can work along with these systems. To expose the IFrame injected in the runtime by JavaScript code, we develop a dynamic analyzer to execute JavaScript and monitor DOM changes. Different from previous works on force execution and symbolic execution [76, 78, 80, 105], our selective multi-execution model is more lightweight with the help of content-based pre-filtering. Previous works have also investigated how to detect compromised/malicious landing pages using static features [37, 102]. Some features (e.g., "out-of-place" IFrames) used by their systems are utilized by `FrameHanger` as well. Comparing to this work, our work differs prominently regarding the detection goal (`FrameHanger` pinpoints the injected IFrame) and the integration of static and dynamic analysis (`FrameHanger` detects online injection).

URL-based detection. Another active line of research in detecting web attacks is analyzing the URLs associated with the webpages. Most of the relevant works leverage machine-learning techniques on the lexical, registration, and DNS features to classify URLs [31, 44, 85, 92]. In our approach, URL-based features constitute one category of our feature set. We also leverage features unique to IFrame inclusion, like IFrame style. To extract relevant features, we propose a novel approach combining both static and dynamic analysis.

Insecurity of third-party content. Previous works have studied how third-party content are integrated by a website, together with the security implications coming along. Nikiforakis et al. studied how JavaScript libraries are included by Alexa top 10K sites and showed many of the links were ill-maintained, exploitable by web attackers [94]. Kumar et al. showed a large number of websites fail to force HTTPS when including third-party content [82]. The study by Lauinger et al. found 37.8% websites include at least one outdated and vulnerable JavaScript libraries [84]. The adoption of policies framework like CSP has been measured as well, but incorrect implementation of CSP rules are widely seen in the wild, as shown by previous studies [36, 121, 122]. In this work, we also measured how third-party content were included, but with the focus on IFrame. The result revealed new insights about this research topic, e.g., the limited usage of CSP, and reaffirms that more stringent checks should be taken by site owners.

Chapter 3

Android Repackaged Malware Detection

3.1 Introduction

The ease of repackaging Android apps makes the apps vulnerable to software piracy in the open mobile market. Developers can insert or modify parts of the original app and release it to a third party market as new. The modification may be malicious. Researchers found 80.6% of malware are repackaged malware, which demonstrates the popularity and severity of repackaged malware [152].

There are two categories of techniques for detecting repackaged malware, i) similarity-based detection specific to repackaged malware and ii) general purpose detection. Specific solutions for repackaged Android apps aim at finding highly similar apps according to various similarity measures. For example, in ViewDroid [143], the similarity comparison is related to how apps encode

the user's navigation behaviors. DNADroid [46] compares the program dependence graphs of apps to examine the code reuse. MassVet [41] utilizes UI structures to compare the similarity among apps. Juxtapp [65] and DroidMOSS [151] examine code similarity through features of opcode sequences.

Although intuitive, similarity-based detection for repackaged malware may have several technical limitations. The detection typically relies on the availability of original apps for comparison, thus is infeasible without them. The pairwise based similarity computation is of quadratic complexity $O(N^2)$ in the number N of apps analyzed. Thus, the analysis is extremely time-consuming for large-scale screening.

General-purpose Android malware detection techniques (e.g., permission analysis [99], dependence analysis [123], API mining [19]) have a limited capability in detecting repackaged malware. The reason is that these analyses are performed on the entire app, including both the injected malicious code and the benign code inherited from the original app. The presence of benign code in repackaged malware substantially dilutes malicious features. It skews the classification results, resulting in false negatives (i.e., missed detections). In a recent study [53], researchers found that most missed detection cases are caused by repackaged malware. Thus, precisely recognizing malicious and benign portions of code in one app is important in improving detection accuracy.

We aim to significantly improve repackaged malware detection through designing and evaluating a new partition-based classification technique, which explores code heterogeneity in an app. Repackaged malware is usually generated by injecting malicious components into an original benign app, while introducing no control or data dependence between the malicious component and

the original app.

We examine Android programs for code regions that seem unrelated in terms of data/control dependences. Regions are formed through data/control dependence analysis and their behavior is examined with respect to security properties (e.g., calling sensitive APIs). We refer to code in different regions as heterogeneous code if regions of the program exhibit distinguishable security behaviors.

Recognizing code heterogeneity in programs has security applications, specifically in malware detection. Repackaged Android malware is an example of heterogeneous code, where the original app and injected component of code have quite different characteristics (e.g., the frequency of invoking critical library functions for accessing system resources). We are able to locate malicious code by distinguishing different behaviors of the malicious component and the original app.

Our main technical challenge is how to identify integrated coherent code segments in an app and extract informative behavioral features. We design a partition-based detection to discover regions in an app, and a machine-learning-based classification to recognize different internal behaviors in regions. Our detection leverages security heterogeneity in the code segments of repackaged malware. Our algorithm aims to capture the semantic and logical dependence in portions of a program. Specifically, we refer to a DRegion (Dependence Region) as a partition of code that has disjoint control/data flows. DRegion is formally defined in Def. 3. Our goal is to identify DRegions inside an app and then classify these regions independently. Malware that is semantically connected with benign and malicious behaviors is out of scope of our model and we explain how it impacts the detection.

While the approach of classifying partitioned code for malware detection appears intuitive, surprisingly there has not been systematic investigation in the literature. The work on detecting app plagiarism [150] may appear similar to ours. It decomposes apps into parts and performs similarity comparisons between parts across different apps. However, their partition method is based on rules extracted from empirical results, and cannot be generalized to solve our problem. A more rigorous solution is needed to precisely reflect the interactions and semantic relations of various code regions.

Our contributions can be summarized as follows:

- We provide a new code-heterogeneity-analysis framework to classify Android repackaged malware with machine learning approaches. Our prototype **DR-Droid**, realizes static-analysis-based program partitioning and region classification. It automatically labels the benign and malicious components for a repackaged malware.

- We utilize two stages of graphs to represent an app: a coarse-grained class-level dependence graph (CDG) and a fine-grained method-level call graph (MCG). The reason for these two stages of abstraction is to satisfy different granularity requirements in our analysis. Specifically, CDG is for partitioning an app into high-level DRegions; MCG is for extracting detailed call-related behavioral features. CDG provides the complete coverage for dependence relations among classes. In comparison, MCG provides a rich context to extract features for subsequent classification.

- Our feature extraction from individual DRegions (as opposed to the entire app) is more

effective under existing repackaging practices. Our features cover a wide range of static app behaviors, including user-interaction related benign properties.

- Our experimental results show a 30-fold improvement in repackaged malware classification. The average false negative rate for our partition- and machine-learning-based approach is 30 times lower than the conventional machine-learning-based approach (non-partitioned equivalent). Overall, we achieve a low false negative rate of 0.35% when evaluating malicious apps, and a false positive rate of 2.96% when evaluating benign apps.

The significance of our framework is the new capability to provide in-depth and fine-grained behavioral analysis and classification on programs.

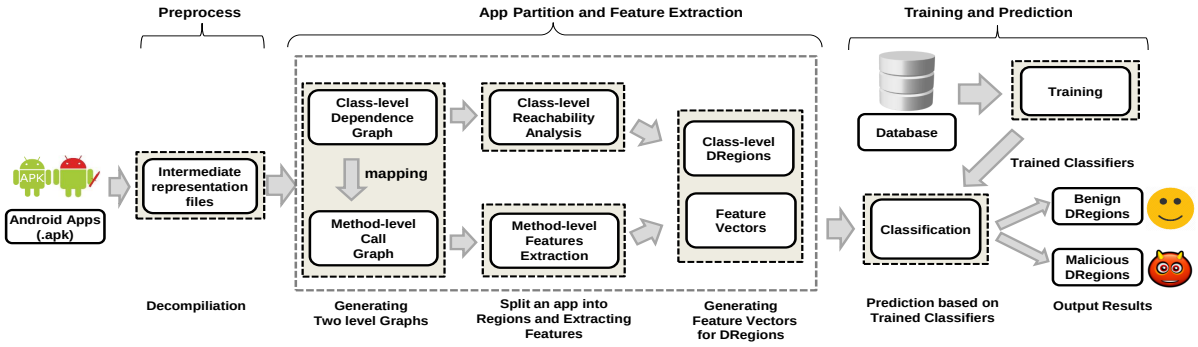


Figure 3.1: Workflow of our partition-based Android malware detection.

3.2 Overview and Definitions

In this section, we present our attack model, technical challenges associated with partitioning, and the definitions needed to understand our algorithms.

Repackaged malware seriously threatens both data privacy and system integrity in Android. There are at least two types of malware abuse through repackaged malware, data leak and system abuse. The danger of repackaged malware is that the malicious code is deeply disguised and is difficult to detect. Repackaged malware appears benign and provides useful functionality; however, they may conduct stealthy malicious activities such as botnet command-and-control, data exfiltration, or DDoS attacks. Our work described in this chapter can be used to screen Android apps to ensure the trustworthiness of apps installed on mission-critical mobile devices, and to discover new malware before they appear on app markets.

Assumption. Our security goal is to detect repackaged malware that is generated by trojanizing legitimate apps with a malicious payload, where the malicious payload is logically and semantically independent of the original benign portion. This assumption is reasonable because all the repackaged malware in the existing dataset contains disjoint code.

How to analyze the more challenging case of connected graphs in repackaged malware is out of the scope of our detection. Mitigations are discussed in Section 4.3.5. Our approach is focused on automatically identifying independent partitions (DRegions) of an app, namely partitions that have disjoint control/data flows. We perform binary classification on each element of the DRegion.

3.2.1 Challenges and Requirements

We analyze dependence-based connectivity as the specific heterogeneous property in code. Heterogeneous code can then be approximated by finding disjoint code structures in Android event

relation/dependence graphs. We aim to detect repackaged malware by identifying different behaviors in its heterogeneous code. Therefore, how to achieve an efficient partition and to acquire representative behaviors of each partition are key research questions.

Partition Challenges: One may analyze dependence relations for the purpose of code partition. A straightforward approach is to partition an app into clusters of methods based on function call relations [75]. However, this straightforward approach cannot solve the following challenges:

- Inaccurate representation of events. Method-level representation is less informative than class-level representation for profiling relations of events. An Android app is composed of different types of events (e.g., activities, services and broadcasts). An Android event is implemented by extending an Java class. Class information for events is scattered or lost in conventional method-level graphs. Furthermore, method-level call analysis cannot resolve the implicit calls within a life-cycle of event methods (e.g., `OnCreate`, `OnStart`, and `onPause`). There are no direct invoking relations among event methods. (These methods are managed by an activity stack by the Android system.) Thus, method-level call partition would generate an excessive number of subcomponents, which unnecessarily complicates the subsequent classification.

- Incompleteness of dependence relations. Call relations alone cannot accurately represent all possible dependence relations. Dependences may occur through data transformation. Android also has asynchronous callbacks, where the call relations are implicit. Thus, focusing on call dependence relations alone is insufficient.

Our approach for partitioning an app is by generating the class-level dependence graph (CDG) through exploring different categories of dependence relations. To partition an app into semantic-independent regions, a class-level representation is more suitable to measure the app semantic dependence relations.

Classification Challenges: Extracting meaningful features to profile each region is important for classification. In our partitioned setting, obstacles during feature extraction may include the following:

- Inaccurately profiling behaviors. Class-level dependences are coarse-grained. They may not provide sufficient details about region behaviors needed for feature extraction and classification. For example, the interactions among components within the Android framework may not be included.

- Insufficient representative features. Features in most of the existing learning based solutions are aimed at characterizing malicious behaviors, e.g., overuse of sensitive APIs. This approach fails to learn benign properties in apps. This bias in recognition may result in missed detection and evasion.

Our approach for achieving highly accurate classification is by extracting semantic features from method-level call graph (MCG). With the help of the MCG, we extract features (e.g., sensitive APIs and permission usage in existing approaches [19, 24]) to monitor malicious behaviors. Furthermore, we discover new user interaction features with the combination of graph properties to screen benign behaviors.

3.2.2 Definitions

We describe major types of class-level dependence relations later in Def. 1. These class-level dependence relations emphasize on the interactions between classes.

Definition 1. *We define three types of class-level dependence relations in an Android app.*

- ***Class-level call dependence.*** *If method m' in class C' is called by method m in class C, then there exists a class-level call dependence relation between C and C', which is denoted by $C \rightarrow C'$.*

- ***Class-level data dependence****. If variable v' defined in class C' is used by another class C, then there exists a data dependence relation between C and C', which is denoted by $C \rightarrow C'$.*

- ***Class-level ICC dependence****. If class C' is invoked by class C through explicit-intent-based inter-component communication (ICC), then there exists an ICC dependence relation between C and C', which is denoted by $C \rightarrow C'$.*

The ICC dependence is specific to Android programs, where the communication channel is constructed by using intents [42]. For the ICC dependence definition, our current prototype does not include implicit intent, which is not common for intra-app component communication. The dependence relations via implicit-intent based ICCs cannot be determined precisely enough in static program analysis.

Definition 2. ***Class-level dependence graph*** *(CDG) of an app $G = \{V, E\}$ is a directed graph, where V is the vertex set and E is the edge set. Each vertex $n \in V$ represents a class. The edge*

$e = (n_1, n_2) \in E$, *which is directed from* n_1 *to* n_2, *i.e.,* $n_1 \rightarrow n_2$. *Edge* e *represents one or more dependence relations between* n_1 *and* n_2 *as defined in Definition 1.*

The purpose of having our customized class-level dependence graphs is to achieve complete dependence coverage and event-based partition. The graph needs to capture interactions among classes. We define method-level call dependence and how to build the method-level call graph (MCG) based on this definition. We formally define DRegions through class-level dependence connectivity.

Definition 3. *Given class-level dependence graph* $G(V, E)$ *of an Android application, DRegions of the application are disjoint subsets of classes as a result of a partition that satisfies following two properties.*

1. *Dependence relations among the classes within the same DRegion form a directed connected graph. Formally, given a DRegions* R, *for any two classes* $(C_i, C_j) \in R$, $\exists$ *a path* $\vec{p} = (C_i = C_0, C_1, \ldots, C_k = C_j)$ *that connects* C_i *and* C_j.

2. *There exist no dependence relations between classes appearing in two different DRegions. Formally, given two DRegions* R_i *and* R_j, *for any class* $C_i \in R_i$ *and any class* $C_j \in R_j$, $\nexists$ *a path* $\vec{p} = (C_i = C_0, C_1, \ldots, C_k = C_j)$ *that connects* C_i *and* C_j.

Definition 4. ***Method-level call dependence.*** *If method* m *calls method* m', *then there exists a method-level call dependence relation between* m *and* m', *which is denoted by* $m \rightarrow m'$. *Method* m *and* m' *may belong to the same or different classes and one of them may be an Android or Java API.*

The purpose of constructing method-level call graphs is to extract detailed behavioral features for classifying each DRegion. The method-level call graph contains the app's internal call relations, and the interactions with the Android framework and users.

3.2.3 Workflow

Figure 3.1 shows the framework of our approach. Our approach can be divided into the following major steps:

1. **IR Generation**. Given an app, we decompile it into the intermediate representations (IR), which may be Java bytecode, `Smali`[1] code, or customized representation. The IR in our prototype is `Smali` code.

2. **CDG and MCG generation**. Given the IR, we generate both class-level and method-level dependence relations through the analysis on the `Smali` opcodes of instructions. We use the obtained dependence relations to construct the class-level dependence graphs (CDG) and method-level call graphs (MCG).

3. **App partition and mapping**. Based on the CDG, we perform reachability analysis to partition an app into disjoint DRegions. We map each method in MCG to its corresponding class in CDG by maintaining a dictionary data structure.

4. **Generating feature vectors**. We extract three categories of features from each DRegion in MCG. We construct a feature vector of each DRegion to describe its behaviors.

[1] `https://ibotpeaches.github.io/Apktool/`

5. **Training and classification**. We train classifiers on the labeled data to learn both benign and malicious behaviors of DRegions. We apply classifiers to screen new app instances by individually classifying their DRegions and integrating the results.

In order to determine the original app, from which a flagged malware is repacked, similarity comparisons need to be performed. Our comparison complexity $O(mN)$ would be much lower than the complexity (N^2) of a straightforward approach, where m is our number of flagged malware and N is the number of total apps analyzed. $m \ll N$, as the number of malware is far less than the total number of apps on markets.

3.3 Graph Generation and Partition

In this section, we provide details of our customized class-level dependence analysis and our partition algorithm.

3.3.1 Class-level Dependence Analysis

Our class-level dependence analysis is focused on Android event relations. It obtains class-level dependence relations based on fine-grained method- or variable-level flows. We highlight the operations for achieving this transformation.

Data dependence. In the variable-level flow F, we trace the usage of a variable v' which is defined in class C'. In case v' is used by another class C, e.g., reading the value from v' and writing it into

a variable v defined in class C, we add a direct data dependence edge from C to C' in CDG.

ICC dependence. ICC dependence is the Android specific data dependence, where data is transformed by intents through ICC. An ICC channel occurs when class C initializes an explicit intent. Method m (generally `OnCreate` function) in class C' is invoked from class C. By finding an ICC channel between class C and C' through pattern recognition, we add a direct ICC dependence edge from C to C' in CDG.

Call dependence. We briefly describe the operations for obtaining class-level call dependence when given the method-level call graph. We first remove the callee functions that belong to Android framework libraries. For the edge $e = \{m, m'\}$ that indicates method m' is called by method m, in case m belongs to a class C and m' belongs to class C', we add a direct call dependence edge from C to C' in CDG.

Pseudocode for generating the Class-level Dependence Graph is shown in Algorithm. 1. Function FINDDEPENDENTCLASS (m_k^i) is used to find the class set $S(m_k^i)$ that any $C_j \in S(m_k^i)$ contains dependence relations with a class C_i ($m_k^i \in C_i$) through control-/data- flow in method m_k^i. Functions ISDATADEPENDENT(C_i,C_j), ISICCDEPENDENT(C_i,C_j), and ISCALLDEPENDENT(C_i,C_j) are used to detect our defined dependence relations between classes.

We give our implementation details to statically infer these relations in Section 3.5.1. All four dependence relations can be identified by analyzing instructions in IR. The complexity of connecting the class-level call graph is $O(\mathcal{N})$, where $\mathcal{N}$ is the total number of the instructions in the IR decompiled from an app. We do not distinguish the direction of the edges when partitioning the

CDG.

3.3.2 App Partition and Mapping Operations

The goal of app partition operation is to identify logically disconnected components. The operation is based on the class-level dependence graph (CDG). We use reachability analysis to find connected DRegions. Two nodes are regarded as neighbors if there is an edge from one node to the other. Our algorithm starts from any arbitrary node in the CDG, and performs breadth first search to add the neighbors into a collection. Our algorithm stops when every node has been grouped into a particular collection of nodes. Each (isolated) collection is a DRegion of an app. In other words, classes with any dependence relations are partitioned into the same DRegion. Classes without dependence relations are in different DRegions.

Our mapping operation projects a method m in method-level call graph (MCG) to its corresponding class C in CDG. Mapping is uniquely designed for our feature extraction. Specifically, its purpose is to map extracted features to the corresponding DRegion. The mapping operation is denoted by $F_{mapping} : S_c \rightarrow P^m_{S_c} = \{G'_{c1}, G'_{c2}, ...\}$, where input S_c is a DRegion in CDG, and output $P^m_{S_c}$ is a set of call graphs in MCG. The mapping algorithm projects a method in MCG to a DRegion in CDG by using a lookup table. We refer to $P^m_{S_c}$ as the projection of S_c. Features extracted from $P^m_{S_c}$ belong to the DRegion S_c. Suppose that a method m^i exclusively belongs to a class C_i, and a class C_i exclusively belongs to a DRegion S_C, thus we have the mapping as $m^i \in C_i \in S_C \rightarrow P^m_{S_c}$. Figure 3.2 illustrates an example of the mapping function. Property 1 demonstrates

the non-overlapping property of the mapping function.

Property 1. *If DRegion S_α and DRegion S_β are disjoint in the class-level dependence graph (CDG), then their corresponding projection $P^m_{S_\alpha}$ and projection $P^m_{S_\beta}$ are disjoint in the method-level call graph (MCG).*

Proof. We prove Property 1 by contradiction. Suppose that two methods m in class C and m' in class C' in two DRegions, there exists a path $\widehat{p} = (m, n_1, n_2, ..., m')$. For any two neighbor nodes n_i, n_{i+1} on V, n_i and n_{i+1} must be dependent through data-/control- relations: 1) if n_i, n_{i+1} are in the same class C_1, C_1 belongs to one DRegion. 2) if n_i, n_{i+1} are in different class C_1 and C_2, then C_1 and C_2 are connected in CDG (through a dependence edge). C_1 and C_2 are categorized to one DRegion after the partition. By induction, C and C' must belong to the same DRegion, which contradicts to our assumption. □

Our partition algorithm guarantees the non-overlapping property during the mapping operation. Features extracted from each MCG belong to exact one DRegion after partition. By the app partition and mapping, we explicitly identify each DRegion and its associated MCGs. We discuss more details on how to extract features from MCG in the following Section 3.4.

3.4 Feature Vector Generation

We analyze APIs and user interaction functions to approximate their behaviors. Our features differ from most existing features by considering DRegion behavior properties. We describe three types

of features in this section.

Feature Engineering. The three types of features are based on previous solutions [99] [19] and our observations. For example, permissions and sensitive APIs have been demonstrated as representative features in the whole-program based malware detection. We reuse these features for our region-based classification. Unlike these features are extracted from the whole program, we customize the feature extraction process for our region-based classification. Our features are extracted from each DRegion instead of the whole program. Furthermore, we introduce new user interaction features that are not explored in the existing machine-learning-based solutions before. Recently, Elish [53] demonstrated the high accuracy of a rule-based malware detection. The rule is based on whether a sensitive API is triggered by a user interaction function. In contrast to the rule-based detection, we extract frequencies of user interaction functions and encode them into the feature vector. We also introduce a new statistic feature, which is called the coverage rate (CR). The feature is obtained by our empirical observation that malware invokes a large number of sensitive APIs without user's involvement.

3.4.1 Feature Extraction

Traditionally permission features analyze the registered permissions in `Androidmanifest.xml` as a complete unit [99]. Because our approach is focused on DRegions and different DRegions may use different permissions for various functionalities, we calculate the permission usage in each DRegion.

Type I: User Interaction Features. Malicious apps may invoke critical APIs without many user interactions [145]. User interaction features represent the interaction frequency between the app and users. Android provides UI components and each of them has its corresponding function for triggering. For example, a `Button` object is concatenated with `onClick` function, and a `Menu` object can be concatenated with `onMenu-ItemClick` function. We record the frequencies of 35 distinct user interaction functions and additional 2 features summarizing statistics of these functions. The statistics features represent the total number of user interaction functions and the number of different types of user interaction functions in a DRegion, respectively. We define a feature called coverage rate (CR), which is the percentage of methods directly depending on user-interactions functions. We compute the coverage rate (CR) for a projection $P^m_{S_c}$ of a DRegion S_c as:

$$CR(P^m_{S_c}) = \frac{\bigcup_{U \in V_i} U.successors()}{|V(P^m_{S_c})|} \tag{3.1}$$

The CR rate statically approximates how closely the user interacts with functions in a DRegion. In Equation (1), $P^m_{S_c}$ is the projection for a DRegion S_c in CDG. V_i is the set of user interaction methods in $P^m_{S_c}$, where $V_i \subseteq P^m_{S_c}$. $U.successors()$ is the successors vertices of method U in MCG. Any method in $U.successors()$ is directly invoked by U. $|V(P^m_{S_c})|$ is the total number of methods in $P^m_{S_c}$. Figure 3.3 shows an example to calculate the coverage rate.

Type II: Sensitive API Features. We divide sensitive APIs into two groups: Android-specific APIs and Java-specific APIs. The APIs are selected based on their sensitive operations [53]. For Android-specific APIs, we focus on APIs that access user's privacy information, e.g., reading geographic location `getCellLocation`, getting phone information `getDeviceId`. For Java-

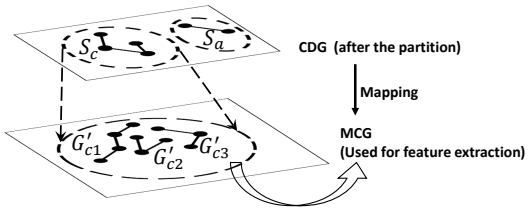


Figure 3.2: An illustration of mapping operation that projects a DRegion in CDG to a set of graphs in MCG. S_c and S_a are two class-level DRegions in CDG. The projection of S_c consists of three method-level call graphs $\{G'_{c1}, G'_{c2}, G'_{c3}\}$.

specific APIs, we focus on file and network I/Os, e.g., writing into files `Write.write()`, and sending network data `sendUrgentData()`. We extract 57 most critical APIs and 2 features on their statistic information (e.g., total count and occurrence of APIs) as features.

Type III: Permission Request Features. We analyze whether a DRegion uses a certain permission by scanning its corresponding systems calls or permission-related strings (e.g., Intent related permissions) [27]. We specify a total of 137 distinguished permissions and 2 features on permission statistics (e.g., total count and occurrence of permissions). The Android framework summarizes all the permissions into 4 groups: `normal`, `dangerous`, `signature` and `signatureOrSystem`. We record the permission usage in each group and the statistics about these groups.

Coverage rate (CR) is new. It is obtained by our empirical observation that malware invokes a large number of sensitive APIs without user's involvement. These complex features cover the behaviors of DRegions from various perspectives. We expect these features to be more obfuscation resilient than signature features extracted from bytecode or structures of the control-flow graph.

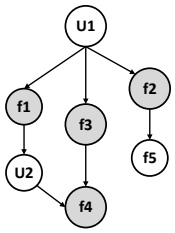


Figure 3.3: An example illustrating the computation of coverage rate, U_1 and U_2 are two user interaction functions, f_1 to f_5 are five method invocations. f_1, f_2, f_3 are successors of U_1 and f_4 is the successor of U_2. The coverage rate for this DRegion is $\frac{4}{7} = 57\%$.

	True Malicious	True Benign
Detected as Malicious	TP	FP
Detected as Benign	FN	TN

Table 3.1: Semantics of true and false positive and true and false negative in our model.

3.4.2 Feature Vector Analysis

We generate a feature vector for each DRegion of an app. For classification, each DRegion is independently classified into benign or malicious.

We perform the standard 10-fold cross-validation to calculate FNR (false negative rate), FPR (false positive rate), TPR (true positive rate) and ACC (accuracy rate) for each fold. These rates are defined as:

$$FNR = \frac{FN}{P},\ FPR = \frac{FP}{N}$$
$$TPR = \frac{TP}{P},\ ACC = \frac{TP + TN}{P + N}$$

where FN represents the number of false negative (i.e., missed detection), FP represents the number of false positive (i.e., false alerts), TP represents the number of true positive (i.e., accuracy of detection), TN represents the number of true negative (i.e., accuracy of identifying benign apps), P represents the number of malicious apps and N represents the number of benign apps.

Classification of Apps. Our classifiers can be used to classify both single-DRegion and multi-DRegion apps. For a multi-DRegion app after classification, we obtain a binary vector showing each DRegion marked as benign or malicious. We define the malware score r_m as follows:

$$r_m = \frac{N_{mali}}{N_{total}} \tag{3.2}$$

In Equation (2), N_{mali} is the number of DRegions labeled as malicious by classifiers, N_{total} is the total number of DRegions and $r_m \in [0, 1]$. If an app contains both malicious and benign DRegions, then we regard this app as a suspicious repackaged app.

3.5 Evaluation

The objective of our preliminary evaluation is to answer the following questions:

- **Q1)** Can our approach accurately detect non-repackaged malware that has a single DRegion (Section 5.2)?
- **Q2)** How much improvement is our approach in classifying repackaged malware that has multiple DRegions (Section 5.3)?
- **Q3)** Can our approach distinguish the benign and malicious code in repackaged malware (Section 5.3.1)?
- **Q4)** What is the false positive rate (FPR) and false negative rate (FNR) of our approach in classifying apps that have multiple DRegions (Section 5.3.2 and Section 5.4)?
- **Q5)** What is the performance of DR-Droid (Section 5.5)?
- **Q6)** Can our approach discover new malware (Section 5.6)?

We implement our prototype with `Smali` code analysis framework `Androguard` [2], graph analysis library `networkX`, and machine learning framework `scikit-learn`. Most existing machine learning based approaches (e.g, [19, 62]) are built on the intermediate representation with `Smali` code. `Smali` code analysis achieves large scale app screening with low performance overhead, because `Smali` code analysis is performed on the assembly code representation. Our current prototype is built on the `Smali` code for the scalability of large-scale app analysis. Our prototype is implemented in `Python` with total 4,948 lines of code [3].

[2] `http://code.google.com/p/androguard.`

[3] `https://github.com/ririhedou/dr_droid`

We evaluated our approach on malware dataset Malware Genome [152] and VirusShare database [4]. We also screened 1,617 benign apps to compute false positive rate and 1,979 newly released apps to discover new malware.

3.5.1 Implementation Details

Building upon the method-level call graph construction [62] from `Smali` code, we construct more comprehensive analysis to approximate the class-level dependence graph and graph partitioning. We highlight how **DR-Droid** approximates various class-level dependence relations with intra-procedure analysis (i.e., discovering dependence relations) and inter-procedure analysis (i.e., connecting the edges). Our experiment results indicate that our dependence relations provide sufficient information for identifying and distinguishing different behaviors in an app.

Inferring class-level call dependence. Opcodes beginning with `invoke` represent a call invocation from this calling method to a targeted callee method. Call dependence can be inferred by parsing the call invocation. E.g., `invoke-virtual` represents invoking a virtual method with parameters. `invoke-static` represents invoking a static method with parameters and `invoke-super` means invoking the virtual method of the immediate parent class. We identify each instruction with `invoke` opcodes and locate the class which contains the callee method. The class-level call dependence is found, when the callee method belongs to another class inside the app. Because we focus on the interactions among classes, Android API calls are not included.

[4] `http://virusshare.com/`

Inferring class-level data dependence. Opcodes such as `iget`, `sget`, `iput`, and `sput` are related with data transformation. For example, the instruction "`iget-object v0, v0, Lcom/geinimi/AdActivity;->d: Landroid/widget/Button;`" represents reading a field instance into `v0` and the instance is a `Button` object named `d`, which is defined in another class `<Lcom/geinimi/AdActivity;>`.

Furthermore, there is a subset of opcodes for each major opcode, e.g., `iget-boolean` specifies to read a boolean instance and `iget-char` specifies to read a `char` instance. By matching these patterns, we obtain the data dependence among these classes.

Inferring class-level ICC dependence. To detect an ICC through an explicit intent, we identify a targeted class object that is initialized by using `const-class`, then we trace whether it is put into an intent as a parameter by calling `Intent.setclass()`. If an ICC is trigged to activate a service (by calling `startService`) or activate an activity (by calling `startActivity`), we obtain the ICC dependence between current class and the target class.

Method-level call graph construction. Our method-level call graph is constructed while we analyze call relations in the construction of the CDG by scanning `invoke` opcode, which is similar to the standard call graph construction [62]. We store more detailed information including the class name, as well as the method name for each vertex in MCG. For example, `<Landroid/telephony/SmsManager;>` is the class name for dealing with messages and `sendTextMessage(...)` is a system call with parameters to conduct the behavior of sending a message out. After the construction of MCG, we use a lookup table structure to store the projection for each DRegion in CDG and to maintain the mapping relation between a method and a DRegion.

Cases	FNR(%)	FPR(%)	ACC(%)
KNN	6.43 ± 5.22	6.50 ± 2.67	93.54 ± 3.33
D.Tree	4.78 ± 2.90	3.52 ± 1.57	95.79 ± 2.14
R.Forest	3.85 ± 3.27	1.33 ± 0.78	97.30 ± 1.96
SVM	7.42 ± 4.85	1.46 ± 0.58	95.28 ± 2.58

Table 3.2: 10-fold cross-validation for evaluating the classifiers' performance in classifying single-DRegion apps.

3.5.2 Non-repackaged Malware Classification

Our first evaluation is on a set of non-repackaged malicious applications and a set of benign applications. Each of them contains just a single DRegion. The DRegion is labeled as benign if the app belongs to the benign app dataset, and the DRegion is labeled as malicious if the app belongs to the malicious app dataset. There are two purposes for the first evaluation: 1) comparing the detection accuracy of different machine learning techniques, 2) obtaining a trained classifier for testing complicated repackaged apps. The classification result is binary (0 for benign and 1 for malicious) for single DRegion apps. We evaluated four different machine learning techniques: Support Vector Machine (SVM), K-nearest neighbor (KNN), Decision Tree (D.Tree) and Random Forest (R.Forest) in non-repackaged (general) malware classification. Our training set is broadly selected from 3,325 app samples, among which 1,819 benign apps from Google Play, and 1,506 malicious apps from both Malware Genome and VirusShare.

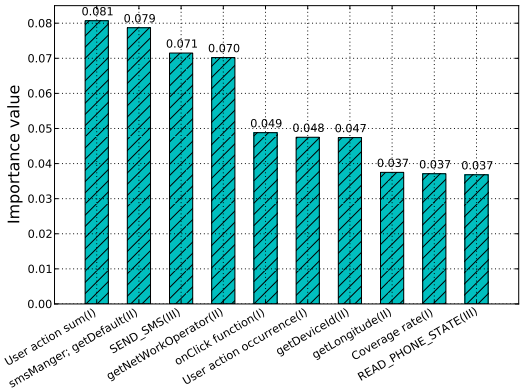


Figure 3.4: Top ten important features with their ranking values, which are computed by Random Forest classifier.

Our feature selection step reduces the size of features from 242 to 80. We choose the radial basis function as SVM kernel, 5 as the number of neighbors in KNN, and 10 trees in the Random Forest.

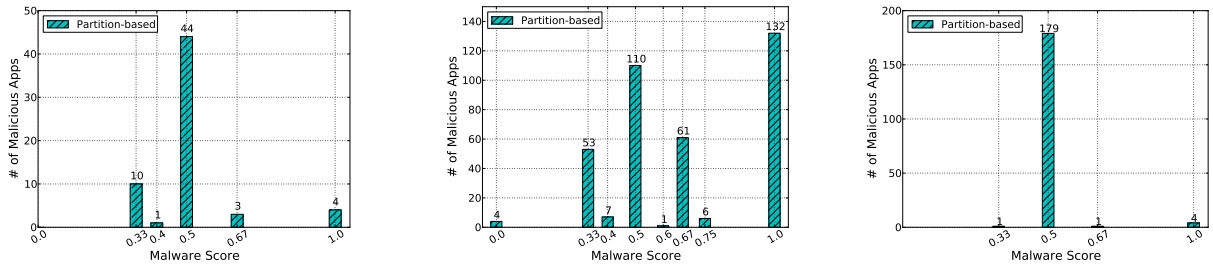
We used a standard measurement 10-fold cross-validation to evaluate efficiency of classifiers. In 10-fold cross-validation, we randomly split the dataset into 10 folds. Each time, we use 9 folds of them as the training data and the 1 fold left as the testing data. We evaluate the performance of classifiers by calculating the average FPR, FNR and ACC. Our 10-fold cross-validation results are shown in Table 3.2, where each value is represented as the average $\pm$ the standard deviation. Figure 3.4 shows the top ten features with their types and ranking importance values, where coverage rate (CR) ranks the ninth. We found four of top ten important features belong to user interaction features (Type I). The user interaction features are important in our classification.

We conclude that: 1) to answer **Q1**, DR-Droid detects non-repackaged malware with single DRegions with high accuracies. 2) The Random Forest classifier achieves the highest AUC value

Malware Families	Geinimi		Kungfu		AnserverBot		Average
	FN	FNR(%)	FN	FNR(%)	FN	FNR(%)	FNR(%)
Partition-based	0(62)	0	4(374)	1.07	0(185)	0	**0.35**
Non-partition-based	12(62)	19.36	12(374)	9.89	3(185)	1.62	**10.29**

Table 3.3: False negative rate for detecting three families of repackaged malware. Our partition-based approach reduces the average false negative rate by 30-fold.

0.9930 in ROC and accurate rate (ACC) 97.3% in two different measurements. 3) Our new user interaction features have a significant influence on the improvement of classifiers.



(a) Geinimi family (b) Kungfu family (c) AnserverBot family

Figure 3.5: Prediction of malware score in different Malware families.

3.5.3 Repackaged Malware Classification

We tested our approach on more complicated repackaged malware which contains multiple DRe-gions. We calculate *malware score* r_m for each repackaged malware. Unlike binary classification in existing machine-learning-based approaches, r_m is a continuous value in $[0, 1]$ to measure DRe-

gions with different security properties.

There are no existing solutions on the classification of multiple DRegions in an app. For comparison, we carefully implemented a control method called the non-partition-based classification. To have a fair and scientific comparison with the non-partition-based which does not consider code heterogeneity, DR-Droid's classification and the control method's classification use the same Random Forest classifier and the same set of features from Section 5.2. The **only difference** between our method and the control method is that the control method treats an app in its entirety. The control method represents the conventional machine-learning-based approach.

We assessed several repackaged malware families: Geinimi, Kungfu (which contains Kungfu1, Kungfu2, Kungfu3, Kungfu4) and AnserverBot multi-DRegion apps in these families. The major reason for choosing these families is that they contain enough representative repackaged malware for testing. Other malware datasets (e.g., VirusShare) do not specify the types of malicious apps. It is hard to get the ground truth of whether an app in the datasets is repackaged or not. The classification accuracy results of our partition-based approach and the non-partition-based approach are shown in Table 3.3.

Our partition-based approach gives the substantial improvement by achieving a lower FNR in all three families. Specifically, the non-partition-based approach misses 12 apps in Geinimi and 3 apps in AnserverBot family. In comparison, our approach accurately detects all the malicious DRegions in Geinimi and AnserverBot families. The non-partition-based approach misses 12 apps in Kungfu family. In comparison, our approach misses 4 apps in Kungfu family. The average FNR for our approach is 0.35%.

To answer **Q2**, our solution gives 30-fold improvement over the non-partition-based approach on average false negative rate in our experiment. This improvement is substantial. The control method without any code heterogeneity analysis is much less capable of detecting repackaged malware.

Case Study of Heterogeneous Properties

For an app (`DroidKungFu1--881e*.apk`) in Kungfu family, the malicious DRegion contains 13 classes whose names begin with `Lcom/google/ssearch/*`. The app attempts to steal the user's personal information by imitating a Google official library. The other DRegion whose name begins with `Lcom/Allen/mp/*` is identified as benign by our approach. There are some isolated classes such as `R$attr`, `R$layout`, which are produced by `R.java` with constant values. The malicious DRegion has its own life cycle which is triggered by a receiver in the class `Lcom/google/ssearch/Receiver`. All the processes run on the background and separately from the benign code.

Table 3.4 shows the distribution of a subset of representative features in two different methods. Particularly, DRegion 1 contains many user interaction functions with no sensitive APIs and permissions. However, DRegion 2 invokes a large number of sensitive APIs and requires many critical permissions. In the experiment, DRegion 1 is classified as benign and DRegion 2 is classified as malicious. The different prediction results are due to the differences in DRegion behaviors, which originally comes from their code heterogeneity. The non-partition-based approach fails to detect this instance. The experiment results validate our initial hypothesis that identifying code heterogeneity can substantially improve the detection of repackaged malware.

To answer **Q3**, our approach successfully detects different behaviors in the original and injected components, demonstrating the importance and effectiveness of code heterogeneity analysis.

False Negative Analysis

We discuss possible reasons that cause false negatives in our approach. 1) Integrated benign and malicious behaviors. Well integrated benign and malicious behaviors in an app can cause false negatives in our approach. `Com.egloos.dewr.ddaycfgc` is identified by Virus Total as a trojan but is predicted as benign by our approach. The reason is that the malicious behavior, which communicates with a remote server, is hidden under the large amount of benign behaviors. The activities are integrated tightly and several sensitive APIs are used in the app. 2) Low code heterogeneity in malicious components. Low code heterogeneity means that malicious code does not exhibit obvious malicious behaviors or is deeply disguised. To reduce false negatives, a more advanced partition algorithm is required to identify integrated benign and malicious behaviors. How to detect low heterogeneity malicious code is still an open question. We provide more discussion in Section 6.

Distribution of DRegions in Different Dataset

We evaluate the distribution of the DRegion number in three different datasets: the Genome repackaged malware dataset, the Virus-Share general malware dataset and the benign app dataset. Figure 3.7 shows the distribution of the number of DRegions in three datasets by randomly testing 1,000 apps. We find 66.9% of apps in Genome has multiple DRegions, in comparison, 6.5% of

DroidKungfu1–881e*.apk		Partition (ours)		Non-partition
Feature	Description	DRegion1	DRegion2	N/A
Type III	READ_PHONE_STATE permission	**0**	**1**	1
	READ_LOGS permission	**0**	**1**	1
Type II	getDeviceId function in Landroid/telephone/ telephoneManager	**0**	**1**	1
	read function in Ljava/io/InputStream	**0**	**3**	3
	write function in Ljava/io/FileOutput	**0**	**1**	1
Type I	onClick function occurrence	**16**	**2**	18
	# of distinct user-interaction functions	**5**	**1**	5
	onKeyDown function occurrence	**3**	**0**	3
Classification		Benign	Malicious	Benign
Correctness		Yes		No

Table 3.4: Our method shows heterogeneous properties in the repackaged app (DroidKungfu1–881e*.apk), where the no-partition based cannot.

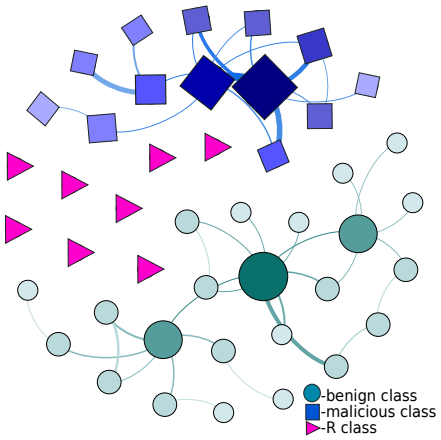


Figure 3.6: A simplified class-level dependence graph for the app DroidKungFu1–881e*.apk. Each node represents a class, and the diameter of a node indicates the total degree of its corresponding class.

apps in Virus Share and 28.1% in benign app dataset have multiple DRegions. Because of repackaged malware, the distribution of Genome malware significantly differs from others in the others.

3.5.4 False Positive Analysis with Popular Apps

The purpose of this evaluation is to experimentally assess how likely our detection generates false positives (i.e, false alerts). We collect 1,617 free popular apps from Google Play market, the selection covers a wide range of categories. We evaluate a subset of apps (158 out of 1,617) that have multiple large DRegions. Each app contains 2 or more class-level DRegions with at least 20 classes in the DRegion. In the 158 apps, Virus Total identifies 135 of them as true benign apps, that apps raise no security warnings.

The most common cause of multi-DRegions is the use of ad libraries. A majority of multiple DRe-

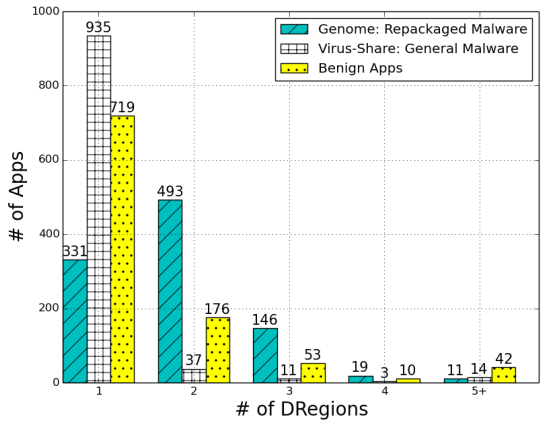


Figure 3.7: The distribution of the number of DRegions in different datasets, where X axis represents the number of DRegions in an app and Y axis represents the count of apps with a certain number of DRegions. Repackaged malware tends to have more DRegions.

gion apps have at least one ad library (e.g., `admob`). The ad library acquires sensitive permissions, access information and monitor users' behaviors to send the related ads for profit. Some aggressive ad libraries, e.g., `Adlantis`, results in a false alarm in our detection. `Adlantis` acquires multiple sensitive permissions, and it tries to read user private information. The ad package involves no user interactions. We identify ad libraries by matching the package name in a whitelist. More effort is needed to automatically identify and separate ad libraries. Table 3.5 presents the false positive rate with and without ads libraries. The normal ad libraries do not affect our detection accuracy, while the aggressive ads libraries dilute our classification results and introduce false alerts into our detection. When excluding aggressive ad libraries, our detection misclassifies 4 out of 135 benign apps. To answer **Q4**, our approach raises a false positive rate (FPR) of 2.96% when classifying free popular apps and a false negative rate (FNR) of 0.35% when classifying repackaged malware.

	w/o Ads	w/ Group 1 Ads	w/ Group 2 Ads
% of Alerts	2.96%	2.96%	5.18%

Table 3.5: For 135 benign apps, how the percentage of alerts changes with the inclusion of ad libraries. Group 1 Ads are benign ad libraries, namely *admob* and *google.ads*. Group 2 Ads refer to the known aggressive ad library *Adlantis*. Group 1 does not affect our detection accuracy, whereas Group 2 increases the number of alerts.

3.5.5 Performance Evaluation

We evaluate the performance of our approach based on the execution time. The detection of a repackaged malware includes graph generation, feature extraction and classification of DRegions. We measure each step separately and evaluate the runtime performance overhead.

We focus on two aspects. One aspect is the graph generating time and features extraction time as the preprocess for detecting malware. The other aspect is the machine learning operations for evaluating the detecting overhead. The performance time can vary a lot in different sizes of apps. We concentrate on the average time for processing an app. Our execution experiment is set on the x86_64 system with a total of 3GM memory.

From Figure 3.8, our average time for generating CDG and MCG graphs is nearly 2.5 seconds. Our average time for extracting features is 27.84 milliseconds, which is considerably small for dealing with a large number of apps. Our graph construction algorithm is efficient by parsing instructions of an app. Table 3.6 shows the runtime of the machine learning operations. We split

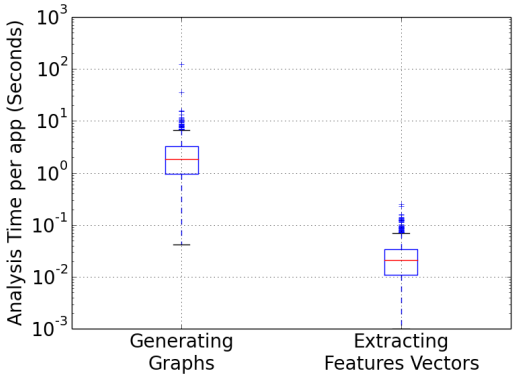


Figure 3.8: Time distribution for generating graphs and extracting features.

machine learning operations into the training part and the prediction part. Our training operation includes feature extraction and feature vector generation. The feature vectors are feed into different classifiers to train machine learning parameters. In the prediction, the classifier predicts each DRegion separately. We measure the runtime for classifying each DRegion. From Table 3.6, our training time is rather small. R.Forest takes 0.539 seconds for training, SVM takes most 3.583 seconds for training. Our prediction time is also negotiable. On average, D.Tree takes 0.51 seconds for prediction, R.Forest takes most 1.38 seconds for prediction.

3.5.6 Discover New Suspicious Apps

We evaluate a total of 1,979 newly released (2015) apps. Our approach raises a total of 127 alarms. Because of the lack of ground truth in the evaluation of new apps, computing FP requires substantial manual efforts on all these flagged apps. We performed several manual studies on the flagged apps. The apps are randomly selected from different categories.

Time	Training/Second	Prediction/Millisecond		
		mean	max	min
R.Forest	0.539	1.38	9.66	0.94
KNN	0.534	1.09	13.81	0.67
D.Ttree	0.588	0.51	2.91	0.36
SVM	3.583	0.85	5.24	0.60

Table 3.6: Execution time analysis for machine learning

Manual Verification Our rule for labeling an app as malware in our manual analysis is that the malware collects privacy information and sends it out without user notification. We identify an app as malware based on a two-step validation: 1) Statistics of permissions and APIs. We compare the frequency of permissions and APIs in the app towards its description. If an app contains critical APIs that do not match its description, e.g., a weather app contains reading and sending SMS APIs, we regard the app is potentially malicious. 2) Sensitive APIs that are not triggered by user inputs. If there exists a secret and sensitive data flow and the data flow path does not include user interaction functions, we regard the app is malicious. In the manual verification, we utilize static analysis and manual inspection to verify the flagged apps. We decompile each app into `Smali` intermediate representations (IR). We extract permissions and APIs from Smali IR. We compare permissions with app descriptions to find potentially malicious apps. We manually analyze the methods that invoke sensitive APIs to detect sensitive data flows. We report an app as malware if it is confirmed by our manual analysis.

We list four of them which are identified as malicious by our manual analysis. The first two suspicious apps are verified by our manual analysis, but are missed by Virus Total. Virus Total does detect the latter two apps. To answer **Q5**, our approach is capable of detecting new single-DRegion and multiple-DRegions malware.

1) `za.co.tuluntulu` is a video app providing streaming TV programs. However, it invokes multiple sensitive APIs to perform surreptitious operations on the background, such as accessing contacts, gathering phone state information, and collecting geometric information.

2) `com.herbertlaw.MortagageCaculator` is an app for calculating mortgage payments. It contains a benign DRegion by the usage of the `admob` ad library. It also contains an aggressive library called `appflood` in the malicious DRegion, which collects privacy information by accessing the phone state and then stores it in a temporary storage file.

3) `com.goodyes.vpn.cn` is a VPN support app with in-app purchase and contains multiple DRegions. A malicious package `Lcom/ccit/mmwlan` is integrated with a payment service `Lcom/alipay/*` in one malicious DRegion. It collects the user name, password, and device information. It exfiltrates information to a constant phone number.

4) `longbin.helloworld` is a simple calculator app with one DRegion. However, it requests 10 critical permissions. It modifies the `SharedPreferences` to affect the phone's storage, records the device ID and sends it out through `executeHttpPost` without any users' involvement.

Summary.

Our results validate the effectiveness of code heterogeneity analysis in detecting Android malware.

We summarize major experimental findings as follows.

- Our prototype is able to identify malicious and benign code by distinguishing heterogeneous DRegions.
- Our partition-based detection reduces false negatives (i.e., missed detection) by 30-fold, when compared to the non-partition-based method.
- Our prototype achieves low false negative rate (0.35%) and low false positive rate (2.96%).

Our tool can also be used to identify ad libraries and separate them from the main app. These components can be confined at runtime in a new restricted environment, as proposed recently in [112].

3.6 Discussion and Limitations

Graph Accuracy. Our current prototype is built on the `Smali` code intermediate representation for a low overhead. Machine-learning based approaches require a large number of apps for training. This graph generation is based on analyzing patterns on the instructions of `Smali` code. Our approach may miss detection of some data-dependence edges (e.g. implicit ICCs [54] and on-Bind functions), because of a lack of flow sensitivity [89] [90]. Our analysis under-approximates the dependence-related graph because of the missing edges. Context- and flow-sensitive program analysis improves the graph accuracy and increases analysis overhead. To balance the performance

and the accuracy in constructing the graphs is one of our future directions. We plan to extend our prototype to an advanced program analysis technique without compromising the performance.

Our graph construction is based on the static code analysis. The current static analysis is not sound because it cannot represent the full app logic [26]. Advanced evasion techniques (e.g., dynamic loading, code obfuscation, and drive-by downloading) introduce inaccuracy in the graph construction because of the missing edges. This limitation poses challenges for detecting repackaged malware because our approach could not identify these dependence relations. In the drive-by downloading attack, the repackaged payload could be a simple logic to download malicious code. The downloaded code is isolated from the main component and performs malicious behaviors separately. To mitigate the limitation caused by the drive-by downloading, a possible solution is to combine our static analysis with dynamic monitoring. We construct the dependence graph including both original code and downloaded code. The aggregated code indicates the full app logic, which is the combination of original code and downloaded code in the app. The graph partition is based on the aggregated code. The classification is based on extracted DRegions from the aggregated code.

To reduce the under-approximation of the dependence-related graph, we also plan to extend the definitions of dependence relations as our future work. Our current approach defines three type of dependence relations (call, data and ICC dependences). Additional edge dependences can include reflection-based call relations and call relations from dynamically loaded code. For each additional dependence relation, we extend our approach with more specific analysis, e.g., dynamic monitoring and string analysis. Our future work is to extend the dependence relations for a more sound graph

construction.

Dynamic Code. Our current prototype is built on static analysis. How to analyze dynamic code is outside the scope of our analysis. Static analysis cannot accurately approximate dependence relations that can be only identified dynamically [115], e.g, calling through Java native interface (JNI) and native code. The lack of dynamic analysis results in missing edges during the graph construction, which may introduce extra DRegions in the classification. Extra DRegions skew classification results because of the imprecision of features. However, the impact of dynamic obfuscation is limited in our analysis. AndroidLeaks [63] found that only 7% of apps contain the native code. In our future work, we plan to extend our prototype with a hybrid static and dynamic analysis. The dynamic analysis is used to detect call relations by dynamic code loading. The hybrid analysis enhances the resistance of obfuscation techniques.

Code Obfuscation. Code obfuscation [81] can be utilized by malware writers to evade malware detection. For example, ProGuard[5] is the default obfuscation tool suggested by Google. ProGuard obfuscates application classes and methods with short and unclear names. Our approach is resilient to the renaming-based obfuscation. The renaming-based obfuscation does not modify the data dependencies and call relations among classes. Our dependence-related graph is constructed by detecting different types of dependence relations. The region is identified based on the connectivity inside the graph, regardless of its class and method names. For more advanced obfuscation techniques, e.g., reflection, our current static analysis cannot detect these dependence relations. For example, there is no direct call invocation in the reflection method. The callee via the reflec-

[5] `https://developer.android.com/studio/build/shrink-code.html`

tion is defined as a string and cannot be identified in the invocation instruction. The obfuscation introduces implicit dependence relations, which cannot be directly resolved in our analysis. For the future work, we aim to extend our approach with advanced reflection-targeted analysis techniques [86].

Integrated Malware. Our future work will generalize our heterogeneity analysis by supporting the analysis of complex code structures where there is no clear boundary between segments of code. Our current prototype is not designed to detect malicious DRegions that are semantically connected and integrated with the rest of an app. Advanced repackaged malware may be produced by adopting code rewriting techniques, where malicious code is triggered by hijacking normal code execution [50]. In that case, partitioning the dependence graph into DRegions would be challenging, because of their connectivities. However, to generate such malware, malicious writers need to have a more comprehensive knowledge about the execution of the original app, which is not common. One may need to make careful cuts to the dependence graph to isolate (superficially) connected components [26] [38], based on their semantics and functionality.

Advanced Malware. There is a trend that malware writers tend to abuse packing services to evade malware screening [149] [74]. Malware adopts code-packed techniques to prevent the analyst from acquiring app bytecode. Typical anti-analysis techniques include metadata modification and DEX encryption. These advanced malware poses challenges for our approach to construct CDG and MCG. Our approach cannot construct the dependence graph without obtaining the original dex code of an app. To mitigate the limitation, we could extend our approach with AppSpear [74] to increase the resilience towards the advanced malware. We apply AppSear as the pre-process

to extract legitimate DEX code. The DEX code is then used as the input in our approach to construct CDG and MCG for partition. Our future work will generalize our heterogeneity analysis by supporting anti-packing code extraction.

3.7 Conclusions and Future Work

We addressed the problem of detecting repackaged malware through code heterogeneity analysis. We demonstrated its application in classifying semantically disjoint code regions. Our preliminary experimental results showed that our prototype is very effective in detecting repackaged malware and Android malware in general. For future work, we plan to improve our code heterogeneity techniques by enhancing dependence graphs with context and flow sensitivities.

Algorithm 1 Class-level Dependence Graph Generation

Require: the class-set $S_C = \{C_i, C_2, \ldots, C_n\}$, each class C_i represents a list of methods. the Method-set of C_i: $S_m^i = \{m_1^i, \ldots, m_k^i\}$, where $m_j^i \in C_i$ is a list of instructions in IR.

Ensure: class-level dependence graph $CDG = \{V, E\}$.

1: $V = \emptyset$, $E = \emptyset$
2: **function** GEN_CDG(CDG, S_C)
3: **for** each $C_i \in S_C$ **do**
4: **for** each $m_k^i \in S_m^i$ **do**
5: $S(m_k^i)$ = FINDDEPENDENTCLASS(m_k^i)
6: **for** each $C_j \in S(m_k^i)$ **do**
7: **if** ISDATADEPENDENT(C_i,C_j) **then**
8: UPDATECDG(C_i,C_j,CDG)
9: **end if**
10: **if** ISICCDEPENDENT(C_i,C_j) **then**
11: UPDATECDG(C_i,C_j,CDG)
12: **end if**
13: **if** ISCALLDEPENDENT(C_i,C_j) **then**
14: UPDATECDG(C_i,C_j,CDG)
15: **end if**
16: **end for**
17: **end for**
18: **end for**
19: **return** CDG
20: **end function**

Chapter 4

Application Customization with Rewriting

4.1 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., [63, 64, 90]). In addition, the use of machine-learning methods enables automatic malware recognition based on multiple data-flow features (e.g., [24, 117]).

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 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 [49]. Rewriting for the Internet permission check has also been performed [61]. AppSealer [148] 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 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. As shown in Figure 4.1a in Section 2, 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 chapter, we demonstrate a practical 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 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** [116] [1]. 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 [2]. 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.

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

[2] Flow ranking is a special case of sink ranking in our Algorithm 2.

4.2 Overview

Before we give the overview of our approach in Section 6.3, we first show a few examples to motivate the needs for ranking sensitive data flows and rewriting apps for security.

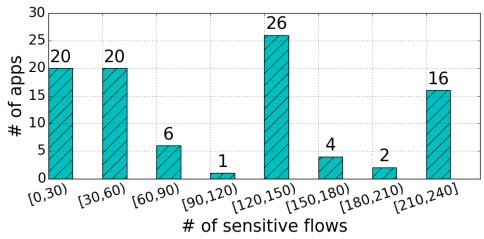
We target data leaks in our current threat model, specifically data flows in an app that may result in the disclosure and exfiltration of sensitive data. With proper source-sink definitions, the proposed sink-ranking and rewriting-based monitoring framework can be extended to support other security applications, which is discussed in Section 4.3.5.

4.2.1 Motivation and Design Choices

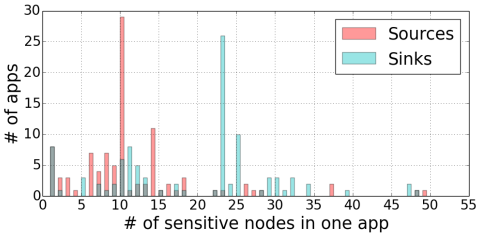
Security Usage of App Rewriting. Table 4.1 summarizes the security applications with our rewriting. Our rewriting can identify multiple vulnerabilities such as ICC hijacking and privacy leak. We rewrite apps to enforce different security policies, these security policies help a security analyst efficiently 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 4.4.1.

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-



(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.

Figure 4.1: The example for the distribution of sensitive flows and sources and sinks.

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.

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 [152]. We use FlowDroid [25] for static program analysis and SuSi [103] 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. Figure 4.1a shows the distribution of the number of sensitive flows. Figure 4.1b 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.

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.

Sink Rewriting vs. Flow Rewriting. Once the most sensitive sink is identified, rewriting that end-point 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.

Sensitive API-based Risk vs. Permission-based Risk. These two risk metrics are equivalent

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

Table 4.1: 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.

in our model. We map the sensitive API calls of a data-flow path into their corresponding Android permissions, as shown in Figure 4.2. 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 [24]).

A Toy Example. In Figure 4.2, 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 Figure 4.2 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.

4.2.2 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,

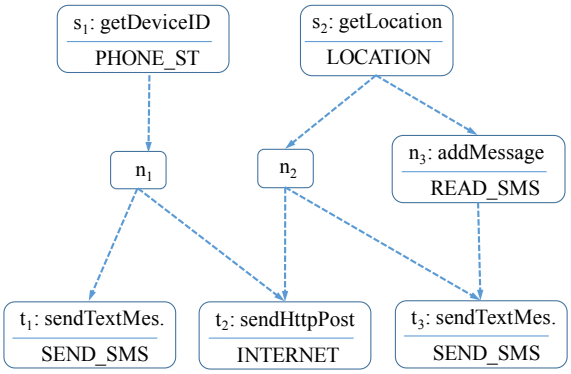


Figure 4.2: An example of a taint-flow graph. Nodes represent function calls or instructions. Permissions (shown at the bottom of a node) associated with the functions (shown at the top of a node) are shown. Directed edges represent data dependence relations.

aggregate risk, and the standard taint-flow graph.

Definition 5. ***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 chapter. We compute risk values of permissions through a maximum likelihood estimation approach.

In Section 4.3, 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.

4.2.3 Workflow

Figure 6.3 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 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 4.3.1.

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 4.3.3. [3]

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

[3] Other permission-risk quantification techniques may be used, e.g., Bayesian-Network based Android permission risk analysis [99].

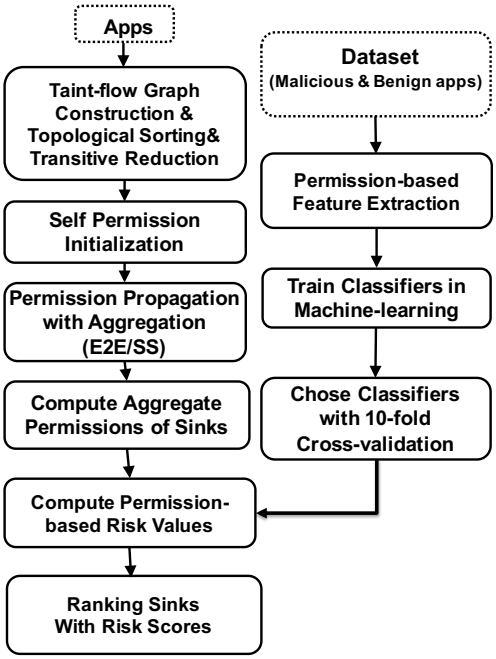


Figure 4.3: Our workflow for prioritizing risky sinks.

security customization techniques in Section 4.3.4. Besides rewriting, the sink-ranking technique is also useful for static analysis based malware detection.

4.3 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 2 in the Appendix.

Next, we describe technical details of our operations. We present risk propagation in Section 4.3.1, permission mapping in Section 4.3.3 and rewriting in Section 4.3.4.

4.3.1 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 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.

Transitive Reduction. The purpose of transitive reduction is to maximally remove redundant edges while preserving reachability of the graph [20]. 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 $G'(V, E', S, T)$.

Transitive reduction produces a directed acyclic graph (DAG). For each sink t, it has a subgraph reversely rooted by t, i.e., there exists a subgraph rooted by t, if the directions of edges are reversed.

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 6.

Definition 6. *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*

$$P[t] \;=\; P_s[t] \cup \left\{ \bigcup_{\{s \in S \;|\; \exists\; f = \{s \rightsquigarrow t\}\;\}} P_s[s] \right\} \tag{4.1}$$

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

$$P[t] = P_s[t] \cup \left\{ \bigcup_{\substack{\{s \in S,\; \{n_1 ... n_k\} \in f \\ |\; \exists\; f = \{s \rightsquigarrow t\}\}}} P_s[s] \cup P_s[n_1] ... \cup P_s[n_k] \right\} \tag{4.2}$$

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. We show pseudocode of the aggregation algorithm.

4.3.2 Aggregation Algorithm

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 2 shows the pseudocode for the permission aggregation and risk computation.

For the example in Figure 4.2, 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 $P[t_3] = \{\text{RECEIVE\_SMS, SEND\_SMS}\}$, whereas E2E aggregation has a larger aggregate risk set for the sink, which is $P[t_3] = \{\text{RECEIVE\_SMS, READ\_SMS, SEND\_SMS}\}$. Our experiments in Section 4.4.2 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.

4.3.3 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 Kirin [58] and Bayesian-based probabilistic models [99]), how to use them for prioritizing sinks for rewriting has not been systematically studied.

Definition 7. ***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:*

$$r(t) = \sum_{p \in P[t]} w(p) \tag{4.3}$$

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 [24]. 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 4.4.5. 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.

4.3.4 Automatic App Rewriting

We rewrite on the app's intermediate representation Jimple, which is based on Java analysis framework Soot. We implement our rewriting framework by supporting Android-specific functions, e.g., ICC. Table 4.2 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 [49,130]), our inter-app ICC relay rewriting approach requires more substantial code modification [4].

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

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

Rewriting Granularity	I-ARM-Droid [51] RetroSkeleton [49]	ReDroid **(Ours)**
Package-level (Repackage)	✓	✓
Class-level (Class Inject)	✓	✓
Method-level (Method Invoc.)	✓	✓
ICC-level (Intent Redirect)	–	✓
Flow-based Rewriting	–	✓
Sink-based Rewriting	–	✓

Table 4.2: 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.

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.

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) and explicit intent. Android ICC mechanism enables the communication among different apps [43].

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 if needed.

4.3.5 Discussion and Limitations

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

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., [63, 90]. 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.

Dynamic Permission. Google has recently introduced Android dynamic permission to protect user privacy [5]. 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.

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 are 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.

[5] `https://goo.gl/9FTnEL`

4.4 Experimental Evaluation

We use FlowDroid [25] for static program analysis. Our mapping from a statement into the requested permission is based on PSCout [27]. 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 source and sink identifiers come from SUSI [103], 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[6]. These apps cover different categories and contain complex code structures. As we show in Figure 4.1, 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 4.4.1). **RQ2:** Does the more complex E2E aggregation method provide higher accuracy in ranking (In Section 4.4.2)? **RQ3:** Are the ranking results consistent with manual validation (In Section 4.4.3) ? **RQ4:** Is ReDroid flow-aware, i.e., being able to differentiate sinks with identical method names

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

(In Section 4.4.4)? **RQ5:** How accurate is our maximum likelihood estimation for the permission-risk mapping (In Section 4.4.5)? **RQ6:** How much is our analysis overhead (In Section 4.4.6)?

4.4.1 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 4.1 summarizes the security applications with our rewriting. We utilize benchmark apps to evaluate the feasibility of our rewriting framework. We also use two grayware examples to demonstrate how to use rewriting to mitigate static analysis limitations.

Benchmark Suits Evaluation. We evaluate our ICC relay and logging rewriting strategies on DroidBench(IccTA)[7] and ICC-Bench[8]. 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[9]. 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,

[7] https://github.com/secure-software-engineering/DroidBench/tree/iccta

[8] https://github.com/fgwei/ICC-Bench

[9] https://developer.android.com/studio/test/monkey.html

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_statActivity	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 4.3: 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 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). VirusTotal reports it as benign [10]. 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

[10]We submitted two grayware APKs to VirusTotal on Aug-10-2016

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.

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 with the permission: 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.

4.4.2 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. 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.

Apps	In-degree	Sink-only	SS Aggre.	E2E Aggre.
Malware	5%	25%	97%	100%
Benign	13%	47%	95%	100%

Table 4.4: Percentage of (malicious or benign) apps whose riskiest sink under a metric (in-degree, sink-only, or SS aggregation) is the same as the riskiest sink under the E2E aggregation metric (i.e., correctness of results using E2E as the ground truth).

Table 4.4 compares 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 achieve higher accuracy, 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() $\rightarrow$ getActiveNetworkInfo() $\rightarrow$ outputStream() $\rightarrow$ 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.

We compare the permission propagation properties in malicious and benign apps. We consider two conditions, A and B, which are defined next. Table 4.5 presents the percentages of apps that exhibit such conditions. The experimental results show a large number of apps, especially malware, involve multiple (≥ 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.

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

Table 4.5: 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.

4.4.3 RQ3: Validation of Sink Priorities

Because of the lack of benchmarks [11], 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. 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

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

that Adlantis libraries cause binary-classification based malware detection to fail [117].

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 [55].

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 [12]. 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 [13].

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 [14]. These manual validation efforts provide the initial evidences indicating the quality of our ranking results.

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

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

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

4.4.4 RQ4: Case Study on Sensitive Taint Flows

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 4.6 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() $\rightarrow T_4$.

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 4.6: 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.

4.4.5 RQ5: Quality of Likelihood Estimation

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 benign apps are collected from official app market Google Play. The malicious apps are selected from popular malware database Genome and VirusShare. 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 folds.

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.

Table 4.7 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. 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 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

	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**

Table 4.7: 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.

sensitive nodes.

4.4.6 RQ6: Analysis Overhead

We compare the runtime of Algorithm 2 under two SS and E2E aggregations in Figure 5.4. [15] Experiments were performed over both benign and malware datasets on a Linux machine with Intel Xeon CPU (@3.50GHz) and 16G memory. Figure 5.4 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

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

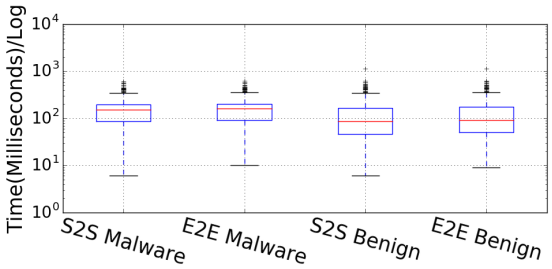


Figure 4.4: Runtime of permission propagation in Algorithm 2 on malware and benign apps under SS and E2E aggregation functions, respectively. Both aggregation methods have a low average runtime of around 0.1 second, with E2E aggregation slightly slower than SS.

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 4.4.1). On average, both logging and ICC relay based rewriting achieves nearly 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.

4.5 Conclusions and Future Work

In this chapter, 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.

Algorithm 2 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 */

Chapter 5

Automatic Application Customization with Natural Language Processing

5.1 Introduction

The popularity of Android apps introduces new opportunities and challenges in security research. Existing research aims at detecting malware [24, 25, 56, 96, 117, 118, 126, 147]. These approaches screen malware with reasonable success. However, these approaches are designed for general-purpose malware detection. They are not targeted to modify apps or restrict app behaviors. The post-detection customization is necessary for app security enforcement. Users utilize app customization to protect privacy and enhance security. In this chapter, we consider the problem of app customization by rewriting apps' bytecode. We explore the possibilities to have a better user

interface for everyone to programs. We motivate our approach to enlarge the popularity of users who can use app customization techniques. Our goal is to provide a more user friendly interface for app customization.

We motivate the need of app customization in three aspects: 1) Customized security preferences. Users have individual privacy preferences on the personal data. The number of people who do not trust mobile apps is growing rapidly [11]. A recent survey suggests that 54% of mobile users are worried about personal privacy [16]. Users would like to customize apps for individual privacy preferences. 2) Economic reasons. Legitimate apps may potentially snoop on users' private data and track their activities. Untrusted libraries are capable of collecting information without user awareness. Users would like to customize apps for better security and privacy. 3) Transparency of app logic. Users are unaware of how their data is used and transformed. Sensitive data could be acquired and transmitted in multiple ways (e.g., to other apps or the internet). Apps contain complex code structures and multiple components (e.g., activities and services), and generally users do not know the inner working of the apps they install.

These factors push the need for app customization. Users have motivations to customize apps and restrict private data usage. A user friendly interface could significantly facilitate the app customization process. Security analysts and normal users benefit from the interface for generating security-oriented customization policies. The *technical challenge* to build a natural language interface to programs is the semantic gap between language-level representations and program-level specifications. Google makes a recent effort with Android dynamic permission [2]. Dynamic permission allows users to deny access to private data on the system level. It only provides limited

operations: turning access on or off. General app rewriting solutions [49,61] are based on straightforward function parsing. They are neither capable of extracting user intentions nor feasible for complex customization requirements. Current language-based control systems support functionalities for particular tasks, e.g., opening a door or turning on a TV. However, security-oriented app customization requires the modification of apps to match domain specific requirements. These domain specific requirements are generally generated by a limited number of security analysts or experts. How to reduce the semantic gap is still unclear in the existing solutions.

Our goal is to provide a natural language interface to programs for app customization. A naive approach is to ask users to complete questionnaires and surveys. Security specifications are extracted by experienced experts from these questionnaires and surveys. This approach needs manual verification and is not scalable for general app customization. It is tedious and error-prone to ask users for security policies. Inspired by recent achievements in language understanding, our approach identifies user intentions from natural languages. We aim to reduce the *semantic gap* from natural languages to rewriting specifications. The semantic gap is that language-level sentences and code-level rewriting contain different structures and representations. Our main technical challenge is to eliminate the semantic gap between the two abstractions of representations.

In this chapter, we present *IronDroid*, a new app customization tool with a natural language interface. *IronDroid* enables automatic app rewriting with security-oriented natural language analysis. We propose a new mining algorithm to extract user security preferences from natural languages. The rewriting specification is extracted via a user-intention-guided taint flow analysis. The rewriting specification focuses on the sensitive sinks that result in privacy violation. The app is au-

tomatically modified by following rewriting specifications. In our design, we provide an efficient interface for language-level and code-level representation transition, blazing the trail toward usable app customization for a more usable rewriting solution. We also demonstrate a general rewriting framework which supports complex app modifications. We summarize our contributions as follows:

- We study the problem of an automatic rewriting framework for app customization. We present an interface design for generating rewriting specifications from languages. Our prototype *IronDroid* extracts user intentions from natural languages and customizes apps appropriately. *IronDroid* efficiently transforms natural language descriptions into rewriting specifications.

- We provide a new graph-mining-based algorithm to identify user intentions from natural languages. We propose a user-intention-based taint flow analysis for mapping natural languages into rewriting specifications. We build a general rewriting framework that is feasible for complex app customization.

- We implemented an automatic rewriting framework for app customization. In the experiment, our approach achieves 90.2% in understanding user intentions, a significant improvement over the state-of-the-art solution. Our framework supports more rewriting operations than existing solutions. Our rewriting successfully rewrites all benchmark apps. Our approach introduces negligible 3.3% overhead in rewriting. We also extend *IronDroid* for practical privacy protection in real-world apps.

In this chapter, we explore the possibilities for a natural language interface for app customization. Our work presents an enabling technology to enhance app security with a natural language interface. Although understanding languages for generating security-oriented rewriting policies is challenging, we make impressive progress toward automatic app customization. Our design includes natural language transition, rewriting specification extraction, user-intention-guided guided taint flow analysis and automatic app rewriting. Our approach presents the first attempt for natural language supported app customization. The integration of app customization with natural language processing would significantly enlarge the popularity to apply security-oriented app customization techniques. The interface would allow users to protect privacy and enhance security more easily.

We point out the challenges for processing language sentences to rewrite specifications. We provide several possible mitigations to improve the practical usability of the prototype. *IronDroid* is able to restrict and rewrite app behaviors for security purposes. A more advanced version of our approach could be embedded into voice control assistants (e.g., Siri [3], Alexa [1]). However, before natural language analysis and taint flow analysis can be made fully reliable and usable, the use scenario is unlikely.

5.2 Overview

In this section, we present security applications of rewriting, our assumption, technical challenges and the definitions needed to understand our approach.

5.2.1 Technical Challenges And Assumption

Security applications of rewriting. We summarize two rewriting scenarios for security.

Policy Enforcement. Organization administrators could use rewriting to enforce organization security policies. The security policies can be customized to meet organization requirements. A natural language interface reduces manual efforts to define security policies.

Privacy Protection. Users could use rewriting to protect privacy. A natural language interface provides possibilities for a user to specify her personal security concern. The interface rewrites the app to satisfy a user's expectations.

Goal. In this chapter, our goal is to design a natural language interface to programs for rewriting. Although accurately understanding natural languages is challenging in practice, we elaborate our efforts towards applying languages processing in security-oriented rewriting. the interface enlarges the population (e.g., normal users and analysts) to adopt app customization for security. The interface transforms a language sentence into program inputs (i.e., sources and sinks) for generating rewriting specifications. Our current rewriting specifications only focus on the sinks with privacy violation. We rewrite sensitive sinks to enhance security of an app.

Assumption. We assume apps that pass the vetting process are still not fully trusted. The assumption is reasonable because a recent study found hundreds of untrusted apps in the official market[1]. Users have privacy concerns on the data usage inside an app. Private data can be potentially leaked via a sensitive sink without user notification.

[1] `https://goo.gl/naZX8S`

Challenges. Identifying semantic elements and dependence relations in sentences has been studied in the NLP community. However, these techniques like name entity recognition [93] or relation extraction [28] *cannot* be directly applied for our purpose. They are not designed for the domain of app customization. Furthermore, app rewriting modifies programs on the code level. Programs consist of type-formatted code, which expresses little semantic information. Developing such tool needs comprehensive considerations across different domains. The main technique challenge is the mapping natural languages to program inputs for rewriting. It would be very difficult in practice to process users' sentences in free forms. In our design, we require the sentences in a structured format that can be parsed by our prototype. We discuss how to deal with more natural expressions as our future direction in Section 5.6. In the example, the user concerns on the device information, and cares whether it is leaked by text messages. We use this sentence as a running example [2] through our analysis.

Never share my device information, if it is sent out in text messages.

We aim to rewrite an app based on the above sentence. In summary, we face two major technical challenges:

- **Sentimental intention towards security objects**. Given a sentence, we extract both the security-related object and the sentimental value towards the object. Table 5.1 demonstrates different goals for traditional NLP analysis and ours. Traditional NLP analysis identifies the sentimental value for the whole sentence [108]. In contrast, our approach detects user intentions toward a security-related object. Extracting user intentions for personalized security is

[2] In our prototype, we expect users express the sentence in a structured and clear format.

Examples	**Security Object**	**StanfordNLP (sentence)**	**Ours (object)**
Please block the location	Location	Positive	Negative
Never disable my location	Location	Negative	Positive
Do not share my location	Location	Negative	Negative

Table 5.1: Difference between sentimental analysis towards a sentence and a security object/-word. The example shows inconsistent sentimental values between sentences and security objects. Comparing with stanfordNLP, we use sentimental analysis for a different purpose. We use the analysis for generating rewriting specifications. The positive/negative in stanfordNLP means the overall combination of positive and negative words in a sentence. The positive/negative in *IronDroid* means the trustworthiness towards a security object from a user perspective. In the first example, StanfordNLP identifies the sentimental value is positive because the overall attitude of this sentence is positive. *IronDroid* identifies the sentimental value for location is negative because the user does not want to share location.

challenging. We build our graph mining techniques to extract security objects. An alternative method is a keyword-based searching algorithm. We demonstrate the imprecision of the keyword-based searching in the evaluation. The imprecision is due to the lack of sentence semantic analysis.

- **Semantic gaps between human languages and app rewriting**. Human languages and app rewriting have different representations. Rewriting specifications are identical to program code structures, while human languages have no direct correlations with the code structures. Mapping language-level user intentions to code-level specifications is challenging. We eliminate the semantic gap by identifying *private data leak vulnerabilities*. We semantically map user intentions into program analysis inputs. A user-intention-guided taint flow analysis is introduced for app rewriting.

5.2.2 Definitions

We give several key definitions used in our model, including user intention predicate, the taint flow graph and rewriting specification. We define user intention predicate T_s for extracting user intentions towards a security object.

Definition 8. ***User Intention Predicate*** *is a tuple $T_s = \{p, s, c\}$ extracted from a sentence S_{user}, p is a security object from the predefined object set P, where $p \in P$. s is a sentimental value $s \in \{1, -1\}$, where -1 means the negative attitude and 1 means the positive attitude. c is the constraint string for the object p.*

User input examples	**Object** p	**Source APIs**
	Sentimental s	**Rewriting Op**
	Constraint c	**Sink APIs**
Never share my device information, if it is sent out in text messages.	Phone	getDevieId() getSubscribeId()
	s = -1	u.stop, u.trigger
	sent out in text messages	sendTextMessage() sendDataMessage()
Do not release my location outside this app.	Location	getLatitude() getLongitude()
	s = -1	u.stop, u.trigger
	outside this app	*
Do not share my location, if it is sent out to the internet.	Location	getLatitude() getLongitude()
	s = -1	u.stop, u.trigger
	to the internet	HttpExecute() SocketConnect() URLConnect()
The app can share my audio content if needed.	Audio	AudioRead()
	s = 1	-
	if needed	N/A

Table 5.2: Examples of extracting user intention predicates and mapping them to taint flow analysis inputs. We aim to generate a mapping from a language to code-level inputs. * means we return all possible functions. N/A means we could not find a semantic mapping. In rewriting, $u.stop|u.trigger$ means the invocation of the unit is prohibited if invoking the unit triggers the privacy violation.

$s = -1$ means users do not trust the app and could limit the use of private data that is related to p.

We define a taint flow graph G for app program analysis. Taint flow graph G statically captures the app behaviors with program analysis. We utilize G for a user-intention-guided taint flow analysis.

Definition 9. ***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 $f = \{v_0 \rightsquigarrow v_n\}$.*

We define rewriting specification R_r for app rewriting. The unit u is a line of code in the intermediate representation (IR). The op captures the rewriting operation towards unit u. The op is based on the sentimental value s in T_s. If $s = -1$, our rewriting would terminate the invocation of u if the invocation of u violates users' expectations. If $s = 1$, our rewriting would pass the verification of u, which has no impact on app behaviors. If an app contains multiple units in $R = \cup_{i=1}^{k} R_r^i$, we iteratively rewrite each unit u.

Definition 10. ***Rewriting Specification*** *is a tuple $R_r = \{u, op\}$ for rewriting an app A. u is a unit/statement in the app and op is an operation to rewrite this unit u. In rewriting, we have $A(R) \Rightarrow A'$, where $R = \cup_{i=1}^{k} R_r^i$, R_r^i is the rewriting specification for unit u_i and A' is the rewritten application.*

Example and Focus. In our running example, for the user intention predicate T_s, we have $p = \{Phone\}$ because the user considers most on the phone data, $s = -1$ because the user does not want to share the data, and $c = \{if\ it\ is\ sent\ out\ in\ text\ messages\}$. We find one unit u for the

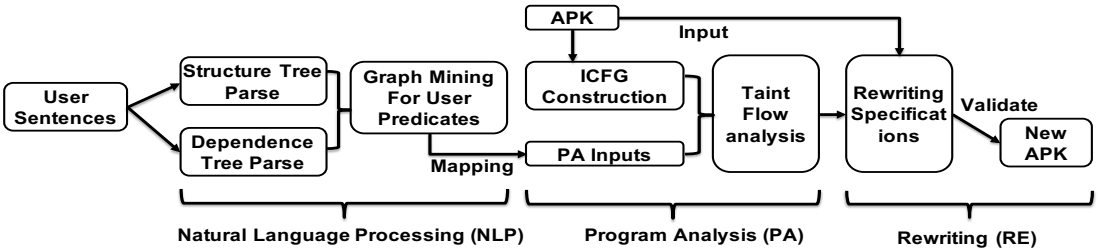


Figure 5.1: Workflow of our approach. Our approach consists of three major components: NLP is used for transforming natural sentence into user intention predicates, PA is used to perform user-intention-guided taint flow analysis, RE is used to automatically rewrite the app.

rewriting specification R_r, u is the exact line of code that sends out the device information, op is the operation to modify u. If u sends out the device information at runtime, the invocation of u is terminated.

Table 5.2 presents examples of user intention predicates. As explained in Section 5.2.1, mapping sentences to program inputs is difficult. If expressions can be interpreted in multiple ways, identifying such expressions needs additional constraints and conditions. For example, in the sentence "do not share my location when I am at work", it is non-trivial to generate code for intercepting "I am at work". The condition to verify whether a user is at work is not deterministic. Additional context (e.g., location range) is required. Currently, our approach only handles the structured sentences that can be mapped to sources and sinks for program analysis. We utilize static taint flow analysis to identify rewriting specifications.

5.2.3 Workflow

Figure 5.1 presents the workflow of our approach. We give an overview of the main operations, including extracting user intention predicate, user-intention-guided taint flow analysis and automatic rewriting.

- **Extract User Intention Predicates**. The goal of this operation is to extract user intention predicate T_s. The input is a natural language sentence S_{user}, and the output is the user intention predicate T_s. We generate two stages of sentence parsing trees. A structure parsing tree is used to analyze the sentence structure. A dependence parsing tree is used to analyze the semantic dependence relations. We utilize graph mining to extract T_s from parsing trees. Section 5.3 describes the algorithm for user intention extraction.

- **User-intention-guided Taint Flow Analysis**. The goal of this operation is to generate the rewriting specification R_r. The input is the user intention predicate T_s, and the output is the rewriting specification R_r. A static taint flow analysis is used to construct rewriting specifications. Section 5.4 describes the algorithm for user-intention-guided taint flow analysis.

- **Automatic Rewriting**. The goal of this operation is to automatically rewrite an app A to protect privacy. The input is the rewriting specification R_r, and the output is the rewritten app A'. An app is rewritten based on R_r. The control and data flow integrity are investigated to guarantee the validation of rewriting. Section 5.5 describes our rewriting strategies.

5.3 Extract User Intention with Natural Language Processing

The purpose of our NLP analysis is to identify user intention predicates. The output is a user intention predicate T_s in Definition 8. The input is a preprocessed sentence. The preprocesses include the sentence boundary annotating (e.g., handling period), stemming and lemmatization, removing punctuation and stopwords. We also parse Android API and permission documents to identify a predefined security object set P. We categorize security objects by considering both Android permissions and API document descriptions. For example, the "phone" category covers keywords from both READ_PHONE_STATE and READ_CONTACT permission. To filter out noises, we define a set of negative verbs V_n (e.g., block, disable) and positive verbs V_p (e.g., share, allow). The sentimental verb set is associated with user attitudes toward the security object. We filter out the sentence if it does not contain any sentimental verb. We utilize two stages of parsing trees: structure parsing tree T_{parse} for sentence structure analysis and semantic dependence parsing tree T_{depen} for semantic dependence analysis.

Structure Parser. We utilize T_{parse} to identify different structure levels of sentences, e.g., clause level and phrase level. Figure 5.2 shows how to identify the subordinate clause given a sentence S. The advantage of recognizing the subordinate clause is to increase the accuracy in identify the security object. In a sentence "stop sharing my location if it sent to other devices", the object is the only "location" in the main clause. The "device" is used as the constraint context for the object "location". A keyword-based solution incorrectly identifies both "location" and "device" as objects.

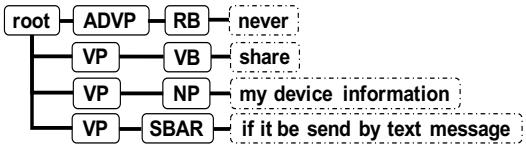


Figure 5.2: A simplified structure parsing tree annotated with structure tags. The tree is generated from the PCFG parser [79]. The bold rectangle node represents the part-of-speech (POS) tag. The dotted rectangle node represents the word and sentence. The edge represents a hierarchy relation from the root to words. S refers to a simple declarative clause, ADVP refers to adverb phase and VP refers to a verb phrase. The subordinate clause (SBAR) is identified correctly by the parser.

Semantic Dependence Parser. Semantic dependence tree T_{depen} represents semantic relations between words. Figure 5.3 presents an example of the semantic dependence tree in the main clause. In the graph T_{depen}, each term is followed by a part-of-speech (POS) tag and the type of semantic dependency. We utilize T_{depen} to construct the dependence path $\vec{P}$ from an object p. In Figure 5.3, the dependence path from object "device" is $\vec{P} = \{share \rightarrow information \rightarrow device\}$.

Word Semantic Similarity. Word semantic similarity is used for calculating the relatedness between a pair of words. Given two words w_1 and w_2, we define the similarity score as $sc = f_{sim}(w_1, w_2)$, where $sc \in [0, 1]$. $f_{sim}()$ is the function to compute similarity score between w_1 and w_2. We identify w_1 and w_2 are a pair of semantic similar words if sc is larger than a threshold. We utilize word semantic similarity to increase the feasibility of object matching in our approach.

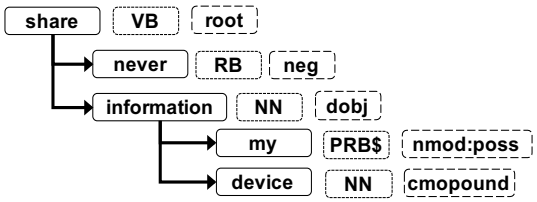


Figure 5.3: A example on the sentence annotated with semantic dependencies in the main clause. The tree is generated with Stanford-typed dependencies [40]. The arrow represents dependence relations. Each term is followed by two tags. The first tag is the part-of-speech (POS) tag, and the second tag represents the semantic relation (e.g., dobj means the direct object relation).

5.3.1 Graph Mining on Parsing Trees

The purpose of graph mining is to extract user intention predicate T_S. We extract $T_S = \{p, s, c\}$ from two stages of parsing trees.

Security Object Identification. To identify the object, we first identify the main clause S_{main} of the whole sentence S_{whole} by using the parsing tree T_{parse}. The subordinate clause S_{sub} is identified at the same time. We have $S_{main} \cup S_{sub} = S_{whole}$ and $S_{main} \cap S_{sub} = \emptyset$. For cases that users use the prepositional phrase (PP) as the constraint context. We also identify the prepositional phrase S_{pp} inside the main clause, where $S_{pp} \subseteq S_{main}$ and $S_{pp} \cap S_{sub} = \emptyset$. We identify an object p by recursively searching from the main clause to the subordinate clause. In the matching process, we aim to match both a word and its POS tag (NN). POS tags help us efficiently mark a word as noun, verb, adjective, etc. Word semantic similarity increases the tolerance in the matching. In Figure 5.3, we identify "device" as the object p in the category "phone", because "device" is a noun (NN as the POS tag) and shares a similar semantic meaning with "phone". In another case,

we do not detect "contact" as an object in "Siri, contact my mum that I am late for home". The POS tag of "contact" is VB (verb), it is used as an operation rather than an object. Our approach returns null if it cannot detect any object through all the steps.

Sentimental Intention. After identifying p, we construct the dependence path $\vec{P}_p = \{w_i | w_i \in S_{whole}\}$ in T_{depen}. We calculate the sentimental intention through the backward propagation in $\vec{P}_p$. We focus on two negative concepts: negative verbs V_n and negative relations R_n. Negative verbs represent the negative intention towards the object p. Negative relations represent the negative dependencies between two words. The negative relation is marked as "neg" in the dependence graph T_{depen}. The sentimental value s over an object p is computed by the total amount of negative concepts along the path $\vec{P}_p$. If the total amount is odd, we denote $s = -1$. If the amount is even (e.g, double negation), we denote $s = 1$. In Figure 5.3, the $s = -1$ because "share" and "never" remains a negative relation.

Constraint Detection. Constraint c provides additional restrictions on the object p. We identify the constraint c by extracting words in the subordinate clause or the prepositional phrase. Constraint c has a huge variance in different sentences. A simple sentence "please block my location" has no constraint for object "location". A complex constraint (e.g.,"when I am driving my car") has no directly inference for the taint flow analysis. In our approach, we focus on constraint c that demonstrates a sink function of the object p. For example, c= $\{by\ text\}$ means the function of sending text messages.

5.4 User-Intention-Guided Taint Flow Analysis

The purpose of the taint flow analysis is to generate a rewriting specification R_r in Definition 10. The input is the extracted user intention predicate T_s in Definition 8. T_s is semantically mapped to program inputs that can be used for taint analysis. The user-intention-guided taint flow analysis detects sensitive data flows that violate users' expectations. The output of the taint flow analysis is the sensitive sinks with privacy leak vulnerabilities.

Semantic Mapping. The taint flow analysis is guided by the user intention predicate T_s. We map T_s into API inputs, including sources and sinks. For $p \in T_s$, we map p to a set of sensitive sources. For $c \in T_s$, we map c to a set of sensitive sinks. Sources are associated with the security object p because private data is read from sources. Sinks are associated with the constraint c because private data is sent out by sinks. In the source mapping, we map p by its category into a predefined set of source APIs. In the sink mapping, we parse the sink list from Susi [103] to generate a lookup table $H(K) = T$, where K is a keyword set and T is the sink set. The lookup table maps a constraint c to a set of sinks $\bigcup_i t_i$, where each t_i is a sink function. We have $k \in c$ and $H(k) = \bigcup_i t_i$, where $k \in K$ and $t_i \in T$. Our approach returns all possible sinks if it cannot detect any matched constraint c.

Taint Flow Analysis. We generate rewriting specifications based on the taint flow analysis. The taint flow analysis is used to identify private data leak vulnerabilities. Reachability between a source and a sink is evaluated in the G_{icfg} and the output is a taint flow graph $G(V, E, S, T)$. A taint flow path $f = \{v_0 \rightsquigarrow v_n\}$ represents a potential data leak from v_0 to v_n. We generate rewriting

specifications based on a taint flow path f. We identify the unit u in $R_r = \{u, op\}$ as $u = v_n \in f$. The unit u is uniquely identified by matching three types of signatures: the class signature, the method signature and the statement signature. The class signature represents the class of a sink. The method signature represents the method in the class. The statement signature represents the exact line of code in a method.

5.5 Automatic Rewriting

The purpose of rewriting is to automatically modify the code based on a rewriting specification R_r. We develop a new rewriting framework to recognize rewriting specifications. Although there exist some rewriting solutions on Smali bytecode (e.g., I-ARM-Droid [51] and RetroSkeleton [49], we choose to reimplement our rewriting framework on Jimple for compatibility and reliability. Jimple is a three-address typed intermediate representation (IR) transformed from Smali by the Soot [83], which allows creation and reorganization of registers. Rewriting on Jimple is less error-prone than the direct modification on Smali. The modified IR is re-compressed as a new APK file and signed with a new certificate. The new APK remains the original logic and functions. Only sensitive sinks are rewritten for privacy protection enforcement.

In this chapter, we only demonstrate the use case of rewriting for privacy protection. The rewriting framework can be used for general purposes, e.g., repackaging and vulnerability mitigation. We propose three basic operations in our rewriting framework.

Unit Insertion. We would insert new local variables, new assignments, Android framework APIs

and calls to user defined functions into a method. We first need to inspect the method's control flow graph (CFG) and existing variables. We generate new variables for storing parameter contents and new units (e.g., add expression, assignment expression) for invoking additional functions. These new variables and units are inserted before u in R_r.

For Android framework APIs, we generate a new $SootMethod$ with the API signature, and insert it as a callee in the app. We generate an invoking unit as a caller to call the callee in the method. To guarantee the consistence of parameters of the API, we inspect the types of local variables and put them into the caller function in order.

For user defined functions in Java code, we generate a new $SootClass$ with the class and method signature. We insert the $SootClass$ as a new class. We invoke a user defined function in the method by generating a new invocation unit. The Java code is compiled into Jimple IR and attached to the app as a third-party library.

Unit Removal. Unit removal is more straightforward, as we directly remove the unit u in the original function. However, we need to guarantee that the removal unit has no-violation of the control-/data-flow integrity on the rest code. We analyze the CFG of the function to guarantee the validation of removing a unit.

Unit Edition. Unit edition modifies the unit u by changing its parameters, callee names or return values. We decompose u into registers, expressions and parameters, and modify them separately. For example, for a unit as a invocation $u = r_0.sendHttpRerquest(r_1...r_n)$, we decompose it as a register r_0, a function call $sendHttpRerquest$ and the parameters $\{r_1, ...r_n\}$. $sendHttpRerquest$

is replaced with $myOwnHttp$ by implementing a customized checking function. We are able to replace parameter r_0 with a new object r_{new} for monitoring network traffic.

We utilize basic operations to generate complex rewriting strategies. An if-else-condition can be decomposed as a unit insertion (initialize a new variable as the condition value), a unit edition (get condition value) and two unit insertions (if unit and else unit). Users can customize rewriting strategies in rewriting. In our approach, $op \in R_r$ is dependent on the sentimental value $s \in T_s$. If $s = -1$, we define our rewriting strategy is to insert a check function $check()$ before u. The check function terminates the invocation of u if privacy violation happens to u at runtime. This rewriting strategy enforces privacy protection with dynamic interruption.

The complexity of rewriting an app is $O(MN)$, where M is the number of units for rewriting and N is the lines of code. Given a rewriting specification R_r = $\{u, op\}$, the searching algorithm is linear by matching the unit in an app. The time for operation op is constant.

5.6 Limitation and Discussion

In our design, we mainly focus on the technical challenges for developing a natural language interface to programs for rewriting. User interfaces and legal issues (e.g., copyright restrictions) are out of the scope of discussion.

NLP Limitation. Our study shows higher precision in understanding user intentions than the state-of-the-art approach for app rewriting. In our prototype, we expect users express the sentence in

a structured format that can be parsed by our prototype. However, our approach still introduces some false positives and incorrect inferences. The inaccuracy most comes from abnormal ways of user presentations, e.g., ambiguous grammars and complex expressions. Also, the standard dependence parsing cannot correctly infer the semantic relations for complex sentences sometimes. The inaccuracy of dependence parsing further introduce wrong identifications of security objects and the sentimental value. In the future work, more efforts are required to better resolve above limitations.

Taint Analysis Limitation. Our prototype is built on the static program analysis framework [25]. The inaccuracy of our prototype comes from the inherent limitations of the static analysis. Static analysis cannot handle dynamic code obfuscation. Static analysis often over-approximates taint flow paths. The detected taint flow paths may not be feasible at runtime. We mitigate limitations of static analysis with an efficient rewriting mechanism. Our rewriting framework can terminate a function's invocation only when privacy violation happens. We plan to extend our prototype with a more capable hybrid analysis. The hybrid analysis enhances the resistance of dynamic obfuscations.

Rewriting Limitation. Our rewriting framework provides multiple operations for app customization. The research prototype is based on the Soot infrastructure with the Jimple intermediate representation. The Jimple IR is extracted from Android Dex bytecode by app decomposition. We recompile Jimple IR into Dex bytecode after rewriting. ART and Dalvik virtual machines [3] are compatible to run Dex bytecode. The efficiency of rewriting relies on app decomposition. Current

[3] `https://goo.gl/EvdMdO`

decomposition frameworks [57, 97] cannot handle native code and dynamic loaded code. Similar to static analysis, we cannot rewrite bytecode that is not compatible to Jimple. How to detect dynamically loaded code is studied recently in [100]. In the future work, we plan to enhance rewiring by increasing decomposition capacity.

5.7 Experimental Evaluation

We present the evaluation of our approach in the section. We list four research questions in our evaluation. **RQ1:** What is the accuracy of *IronDroid* in identifying user intentions? (precision, recall and accuracy) **RQ2:** How flexible is *IronDroid* in rewriting apps? (success rate and confirmation of modified behaviors) **RQ3:** What is the performance overhead of *IronDroid* ? (size overhead and time overhead) **RQ4:** How to apply *IronDroid* for practical security applications? (security applications)

To answer the research questions, we evaluate the accuracy of user intention extraction in Section 5.7.2. We evaluate the feasibility of rewriting in Section 5.7.3. We measure the performance of our approach in Section 5.7.4 We provide three case studies in Section 5.7.5.

5.7.1 Experiment Setup

We implement our research prototype by extending StanfordNLP library for user intention analysis. We utilize PCFG parsing [79] for structure parsing and Stanford-Typed dependencies [40] to

detect semantic relations. We use WordNet [98] similarity to compute the relatedness between two words. Our taint flow analysis is based on the cutting-edge program analysis tool FlowDroid [25]. We extend FlowDroid to support natural language transformation and rewriting specification generation. We implement our own Android rewriting framework with Jimple IR based on Soot.

To evaluate the accuracy of user intention analysis, we evaluate totally 153 sentences collected from four users with different security backgrounds. Users express their concerns on the personal information. The average length of a sentence is 10. 30.1% of sentences contain subordinate clauses. We have two volunteers to independently annotate each sentence in the corpus. Each sentence is discussed to a consensus. We use the corpus as ground truth for user intention analysis. For the taint flow analysis and app rewriting, we evaluate benchmark apps (total 118 apps) from DroidBench. We dynamically trigger sensitive paths in rewritten apps for validation. We compute time and size overhead for the performance evaluation. In security applications, we show how to potentially apply our approach to privacy protection. We also demonstrate how to achieve a more fine-grained location restriction. The location restriction is beyond the current Android dynamic permission mechanism.

5.7.2 RQ1: User Intention Accuracy

The output predicates $T_s = \{p, s, c\}$ is a set of tuples. We mostly focus on the accuracy of the user object p and the user sentimental value s. These two concepts are most important to express user concerns and attitudes. The constraint c has a huge variance and is hard to perform standard

statistics evaluation. We compare precision with following models:

- **StandfordNLP** includes a basic sentiment model for sentence-based sentimental analysis. In our problem, StanfordNLP does not provide the security object identification.
- **Keyword-based Searching** identifies the security object and the sentimental value by keyword matching in the sentence. Keyword-based searching ignores the structure and semantic dependencies. Keyword-based searching is regarded as the state-of-the-art analysis in user intention identification.
- ***IronDroid*** captures the insight of sentence structure and semantic dependence relations. *IronDroid* identifies the object and the sentimental value via graph mining from two stages of parsing trees. *IronDroid* also utilizes the semantic similarity for word comparison.

For a scientific comparison, the keyword-based searching and *IronDroid* share the same predefined object set. We identify categories of objects: $\{$*phone, calendar, audio, location*$\}$. The four categories present a high coverage of private data for mobile app users. We measure the precision, recall and accuracy of user sentimental value identification. TP(true positive), FP(false positive), TN(true negative), and FN(false negative) are defined as:

1. TP: the approach correctly identifies the user sentimental value as negative (“not sharing”).
2. FP: the approach incorrectly identifies the user intention value as negative (“not sharing”).
3. TN: the approach correctly identifies the user intention value as positive (“sharing”).

4. FN: the approach incorrectly identifies the user intention value as positive ("sharing").

We further define precision (P), recall (R) and accuracy (Acc) as:

$$P = \frac{TP}{TP + FP}, R = \frac{TP}{TP + FN}, Acc = \frac{TP + TN}{TP + FP + TN + FN} \tag{5.1}$$

IronDroid achieves the highest accuracy than the other two approaches. Table 5.3 presents the sentimental analysis of three different approaches, our approach achieves 95.4% accuracy in identifying user sentimental value. StandfordNLP achieves very low accuracy 68.6% in the experiment. We believe the standard NLP model is not suitable for our problem. In our problem, we compute the sentimental value towards the security object. StandfordNLP computes the sentimental value for the whole sentence. There is no direct correlation between two different sentimental levels. For example, standfordNLP identifies "I do care, please block the location information" and "it is okay to share my location information" with the same optimistic/positive attitude. However, the first sentence presents a negative intention (blocking) for the object "location". Therefore, we cannot directly apply standfordNLP for our problem. Keyword-based searching fails because of the lack of insight in sentence structure and semantic dependence. For example, "do not share my location if I am not at home" is misclassified by the keyword-based searching. The main reason is that "not" in the subordinate clause has not direct semantic dependence on the main clause. Our approach has more insights in capturing sentence structures and semantic dependence relations. In summary, our approach is very accurate in understanding user intentions.

Table 5.4 presents the improvement of *IronDroid* comparing with the keyword-based searching for the object identification. For each category, we define the improvement in precision, recall and

accuracy as:

$$\begin{aligned} \triangle\ P &= P_{Ucer} - P_{keyword} \\ \triangle\ R &= R_{Ucer} - R_{keyword} \\ \triangle\ Acc &= Acc_{Ucer} - Acc_{keyword} \end{aligned} \tag{5.2}$$

How to compute the precision, recall and accuracy for each object can be easily inferred as the user sentimental evaluation. Our results show that, comparing with keyword-based searching, *IronDroid* effectively identifies objects with the average increase in precision, recall and accuracy of 19.66%, 28.23% and 8.66%. The improvement comes from two major reasons: word semantic similarity to increase true positives and dependence analysis to reduce false positives. Keyword-based searching introduces large false positives by over-approximating security objects. For example, in the sentence "I do not want share my calendar information if it is sent to other devices", keyword-based searching identifies both "calendar" and "device" as security objects because of the lack of structure analysis.

Table 5.5 presents the overall accuracy for user intention identification. The evaluation is based on the combination of the sentimental intention s and the object p. We mark the result $\{p, s\}$ true only when both s and p is identified correctly. *IronDroid* achieves 90.2% accuracy rate, while keyword-based approach only achieves 53.6% accuracy rate. Our approach has nearly 2-fold improvement than the state-of-the-art approach. The experimental results validate that our approach is effective in identifying user intentions from natural languages.

	TP	**FP**	**TN**	**FN**	**P**(%)	**R**(%)	**Acc**(%)
StandfordNLP	55	18	50	30	75.3	64.7	68.6
Keyword-based	64	9	63	17	87.7	79.0	83.0
Ours	68	5	78	2	**93.2**	**97.1**	**95.4**

Table 5.3: Sentimental analysis accuracy for *IronDroid* , StanfordNLP and keyword-based searching. *IronDroid* achieves higher accuracy, precision and recall than the other two approaches. our approach effectively achieves the improvement in 19.66% for precision, 28.23% for recall and 8.66% for accuracy on average.

Object	△**P** %	△**R**%	△**Acc**%
Location	19.23	14.52	11.11
Phone	20.00	38.15	9.80
Calendar	26.09	28.57	7.84
Audio	13.33	31.66	5.88
Average	**19.66**	**28.23**	**8.66**

Table 5.4: object detection improvement comparing with keyword-based approach. Our approaches have the average increase in precision, recall and accuracy of 19.66%, 28.23% and 8.66% for four categories of objects.

	Keyword-based	***IronDroid***
Accuracy	53.6%	**90.2**%

Table 5.5: The overall accuracy for understanding sentimental and the object. Our approach has a significant improvement than the keyword based searching approach.

5.7.3 RQ2: Rewriting Robustness

DroidBench is a standard test suite for evaluating the effectiveness of Android program analysis [25, 64]. DroidBench is specifically designed for Android apps. We tested our rewriting capability based on the benchmark apps. We use the example "never share my device information, if it is sent out in text messages" as the input for user intention analysis. We generate the user intention predicate T_s as p = "Phone", $s = -1$ and c ="sent out in text messages". In the user-intention-guided taint flow analysis, we map T_s to a list of code inputs. Table 5.6 presents how we map "Phone" into source APIs and "text" into sink APIs. We perform the taint flow analysis in 13 different categories among 118 apps. Each category represents one particular evasion technique (e.g., callback). Without loss of generality, we randomly choose an app for rewriting in each category. The tested apps have a high coverage to mimic real-world apps. We identify a taint flow from a source to a sink within the taint flow graph. We generate the rewriting specification R_r by extract the unit u of the sink. For the rewriting operation op, we use unit removal to remove the sensitive sink API. We use unit insertion to add a dynamic logging function. The sink API is the leaf node in the ICFG, the unit removal has no impact on the rest code. To evaluate the feasibility of the rewriting, we install all the rewritten apps in a real-world device Nexus 6P with Android 6.0.1. We aim to answer two major questions:

1. Whether these apps remain valid program logics (do not break the functionality)?

2. Whether we would observe the modified behavior at runtime (protect privacy)?

To answer the first question, we install rewritten apps on the device. We utilize $monkey$ tool to

generate a pseudorandom stream of user events. We generate 500 random inputs in testing one app. We perform the stress test on the rewritten app. To answer the second question, we need to trigger taint flow paths and observe the modified behaviors. We use $logcat$ to record the app running state. The dynamic logging function $Log.e()$ can be captured by $logcat$ at runtime. To increase the coverage of user interactions, we also manually interact with the app, e.g., clicking a button. If we observe such log information, the behavior of an app is modified and we validate our rewriting on the app.

Table 5.7 presents the robustness of our rewriting. Our approach successfully runs and detects all the rewritten apps. For example, we detect a taint flow path as $f = \{getDeviceId() \rightsquigarrow sendTextMessage()\}$ in $Button1.apk$. Our user-intention-guided taint flow analysis successfully captures the sensitive taint path triggered by a callback $onClick()$. Our rewriting technique identifies the taint flow path f and the rewriting unit $sendTextMessage()$. We modify the code in the app without violates its control and data dependence. These steps guarantee the success of our rewriting technique.

5.7.4 RQ3: Performance Overhead

We compare the runtime overhead of natural language processing (NLP), taint flow analysis (PA) and rewriting (RE) in Figure 5.4. Experiments were performed over on a Linux machine with Intel Xeon CPU (@3.50GHz) and 16G memory. Figure 5.4 presents the three runtime distributions in log scale. The average time for NLP is 0.16 second. The average time for PA is 1.43 seconds. The

Category	Mapping
Device info	getDeviceId(),getSubscriberId(), getSimSerialNumber(), getLine1Number()
Sent by text	sendTextMessage(),sendDataMessage(), sendMultipartTextMessage(),sendMessage()
Never share	unit removal (sink unit), unit insertion (log unit)

Table 5.6: Mapping the sentence to analysis inputs for rewriting in the experiment. The sentence is from the example “Never share my device information, if it is sent out in text messages”.

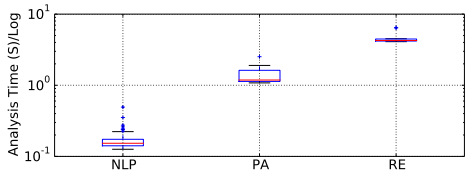


Figure 5.4: Time overhead for analysis, the average time for NLP analysis is 0.16 second, the average time for PA analysis is 1.43 seconds, the average for RE is 4.68 seconds.

average time for RE is 4.68 seconds. The average runtime of RE is larger than that of NLP and PA. RE needs to decompile each class for inspection and recompile the app after rewriting.

Figure 5.5 presents the size overhead of rewriting benchmark apps. The size overhead comes from two major sources: 1) the complexity of rewriting specifications; 2) the number of impacted code in rewriting. In the experiment, the average file size of the benchmark apps is 305.23 KB. Our approach achieves 3.3% size overhead on average, which is relatively negotiable. In summary, our approach is very practical in rewriting apps.

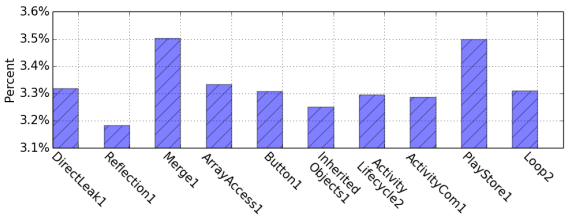


Figure 5.5: Size Overhead for benchmark apps, our rewriting introduces 3.3% percent overhead on file size for benchmark apps.

5.7.5 RQ4: Security Applications

In this section, we show how to extend *IronDroid* for real-world problems. We present three security applications that cover 1) proactive privacy protection, 2) mitigating ICC vulnerability and 3) location usage restriction. These security applications demonstrate the feasibility of our rewriting framework. Table 5.8 presents the statistics of three demo apps.

Proactive Privacy Protection

The purpose of this demo is to show the rewriting flexibility for normal users. $com.jb.azsingle.dcejec$ is an e-electronic book app. The app passes all the 56 anti-virus tools for vetting screening. The user concerns on the data leakage to the storages. Users can proactively restrict the data usage. The sentence is that "do not reveal my phone information to outside storages". We use this example to show how a user could use *IronDroid* for a proactive privacy protection. Based on the our NLP analysis, we identify user intention predicate T_s = {"phone", -1, "outside storage"}. The user-intention-guided taint flow analysis identifies a critical taint flow graph f_1 in a class

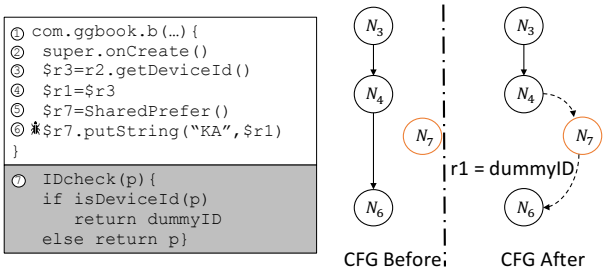
$com.ggbook.i.c$. The taint flow represents that the phone information is written into a file $KA.xml$. The sink unit is a sensitive function $putString()$. $KA.xml$ is located in the device storage. The taint flow path f_1 is presented as below:

- f_1: [r3 = invoke r2.<TelephonyManager: String getDeviceId()>(), r4 = r3, return r4, <com.ggbook.c: String z> = r1, r1 = <com.ggbook.c: String z>, r7 = invoke r7.<SharedPreferences: putString()>("KA", r1)]

Our rewriting is based on the unit that invokes the $putString()$ function. The unit is identified as r7 = invoke r7.<SharedPreferences: putString()>("KA", r1). In rewriting, we implement a dynamic checking function $IDcheck()$. $IDcheck()$ accepts the parameter $r1$ from the unit. The type of $r1$ is Java.lang.String. $IDcheck()$ dynamically checks $r1$ by comparing the value with the actual device ID $p = getDeviceId()$. If $p = getDeviceId() \in r1$, $IDcheck()$ returns a dummy ID and overwrite the parameter $r1$. The dummy ID can be further used without violating control flow dependencies. Figure 5.6 presents the code snippet and the CFG after we insert the $IDcheck()$ function.

To validate our rewriting feasibility and dummy ID insertion. We run the app with $monkey$ on a real device. We manually find the shared preference file in the app file directory. The shared preference file is stored in `/data/data/com.jb.azsingle.dcejec/shared_prefs`, we identify the dummy ID as 12345678910. The entry $IsImei = true$ suggests that the dummy ID is successfully inserted. The private phone data is protected from outside storages.

```
<map><string name="KA">12345678910</string>
```



(a) A simplified code structure of the method. The IDcheck function is inserted after rewriting.

(b) The before and after CFG. The CFG is changed. The privacy is protected by rewriting.

Figure 5.6: The demo of code snippet and the modified control flow graph (CFG). We insert a IDcheck() function as a new node in the original CFG.

```
<boolean name="IsImei" value="true"/></map>
```

Mitigating ICC Vulnerability

The purpose of this demo is to present the rewriting feasibility on mitigating vulnerabilities. In this example, we focus on preventing inter-component communication (ICC) vulnerability. Existing detection solutions aim at detecting vulnerable ICC paths [54, 96] with static program analysis. How to avoid realistic ICC vulnerabilities at runtime is still an open question.

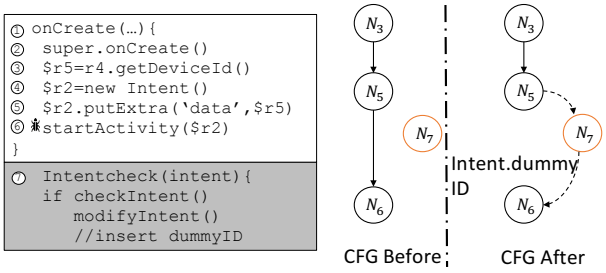
We apply *IronDroid* to showcase a practical mitigation solution to prevent ICC vulnerability. $com.xxx.explicit1$ is an app in ICCBench[4]. The app contains a typical ICC vulnerability. The

[4] https://goo.gl/EPjb3n

sensitive phone data (IMEI) is read from a component. The IMEI is put into a data field in an intent. The data is sent out to another component by the intent. The receiver component has access to IMEI without acquiring for the permission READ_PHONE_STATE. The ICC vulnerability results in a privilege escalation for the receiver component.

Our rewriting is based on a user command: "do not share my device information with others". The user-intention-guided taint flow analysis identifies a critical taint flow as f_2 in a class $explicit1.MainActivity$. The taint flow path f_2 is presented as:

- f_2: [r5 = invoke r4.< TelephonyManager: getDeviceId() > (), invoke r2.< putExtra(String,String)>("data",r5), invoke r0.< startActivity(android.content.Intent) >(r2)]



(a) A simplified code structure of the method. The IDcheck function is inserted after rewriting.

(b) The before and after CFG. The CFG is changed and the vulnerability is mitigated by rewriting.

Figure 5.7: The demo of code snippet and the modified control flow graph (CFG). We insert a Intentcheck() function as a new node in the original CFG.

Our rewriting specification focuses on the unit that invokes the $startActivity$ function. $startActivity$

is used for triggering an ICC. The intent $r2$ is an object that consists of the destination target (e.g., the package name) and data information (e.g., extras). Replacing intent would cause runtime exceptions. The communication of components is based on intents. The component that can receive the intent is defined in the package field of an intent. We aim to protect user privacy without interrupting communication channels. In rewriting, we implement a hierarchy checking function $Intentcheck()$. $Intentcheck()$ accepts the parameter of an intent and modifies the intent in-place. $Intentcheck$ can recursively check all the fields in an intent and rewrite the intent. In this example, we rewrite an intent by inserting dummy data in the data field. Figure 5.7 presents the code snippet and the CFG after we insert the $Intentcheck()$ function.

To validate the modified intent, we run the app with $monkey$ on a real device and manually intercept the receiver component. We successfully detect the dummy ID as 12345678910 in the data field of $getIntent()$.

The above example demonstrates how a normal user could use *IronDroid* to reduce ICC vulnerability. For better privacy, our rewriting is also capable of redirecting an intent. One possible way to redirect an intent for security is changing an implicit intent to an explicit intent. In the explicit intent, the destination of the intent can be verified by our rewriting. We plan to extend our approach to mitigate other vulnerabilities, e.g., adversarial advertisement libraries.

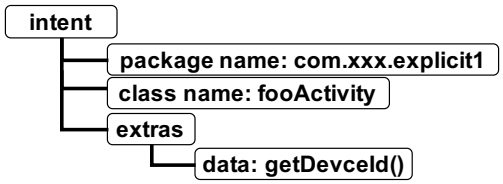


Figure 5.8: The hierarchy of an intent object. The privacy data getDdeviceId() is stored with a key $data$ in a HashMap in $extras$ filed.

Location Usage Restriction

The location is a primary concern for most users. $com.bdsmartapp.prayerbd$ is an app to provide location-based service. It is unclear whether the app would abuse the location and send it to untrusted websites. The purpose of this demo is to show a fine-grained control of location usage. Current Android dynamic permission mechanism only provides turning on and off of the location permission. Our approach achieves extended functionalities: 1) Target inspection, a user can inspect a targeted function (e.g., to web servers, by text messages and to other apps). 2) Destination inspection, a user can inspect whether the packet is sent to an untrusted destination. 3) Sensitivity restriction, a user can protect sensitive locations by sending dummy locations.

We extend our approach to providing a fine-grained restriction with a $LocationCheck$ function. We rewrite the app based on the user sentence "do not send my home location to untrusted websites". We extracted user intention predicate T_s ={"Location", -1, "to untrusted website" }. The security object p = {"Location"} is mapped to APIs for reading locations (e.g., $getLatitude()$ and $getLongitude()$). The the constraint c = {"to untrusted website" } is mapped to APIs for generating network requests (e.g., $URLConnect()$). In addition, we implement a function $Locationcheck$

to determine 1) the home location and 2) the trustworthiness of a website. Advertisement websites (e.g., Admob, Flurry) are regarded as untrusted. The home location is defined by a location range. If the URL contains untrusted addresses or the current location is in the location range, the URL is modified by $Locationcheck$ for privacy protection.

At runtime, a location request `googleapis.com/maps/api/geocode/json?latlng=xxx,xxx&sensor=true` is captured by $Locationcheck$. The location request is sent to the Google map server with $googleapis$. $Locationcheck$ inspects the latitude and longitude in the location request. $Locationcheck$ replaces the original location with a dummy latitude and longitude. A dummy location is shown on the screen layout when we test the rewritten app. In this demonstration, *IronDroid* protects user private location by supporting function-level restriction. More advanced version of our approach can be combined with anomaly data detection solutions.

5.8 Conclusions and Future Work

We investigated the problem of app customization for personalized security. We proposed a user-centric app customization framework with natural language processing and rewriting. Our preliminary experimental results show that our prototype is very accurate in understanding user intentions.

We show its applications in providing more fine-grained location control and mitigating existing vulnerabilities. Our approach makes impressive progress toward personalized app customization. More efforts are needed to improve the usability of personalized app customization. The computation could be further optimized using parallel platforms with vector processing (or SIMD) [68, 70,

71,73,91,114,120,137,139]. Moreover, manycore GPUs [66,67,69,72,134–136,140–142] and the reconfigurable devices [95, 127, 138] could also be promising platforms to boost the performance of the computation shown in the dissertation. For future work, we plan to improve the usability of our prototype with more engineering efforts.

Category	**AppName**	**PA time (s)**	**RE time (s)**	**Run**	**Logging Confirm**
Alias	Merge1	1.74	6.40	✓	✓
ArrayList	Array Access1	1.13	4.34	✓	✓
Callbacks	Button1	2.52	4.48	✓	✓
FieldandObjctSensitivity	Inherited Objects1	1.27	4.24	✓	✓
InterAppCom	N/A	-	-	-	-
InterComponentCom	Activity Communication1	1.08	4.51	✓	✓
LifecyCycle	ActivityLifecycle2	1.15	4.20	✓	✓
GneralJava	Loop2	1.14	4.18	✓	✓
Android Specific	DirectLeak1	1.11	4.10	✓	✓
ImplicitFlows	N/A	-	-	-	-
Reflection	Reflection1	1.22	4.27	✓	✓
Threading	N/A	-	-	-	-
Emulator Detect	PlayStore1	1.90	6.46	✓	✓
Summary	10			**10**	**10**

Table 5.7: Runtime scalability for testing benchmark apps. N/A means we did not find any app matched the user command in the taint analysis. Run w. monkey means the rewritten apps can run on the real-world device Nexus 6P. We use $monkey$ to generate 500 random inputs in one testing. Confirmation means we detect the modified behavior using the adb $logcat$. PA is short for taint-flow-based program analysis time, RE is short for automatic rewriting time.

Package	Report	Label
com.jb.azsingle.dcejec	0/56	benign
com.xxx.explicit1	1/57*	grayware
com.bdsmartapp.prayerbd	0/56	benign

Table 5.8: All three apps pass the vetting screening of anti-virus tools. * means the alert mentions it contains a critical permission $READ\_PHONE\_STATE$ without any additional information. We identify the second app as grayware [22]. Users have the preference for customizing these apps for personalized security.

Chapter 6

Web Security with Malicious Iframe Detection

6.1 Introduction

The past decade has seen the strong trend of content consolidation in web technologies. Nowadays, a web page delivered to a user usually contains content pulled from many third-parties. A typical web primitive employed by website developers for this purpose is Iframe tag, which automatically renders web content in a container within a webpage. It gains popularity since the dawn of web and is still one major technique driving the Internet economy. Although Iframe facilitates the third-party content rendering, it introduces potential abuse. A recent take-down operation against Rig Exploit Kit shows Iframe is the major "glue" for its infrastructure [12]. After an attacker

gains control over a vulnerable website, Iframe is usually injected into a webpage to make the site a gateway to attacker's infrastructure. Every time a user visits the compromised wepage, she is redirected to other website that hosts malicious payload.

Although malicious Iframe usage could be dangerous, Iframe injection characteristics and its countermeasure are not well studied. To fill this gap, we performed a large-scale analysis on *Alexa top 1 million* websites to understand how Iframe is injected in both offline (embedded through Iframe tag) and online (generated through JavaScript) scenarios. We find Iframe inclusion is widely used by legitimate site owners. In particular, offline inclusion is more popular comparing to online inclusion, covering 30.8% of the pages returned to our crawler. A closer look into these Iframes shows a large portion of them point to several giant IT companies serving advertisements, social networks and web analytics. Techniques used extensively for injecting malicious Iframes have limited adoption in legitimate websites, like hidden style (except by several well-known third-parties) and obfuscation. On the downside, the support for browser policies, like CSP and `X-Frame-Options`, is still insufficient among site owners, though these policies could contain the damage caused by Iframe injection.

Because of the limited website protection, Iframe injection has already become a powerful weapon in hacker's arsenal [12]. Injected Iframes in compromised websites can point to arbitrary attacker's infrastructure for malware propagation. Therefore, we believe a system capable of pinpointing and classifying the Iframe injection and is very important in guarding the safety of web users and integrity of websites.

In this chapter, we propose a new detection system named `FrameHanger` [119] to mitigate the

threat from Iframe injection. The system is composed of a static analyzer against offline injection and a dynamic analyzer against online injection. To counter the obfuscation and environment profiling heavily performed by malicious Iframe scripts, we propose a new technique called *selective multi-execution*. After an Iframe is extracted by `FrameHanger`, a machine-learning model is applied to classify its intention. We consider features regarding Iframe's style, destination and context. The evaluation result shows the combination of these features enables highly accurate detection: 0.94 accuracy is achieved by the static analyzer and 0.98 by the dynamic analyzer.

While prior works can detect webpages tampered by hackers [32, 37, 45, 87, 92, 102], they either work at coarse-grained level (page- or URL-) or require a clean reference to perform differential analysis. On the contrary, `FrameHanger` is able to spot the malicious Iframe at *tag-level* without the dependency on any reference. As such, by applying `FrameHanger`, finding Iframe injection should become less labor-intensive and error-prone. We release source code of components of `FrameHanger`, in hopes of propelling the research on countering web attacks [6]. The contributions of the chapter are summarized as follows:

- We propose a new *tag-level* detection system `FrameHanger` for malicious Iframe injection detection by combining selective multi-execution and machine learning. We design multiple contextual features, considering Iframe style, destination and context properties.
- We implement the prototype of `FrameHanger`, which contains a static analyzer and a dynamic analyzer. The experimental results demonstrate the high precision of `FrameHanger`, i.e., 0.94 accuracy for offline Iframe detection and 0.98 for online Iframe detection.

- We carried out a large scale study on Iframe injection, in both legitimate and malicious scenarios.

6.2 Background

In this section, we give a short introduction about Iframe tag, including its attributes and capabilities first. Then, we describe how Iframe is abused by network adversaries to deliver malicious content.

6.2.1 Iframe Inclusion

HTML Frame allows a webpage to load content, like HTML, image or video, independently from different regions within a container. There are four types of Frame supported by mainstream browsers, including `frameset`, `frame`, `noframe` and `iframe`. Except Iframe, all the others were deprecated by the current HTML5 standard. In this work, we focus on the content inclusion through Iframe.

Iframe attributes. How Iframe is displayed depends on a set of tag attributes. Attribute `height` and `width` determine the size of Iframe. The alignment of content inside Iframe can be configured by `marginheight`, `marginwidth` and `align`. For the same purpose, the developer can assign CSS properties into `style`, which provides more handlers for tuning the Iframe display. For instance, the position of Iframe within the webpage can be adjusted through two properties of

`style` (`top` and `position`). When the Iframe has a parent node in the DOM tree, how it is displayed is influenced by the parent. The origin of Iframe content is determined by `src` attribute, which is either a path relative to the root host name (e.g., /iframe1.html) or an absolute path (e.g., http://example.com/iframe1.html). Figure 6.1 gives an example about what an Iframe tag looks like.

Browser policies. One major reason for the adoption of Iframe inclusion is that its content is isolated based on browser's Same Origin Policy (SOP) [13]. Bounded by its origin, the code within an Iframe is forbidden to access the DOM objects from other origins, which reduces the damage posed by third parties. However, the threat is not mitigated entirely by this base policy. All DOM operations are allowed by default within the Iframe container. To manage the access at finer grain, three mechanisms have been proposed:

- **`sandbox` attribute.** Starting from HTML5, developers can enforce more strict policy on Iframe through the `sandbox` attribute, which limits what can be executed inside Iframe. For instance, if a field `allow-scripts` is disabled in `sandbox`, no JavaScript code is permitted to execute. Similarly, submitting HTML form is disallowed if `allow-forms` is disabled.

- **CSP (Content Security Policy).** CSP provides a method for site owner to declare what actions are allowed based on origins of the included content. It is designed to mitigate web attacks like XSS and clickjacking. For one website, the policy is specified in the `Content-Security-Policy` field (or `X-Content-Security-Policy` for older

policy version) within server's response header. CSP manages Iframe inclusion through attributes like `frame- ancestors`, which specifies the valid origin of Iframe.

- **`X-Frame-Options`.** This is also a field in the HTTP response header indicating whether an Iframe is allowed to render. There are three options for `X-Frame-Options`: `SAMEORIGIN`, `DENY` and `ALLOW-FROM`.

Though these primitives could help mitigate the threat from malicious Iframe, we found they were not yet widely used, as shown in Section 6.4.3. What's more, when the attacker is able to tamper the response header in addition to page content, all these policies can be disabled.

6.2.2 Malicious Iframe Injection

The capabilities of Iframe are bounded by different browser policies like SOP, but it is still extensively used by attackers to push malicious content to visitors, according to previous research [102] and reports [9, 12]. In many cases, after attacker compromises a website and gains access, an Iframe with `src` pointing to a malicious website is injected to webpages under the compromised site. Next time when a user visits the compromised site, her browser will automatically load the content referred by the malicious Iframe, like drive-by-download code. In fact, it is not unusual to find a large number of websites hijacked to form one malware distribution network (MDN) through Iframe Injection [102]. So far, there are mainly two mechanisms used widely for Iframe injection and we briefly describe them below.

Offline Iframe injection. In this case, the attacker injects the Iframe tag to the compromised

```
1 <p><span style="position: absolute; top: -1175px; width: 312px; height:
  306px;"> npz
2 <iframe src="http://gfd.JOSEPHANDRITO.COM/?
  q=wHnQMvXcJwDGFYbGMvrESqNbNknQA0OPxpH2_drWdZqxKGni0Ob5UU
  Sk6FSCEh3&amp;que=border.102ky68.406r3r5s3&amp;biw=border.111tk83.4
  06j4t3y2&amp;ct=border&amp;fix=border.98yv62.406y1w9o4&amp;oq=h9_Eq
  LbZROALjiBOJcwJnnY5fVQlA8qisi0iAyhDKicTQ-ByMZg91z6LRVvQ-2w"
  width="258" height="261"></iframe>
3 yqpwc </span>htkz</p>
4 <noscript>
5 <!DOCTYPE html >
6 <html lang="es">
7 <head>
8     <base href="http://www.selfprinting.es/" />
9 </noscript>
```

Figure 6.1: An example of offline Iframe injection. Line 1-4 are added by attacker. The Iframe can only be detected in the HTML source code. Visitors are unable to see this Iframe from the webpage.

```
1 </script>
2 <body> </body>
3 <script type="text/javascript">
4 var hhfcdro = "iframe"; var mbusui = document.createElement(hhfcdro); var
  qqdbctd = ""; mbusui.style.width = "8px"; mbusui.style.height = "12px";
  mbusui.style.border = "0px"; mbusui.frameBorder = "0";
  mbusui.setAttribute("frameBorder", "0"); document.body.appendChild(mbusui);
  qqdbctd = "http://one.bestwingsinmemphis.com/?
  qtuif=2139&ct=soul&q=w3nQMvXcJxfQFYbGMvPDSKNbNkzWHViPxoqG9Mil
  dZ-qZGX_k7HDfF-
  qoV_cCgWR&oq=xfFf7tZNAOyikWJfQlznIxeUltBpfimj0PRyR_K1pKFrByJZA9
  H-qKlJLd_mhj2"; mbusui.src = qqdbctd; </script>
5 </body>
6 </html>
```

Figure 6.2: An example of online Iframe injection. Line 3-4 are added by attacker.The Iframe is injected by running JavaScript code. Visitors are unable to notice the injection and see the Iframe from the webpage.

webpage without any obfuscation. Figure 6.1 illustrates one real-world example. To avoid visual detection by users, the attacker sets `top` field of `style` to a very large negative value (`-1175px`). Instead of setting `style` in the Iframe tag, the attacker chooses to set it on the parent `span` node. Also interesting is that the injected Iframe appears at the very beginning of the HTML document, a popular pattern also mentioned by previous works [87]. The URL in `src` points to a remote site which aims to run Rig Exploit code in browser [9].

Online Iframe injection. To protect the Iframe destination from being easily matched by URL

blacklist, attacker generates Iframe on the fly when the webpage is rendered by user's browser. Such runtime generation is usually carried out by script injected by attacker, like JavaScript code. We call such code *Iframe script* throughout the chapter. Figure 6.2 illustrates one example. The code within the script tag first creates an Iframe tag (`createElement`) and adds it to the DOM tree (`appendChild`). Then, it sets the attributes of Iframe separately to make it invisible and point to malicious URL.

The sophistication of this example is ranked low among the malicious samples we found, as the malicious URL is in plain text. By applying string functions of JavaScript, the URL can be assembled from many substrings in the runtime. Recovering the Iframe URL will incur much more human efforts. Even worse, attacker can use off-the-shelf or even in-house JavaScript obfuscator [10, 48, 111] to make the entire code unreadable and impede the commonly used analyzers. We show examples of advanced destination-obfuscation in Section 6.6.3.

6.3 Detecting Iframe Injection

Iframe is used extensively to deliver malicious web code, e.g., drive-by-download scripts, to web visitors of compromised sites since more than a decade ago [102]. How to accurately detect malicious Iframe is still an open problem. The detection systems proposed previously work at *page-*, *URL-* or *domain-level*. Given that many Iframes could be embedded by an individual page, *tag-level* detection is essential in saving the analysts valuable time for attack triaging. To this end, we propose `FrameHanger`, a detection system performing tag-level detection against Iframe injec-

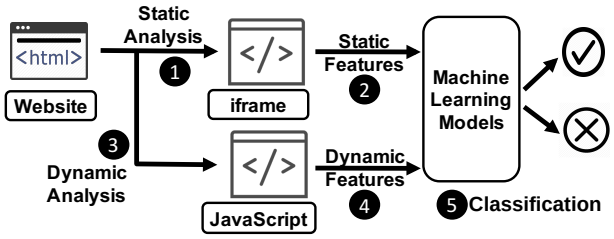


Figure 6.3: The framework of `FrameHanger`.

tion. We envision that `FrameHanger` can be deployed by a security company to detect Iframe injection happening on any compromised website. In essence, `FrameHanger` consists of a static crawler to patrol websites, a static analyzer to detect offline injection and a dynamic analyzer to detect online injection. In what follow, we provide an overview of `FrameHanger` towards automatically identifying Iframe injection (the crawler component is identical to the one used for measurement study) and elaborate each component in next section. The system framework is illustrated in Figure 6.3.

1) Static analyzer. When a webpage passes through crawler, static analyzer parses it to a DOM tree (step ❶). For each Iframe tag identified, we create a entry and associate it with features derived from tag attributes, ancestor DOM nodes and the hosting page (step ❷).

2) Dynamic analyzer. An Iframe injected at the runtime by JavaScript code can be obfuscated to avoid its destination being easily exposed to security tools (e.g., URL blacklist) or human analysts. Static analyzer is ineffective under this case so we developed a dynamic analyzer to execute each extracted JavaScript code snippet with a full-fledged browser and log the execution traces (step ❸). While there have been a number of previous works applying dynamic analysis to detect malicious

JavaScript code (e.g, [78]), these approaches are designed to achieve very high code coverage, but suffering from significant overhead. Alternatively, we optimize the dynamic analyzer to focus on Iframe injection and use a lightweight DOM monitor to reduce the cost of instrumentation. When an Iframe tag is discovered in the runtime, the features affiliated with it and the original Iframe script are extracted and stored in the feature vector similar to static analyzer (step ❹).

3) Classification. Every Iframe tag and Iframe script is evaluated based on machine-learning models trained ahead (step ❺). We use separate classifiers for offline and online modes, as their training data are different and feature set are not entirely identical. The detection result is ordered by the prediction score and analysts can select the threshold to cap the number of Iframes to be inspected.

6.4 Large-scale Analysis of Iframe Inclusion

To get better understanding of how Iframe is included by websites around Internet, we crawled a large volume of websites and analyzed their code and runtime behaviors. Below we first describe our data collection methodologies. Then, we characterize the category, intention and sophistication of inclusion in both offline and online scenarios.

6.4.1 Data Collection

We choose the sites listed in Alexa top 1 million [1] as our study subject. For each site, our crawler visits the homepage and passes the collected data to a DOM parser and an instrumented browser for in-depth analysis. Some previous works use different strategies for web measurement, e.g., crawling 500 pages per site in Alexa top 10K list [94]. In this study, we are also interested in how less popular sites include Iframe, therefore we use the entire list of Alexa 1M.

We built the crawler on top of Scrapy [14], an application framework supporting customized crawling jobs. We leverage its asynchronous request feature and are able to finish the crawling task within 10 days. For each site visited, our crawler downloads the homepage and also saves the response header to measure the usage of browser policies. We use an open-source library *BeautifulSoup* to parse the DOM tree and find all Iframe tags to measure offline inclusion. For online inclusion, we let a browser load each page and capture the dynamically generated Iframe through `MutationObserver` (detailed in Section 6.5.2). This approach allows us to study Iframe gen-

[1] `http://s3.amazonaws.com/alexa-static/top-1m.csv.zip`

erated by *any* scripting code (JavaScript, Adobe Flash and etc.) within the entire page. In total, we obtained 860,873 distinct HTML pages (we name this set P) and their response headers. In total we spent 5 hours analyzing offline Iframes and 148 hours analyzing online Iframes. The ratio of websites returning valid to our crawler (86%) is slightly lower than a previous work also measuring Alexa top 1M sites (91%) [59]. We speculate the difference is caused by the crawler implementation: their crawler is built on top of an off-the-shelf browser and they set the timeout to 90 seconds.

To notice, we did not perform deep crawling (e.g., visiting pages other than the homepage, simulating user actions and attempting to log in) for each studied website. Our "breadth-first" crawling strategy aligns with previous works in large-scale web measurement [59]. Though deep crawling could discover more Iframe inclusions, it will take much more execution time.

6.4.2 Distribution of Iframes

Popularity of Iframe inclusion. Table 6.1 presents the statistics of Iframe inclusion in the surveyed pages. In short, the usage of Iframe is moderate in the contemporary Internet world: only 31.6% pages include Iframe [2], whereas more than 92% pages have script tags. To notice, our instrumented browser captures any type of online Iframe inclusion, including that caused by script tags, which means script inclusion is rarely intended to insert Iframe.

Specifically, we found 30.8% pages include Iframe in the offline fashion, whereas only 2% include

[2]Only Iframe with `src` is considered in the measurement study.

	Offline Iframe	Online Iframe	Script Tag
# pages	265,087	16,292	799,673
Percent	30.8%	2%	92.9%
# pages with external `src`	229,558	6,016	587,946
Percent	26.7%	0.7%	68.3%

Table 6.1: Statistics for the 860,873 webpages (P). We count the pages that embed Iframe and their ratio. We also count the pages embedding script tag.

Iframe during runtime. We speculate such prominent difference is resulted from the separation of interfaces from third-party services. As an example, Google provides two options for using its web-tracking services [7]: placing an Iframe tag or a script tag. When the developer chooses the latter, the tracking code directly collects the visitor's information and no Iframe is created.

Then, we investigate the pages with offline Iframes which we name as P_{Off}. As expected, most pages (86.6% over P_{Off}) use offline Iframe to load content from external source (external host-name). For the remaining pages, though most of destinations are relative paths, we still find a large number of paths like `file://*`, `javascript:false` and `about:blank`, which would not load HTML content. It turns out `file://*` is used to load file from user's local hard-disk and and the latter two is used as placeholder which is assigned by script after a moment [3]. Among

[3]The update of `src` might be triggered by user's action, like moving mouse. User actions are not simulated by our system so the update might be missed.

Domain	# Pages	Category	Percent
googletagmanager.com	102,207	web analytics	44.52%
youtube.com	52,308	social web	22.79%
facebook.com	37,466	social web	16.32%
google.com	7,564	search engines	3.29%
vimeo.com	6,943	streaming	3.02%
doubleclick.net	3,838	advertisement	1.67%

Table 6.2: Top 6 external domains referenced through offline inclusion. The percent is counted over P_{OffEx}.

the 16,292 pages including Iframe in online fashion (named P_{On}), only 6,016 (36.9%) point to external destination (named P_{OnEx}). Non-standard paths described above are extensively used in online Iframes.

Characterization of Iframe destinations. Table 6.2 and Table 6.3 list the domain name of most popular destinations in terms of referenced pages. For each domain, we obtained domain report from VirusTotal and used the result from Forcepoint ThreatSeeker to learn its category. In the offline case, services from the giant companies like Google and Facebook dominate the destinations. The number 1 domain is `googletagmanager.com`, a web analytics provided by Google. In the online case, while `doubleclick.net` takes 54.3% of P_{OnEx}, the percent for remaining domains are all small, 3.5% at most for `facebook.com`. Next, we compute the distribution of web categories regarding external destination. The result is shown in Figure 6.4. It turns out the

Domain	# Pages	Category	Percent
doubleclick.net	3,269	advertisement	54.34%
facebook.com	213	social web	3.54%
youtube.com	172	social web	2.85%
ad-back.com	170	advertisements	2.82%
prom.ua	153	business&economy	2.54%

Table 6.3: Top 5 external domains referenced through online inclusion. The percent is counted over P_{OnEx}.

distribution is quite different for offline and online case: social network and web analytics are the two main categories in offline case while advertisement Iframe is used more often in online case.

Comparison to script inclusion. Finally, we compare the Iframe inclusion and script inclusion in terms of amount and categories. Previous work has studied the offline script inclusion and showed that most script are attributed to web analytics, advertisement and social network, which is consistent with our findings here. Still, there are two prominent differences. 1) The number of script tags is an order of magnitude more than Iframe tags (8,802,569 vs 561,048 from the crawled 860,873 pages). 2) The Iframe usage per site is quite limited. As shown in Figure 6.5, more than 60% webpages with Iframe inclusion only embed 1 Iframe, whereas at least 2 script tags are seen for 90% webpages doing script inclusion.

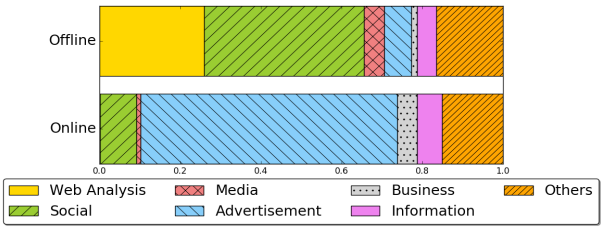
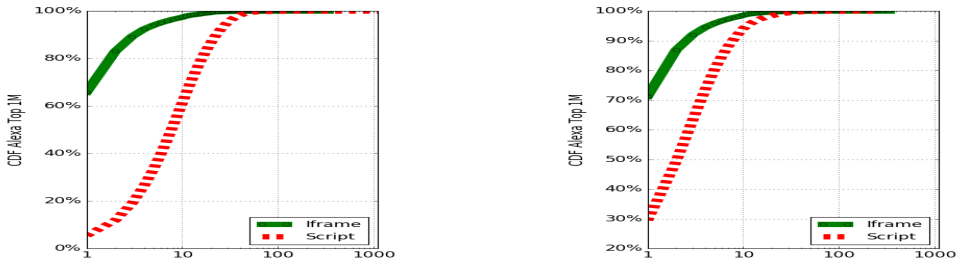


Figure 6.4: Categories of offline and online Iframe destinations.



(a) All.

(b) External.

Figure 6.5: ECDF of number of offline Iframe and script tag per site.

6.4.3 Usage of Browser Policies

Site developers can leverage policies defined by browser vendors to control the threat posed by Iframes from third-party or unknown destinations. We are eager to know how the policies are enforced. To this end, we measure the usage of CSP, `X-Frame-Options` and `sandbox`. The result is elaborated below.

Regarding the usage of CSP, we found only 18,329 websites (2.1%) specify CSP or X-CSP and less than half of them use Iframe-related fields (see Table 6.4). CSP has much better adoption than X-CSP and we speculate it is caused by the deprecation of X-CSP in the latest specification.

On the other hand, the adoption of `X-Frame-Options` is much better, as 107,553 websites

Site Count	CSP	X-CSP
frame-src	1,637	251
frame-ancestor	3,049	307
child-src	1,062	86

Table 6.4: Usage of CSP.

Site Count	`X-Frame-Options`	Ratio
SAMEORIGIN	95,897	89.2%
DENY	9,547	8.88%
ALLOW-FROM	1,200	1.12%

Table 6.5: `X-Frame-Options` Usage.

make use of this feature (see Table 6.5). The reason is perhaps that `X-Frame-Options` is more compatible with outdated browsers [18]. Yet, on closer look, this result turns out to be quite puzzling. First, only 1.12% sites use the `ALLOW-FROM` option, meaning that most of the site owners make no differentiation on Iframe destinations. Second, a non-negligible amount of sites make mistakes that disable this policy: 979 sites use conflict options at the same time (e.g., both `SAMEORIGIN` and `DENY` are specified) and 1,338 sites misspell the options (`DANY`:6, `SAMEORGIN`:12, `ALLOWALL`:1,320). In the end, 86% percent of websites do not use either CSP or `X-Frame-Options`, suggesting there is a long way ahead for their broad adoption.

It is a good practice to restrict the capabilities of Iframe through `sandbox` attribute, but we found it only appears in 0.37% Iframes (2,104 out of 561,048).

6.4.4 Iframe Features

In this subsection, we take a close look at how Iframe tag is designed and included. In particular, we measured the `style` attribute of Iframe tag and the statistics of Iframe script.

Iframe `style`. Though the original purpose of Iframe is to split webpage's visible area into different regions, we found many Iframes are designed to be invisible to visitors, through the use of `style` attribute. Specifically, we found 31.0% of the Iframes are hidden for offline inclusion, in terms of size (e.g., `width`:0 and `height`:0) and visual position. However, when excluding 3 popular domains (`googletagmanager.com`, `doublclick.net` and `yjtag.jp`), only 3.8% Iframes are hidden. Similarly for online inclusion, 64.1% Iframes are hidden, but 80.2% among them are belong to 2 domains (`doubleclick.net` and `ad-back.net`). The result shows web analytics and advertisements are the major supporters for hidden Iframe.

Iframe script. Through the use of script, the destination of Iframe can be obfuscated, which could hinder existing web scanners or human analysts. We are interested in whether obfuscation is heavily performed by legitimate scripts and the answer turns out to be negative. We search the destination of all dynamically generated external Iframe among the script from the 6,016 pages (see Table 6.1) and found the match of domain name in 5,326 pages (88%). This result suggests developer usually has no intention to hide the destination. More specifically, among the 3,269 pages embedding `doublelcick.net` Iframe (see Table 6.3), match is found in 3,237 pages ($>$ 99%). One may question why developers choose Iframe script when it is only used in a light way. It turns out the purposes are merely delaying the assignment of destination and attaching more information to URL (e.g., add random nonce to URL parameter). This pattern significantly differs from the malicious samples where destination-obfuscation is extensively performed, which motivates our design of selective multi-execution (Section 6.5.2).

6.4.5 Summary

In conclusion, 30.8% sites perform offline injection and 2% perform online injection. Iframe is mainly used for social-network content, web analytics and advertisements. Several giant companies have taken the lion's share in terms of Iframe destination. The adoption of browser policies related to Iframe is still far from the ideal state and even writing policy in the correct way is not always done by developers.

6.5 Design and Implementation

In this section, we elaborate the features for determining whether an Iframe is injected. To effectively and efficiently expose the Iframe injected in the runtime by JavaScript, we developed a lightweight dynamic analyzer and describe the implementation details.

6.5.1 Detecting Offline Iframe Injection

By examining online reports and a small set of samples about Iframe injection, we identified three categories of features related to the style, destination and context of malicious Iframe, which are persistently observed in different attack campaigns. Through the measurement study, we found these features are rarely presented in benign Iframes. All of the features can be directly extracted from HTML code and URL, therefore `FrameHanger` is able to classify Iframe immediately when a webpage is crawled. We describe the features by their category.

Style-based features. To avoid raising suspicion from the website visitors, the malicious Iframe is usually designed to be hidden, as shown in the example of Section 6.2. This can be achieved through a set of methods, including placing the Iframe outside of the visible area of browser, setting the Iframe size to be very small or preventing Iframe to be displayed. All of these visual effects can be determined by the `style` attribute of the Iframe tag. As such, `FrameHanger` parses this attribute and checks each individual value against a threshold (e.g., whether `height` is smaller than 5px) or matches it with a string label (e.g., whether `visibility` is `hidden`). As a countermeasure, attacker can create a parent node above the Iframe (e.g., `div` tag) and configure `style` there. Therefore, we also consider the parent node and the node above to extract the same set of features. We found style-based features could distinguish malicious and benign Iframe, as most of the benign Iframes are not hidden, except the ones belong to well-known third parties, like web analytics, as shown in Section 6.4.4. These benign but hidden Iframes can be easily recognized with the help of public list, like EasyList [4] and EasyPrivacy [5], and pre-filtered.

Destination-based features. We extract the lexical properties of the `src` attribute from the Iframe tag to model this category. `FrameHanger` only considers the properties from external destination, as this is the dominant way to redirect visitors to other malicious website, suggested by a large corpus of reports (e.g., [9]). For attacker, using relative path requires compromising another file or uploading a malicious webpage, which makes the hacking activity more observable. An Iframe without valid `src` value does not do any harm to visitors. The lexical properties `FrameHanger` uses is similar to what has been tested by previous works on URL classification [92], and we omit the details.

Context-based features. Based on our pilot study, we found attacker prefers to insert malicious Iframe code into abnormal position in HTML document, e.g., the beginning or end of the document. In contrast, site developers often avoid such positions. For this category, we consider the distance of the Iframe to positions of our interest and the distance is represented by number of lines or levels in DOM tree. Exploiting the code vulnerability of website, like SQL injection and XSS, is a common way to gain control illegally. Websites powered by web templates, like WordPress, Joomla and Drupal, are breached frequently because the related vulnerabilities are easy to find. We learn the template information from the META field of header and consider it as a feature. Finally, we compare the Iframe domain with other Iframe domains in the page and consider it more suspicious when it is different from most of others.

6.5.2 Detecting Online Iframe Injection

When the Iframe is injected by obfuscated JavaScript code, feature extraction is impossible by static analyzer. Deobfuscating JavaScript code is possible, but only works when the obfuscation algorithm can be reverse-engineered. Dynamic analysis is the common approach to counter obfuscation, but it can be evaded by environment profiling [80]: e.g, the malicious code could restrict its execution on certain browsers based on `useragent` strings. To fix this shortcoming, force execution [76, 78] and symbolic execution [80, 105] are integrated into dynamic analysis. While they ensure all paths are explored, the overhead is considerable, especially when the code contains many branches and dependencies on variables.

Based on our large-scale analysis of Iframe script in Alexa top 1M sites and samples of malicious Iframe scripts, we found the existing approaches can be optimized to address the overhead issue without sacrificing detection rate. For legitimate Iframe scripts, obfuscation and environment profiling is rarely used. For malicious Iframe scripts, though these techniques are extensively used, only popular browser configurations are attacked (e.g., Google Chrome and IE). Therefore, we can choose different execution model according to the complexity of code: when the script is obfuscated or wrapped with profiling code, `FrameHanger` sends the code to multi-execution engine using a set of popular browser configurations; otherwise, the script is executed within a single pass. When an Iframe injection is observed (i.e., new node added to DOM whose tag name is Iframe), the script (and the Iframe) will go through feature extraction and classification.

Pre-filtering. At first step, `FrameHanger` uses the DOM parser to extract all script tags within the page. In this work, we focus on the Iframe injected by *inline script* (we discuss this choice in Section 6.7). The script that has no code inside or non-empty `src` attribute is filtered out.

The next task is to determine whether the code is obfuscated or the running environment is profiled. It turns out though deobfuscation is difficult, learning whether the code is obfuscated is a much easier task. According to previous studies [77, 131], string functions (e.g., `fromCharCode` and `charCodeAt`), dynamic evaluation (e.g., `eval`) and special characters are heavily used by JavaScript obfuscators. We leverage these observations and build a set of *obfuscation indicators* to examine the code. Likewise, whether environment profiling is performed can be assessed through a set of *profiling indicators*, like the usage of certain DOM functions (e.g., `getLocation` and `getTime`) and access of certain DOM objects (e.g., `navigator.userAgent`). In particular,

a script is parsed into abstract syntax tree (AST) using Slimit [8] and we match each node with our indicators.

Code wrapping and execution monitoring. Since we aim to detect Iframe injection at the tag level, `FrameHanger` executes every script tag separately in a *script container*, which wraps the script in a dummy HTML file. When the attacker distributes the Iframe script into multiple isolated script tags placed in different sections, the script code might not be correctly executed. However, such strategy might break the execution in user's browser as well, when a legitimate script tag is executed in between. Throughout our empirical analysis, code splitting is never observed, which resonates with findings from previous works [87].

Each script container is loaded by a browser to detect the runtime creation of Iframe. We first experimented with an open-source dynamic analysis framework [106] and instrument all the JavaScript function calls. However, the runtime overhead is considerable and recovering tag attributes completely is hindered when attacker plays string operations extensively. Therefore, we moved away from this direction and explore ways to directly catch the injected Iframe object. Fortunately, we found the mainstream browsers provide an API object named `MutationObserver`[4] to allow the logging of all sorts of DOM changes. Specifically, the script container first creates a `MutationObserver` with a callback function and then invokes a function `observe(target,config)` to monitor the execution. For the parameter `config`, we set both options `childList` and `subtree` to `true` to catch the insertions and removals of the DOM children's and their descendants. When the callback function is invoked, we perform logging when the input belongs to

[4] `https://developer.mozilla.org/en-US/docs/Web/API/MutationObserver`

`mutation.addedNodes` and its name equals "`iframe`". All attributes are extracted from the newly created Iframe for the feature exaction later. Because only one script tag exists in the script container, we are able to link the Iframe creation with its source script with perfect accuracy. We found this approach is much easier to implement (only 32 lines of JavaScript code in script container) and highly efficient, since `MutationObserver` is supported natively and the monitoring surface is very pointed.

Execution environment. We choose full-fledged browser as the execution environment. To manage the sequence of loading script containers, we leverage an automatic testing tool named Selenium [15] to open the container one by one in a new browser window. All events of our interest are logged by the browser logging API, e.g., `console.log`. The logger saves the name of hosting page, index of script tag and Iframe tag string (if created) into a local storage. We also override several APIs known to enable logic bomb, e.g., `setTimeout` which delays code execution and `onmousemove` which only responds to mouse events, with a function simply invoking the input. When a container is loaded, we keep the window open for 15 seconds, which is enough for a script to finish execution most of time. Depending on the pre-filtering result, single-execution or multi-execution will be preformed against the container. For multi-execution, we use 4 browser configurations (IE, Chrome, FireFox, Internet Explorer and Safari) by maneuvering `useragent` [5].

Changing `useragent` instead of actual browser for multi-execution reduces the running overhead significantly. However, this is problematic when the attacker profile environment based on browser-specific APIs. As revealed by previous works [87], `parseInt` can be used to determine

[5] `useragent` strings extracted from `http://useragentstring.com/`.

whether the browser is IE 6 by passing a string beginning with "0" to it and checking whether the string is parsed with the octal radix. In this case, changing `useragent` is not effective. Hence, we command Selenium to switch browser when such APIs are observed.

Feature Extraction Similar to static analyzer, we consider the features related to style, destination and the context, but extract them from different places. The destination and style features come from the generated Iframe while the context features come from the script tag. The number and meanings of features are identical.

6.6 Evaluation

In this section, we present the evaluation on `FrameHanger`. We first examine the accuracy of static and dynamic analyzer on a labeled dataset and then perform an analysis on the importance of features. Next, we measure the performance overhead of `FrameHanger`. Finally, we show cases of destination-obfuscation discovered by our study.

Dataset. We obtained 17,273 webpages detected by a security company through either manual or automated analysis. To notice, the malicious Iframe tag or script was not labeled by the company. To fill this missing information, we first extracted all Iframe tags and scanned their destination with VirusTotal. An Iframe tag triggers at least 2 alarms is included in our evaluation dataset. Similarly, all Iframe scripts were executed and labeled based on the response from VirusTotal regarding the generated Iframe tags. To avoid the issue of data bias, we select only one Iframe or script per unique destination, which leaves us 1,962 and 2,247 malicious samples for evaluating static and

dynamic analyzers.

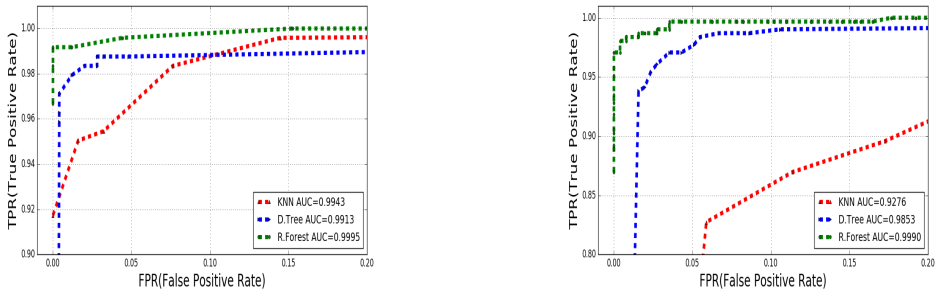
The benign samples come from the 229,558 tags and 6,016 scripts used for the measurement study. If we include all of them for training, the data will be very unbalanced. Therefore, we perform down-sampling to match the size of malicious set. Any sample triggering alarm from VirusTotal was removed.

6.6.1 Effectiveness

We tested 3 different machine-learning algorithms on the labeled dataset, including KNN (K-nearest neighbors), D.Tree (Decision Tree) and R. Forest (Random Forest). 10-fold cross validation (9 folds for training and 1 for testing) is performed for each algorithm, measured by AUC, accuracy and F1-score.

Figure 6.6a and Figure 6.6b present the ROC curve for static and dynamic analyzer. To save space, the result with over 0.2 False Positive Rate is not shown. R. Forest achieves the best result in terms of AUC. In addition, the accuracy and F1-score are (0.94, 0.93) for static analyzer, and (0.98,0.98) for dynamic analyzer. R.Forest has shown its advantage over other machine-learning algorithms in many security-related tasks and our result is consistent with this observation. We believe its capability of handling non-linear classification and over-fitting is key to its success here.

Feature analysis. We use feature importance score of R. Forest [17] to evaluate the importance of features. For the offline detection, the most important features belong to the context category, with 0.025 mean and 0.001 variance. For the online detection, the most important features belong



(a) ROC of static analyzer.

(b) ROC of dynamic analyzer.

Figure 6.6: ROC curve for the static and dynamic analyzer of `FrameHanger`.

to destination category, with 0.030 mean and 0.002 variance. It turns out the importance varies for the two analyzers, and most features have good contribution to our model.

6.6.2 Runtime Overhead

We ran `FrameHanger` on a server with 8 CPUs (Intel(R) Xeon(R) CPU 3.50GHz), 16GB memory and Ubuntu 14.04.5. The time elapse was measured per page. For static analyzer, it takes 0.10 seconds in average (0.018 seconds variance) from parsing to classification, suggesting `FrameHanger` is highly efficient in detecting offline injection. For dynamic analyzer, the mean overhead is 17.4 seconds but the variance is as high as 386.4 seconds. The number of script tags per page and the code sophistication per script (e.g., obfuscation) are the main factors accounting for the runtime overhead. In the mean time, we will keep optimizing the performance on dynamic analyzer.

6.6.3 Destination Obfuscation

Obfuscating destination is an effective evasion technique against static analysis. We are interested in how frequent this technique is used for real-world attacks. To this end, we compute the occurrences of obfuscation indicators in the 2,247 malicious Iframe scripts. The result suggests its usage is in deed prevalent: as an evidence, 1,247 has JavaScript function `fromCharCode` and 975 have `eval`.

Detecting destination obfuscation is very challenging for static-based approach, but can be adequately addressed by dynamic analyzer. Figure 6.7a and Figure 6.7b show examples of obfuscated Iframe scripts. We summarize two categories of obfuscation techniques from attack samples. 1) lightweight obfuscation and 2) heavyweight obfuscation. In the lightweight obfuscation, attackers only hide the destination with ASCII values. Figure 6.7a is an example of lightweight obfuscation. The static approach is able to find the "iframe" string but hard to infer the destination. More common cases are the heavyweight obfuscation, where both the "iframe" string and the destination are obfuscated. Figure 6.7bis the example of heavyweight obfuscation. Figure 6.7b applies `fromCharCode` to concatenate the destination string. Attackers hide the Iframe payload with ASCII values or string concatenation or even masquerade the string function with DOM property. However, these samples were picked up by our dynamic analyzer and the Iframes were exposed after runtime execution.

```
<script type="text/javascript">
var jojo = document.createElement('iframe'); jojo.setAttribute('width', '1');
jojo.setAttribute('height', '1'); jojo.setAttribute('style', 'display:none');
jojo.setAttribute('src',
'\x68\x74\x74\x70\x3A\x2F\x2F\x6D\x65\x66\x61\x2E\x77\x73\x2F\x32\x2F\
x6E\x65\x77\x73\x2E\x70\x68\x70\x3F\x73\x3D\x31\x63\x31\x38\x30\x34\x
36\x31\x36\x62');
document.body.appendChild(jojo);</script>
```

(a) Iframe destination in string of ASCII values.

```
<script type="text/javascript">
var
a="'1Aqapkrv'02v{rg'1F'00vgzv.....hctcqapkrv'00'1G'2C'2;tcp'02pgdgpgp'02'1F'02
glamfgWPKAmormlglv'0:fmawoglv,pgdgppgp'0;'1@'2C'2;tcp'02fgdcwnvjig{umpf'0
2'1F'02glamfgWPKAmormlglv'0:fm00pag'1F'00'02'02jmqv'1@'2C'2;fmawoglv,'mf
{,crrglfAjknf'0:kdpcog'0;'1@'2C'1A-qapkrv'1G';b='';c='';var clen;clen=a.length;
for(i=0;i<clen;i++){b+=String.fromCharCode(a.charCodeAt(i)^2)}
c=unescape(b);document.write(c);
</script>
```

(b) Iframe destination assembled using `fromCharCode`.

Figure 6.7: Obfuscations detected by the dynamic analyzer of `FrameHanger`.

6.6.4 Summary

We summarize our experimental findings. 1) `FrameHanger` achieves high precision of detection, i.e., 0.94 accuracy for offline Iframe detection and 0.98 for online Iframe detection. 2) `FrameHanger` encseounters moderate runtime overhead, 0.10 seconds for offline detection and 17.4 seconds for online detection. 3) `FrameHanger` successfully captures obfuscations in online Iframe injection, which is prevalent used by real-world attackers.

6.7 Discussion

We make two assumptions about adversaries, including their preferences on Iframe tag/script and self-contained payload. Invalidating these assumptions is possible, but comes with negative impact on attackers' themselves. Choosing other primitives like Adobe Flash subjects to content blocking by the latest browsers. Payload splitting could make the execution be disrupted by legitimate code running in between. On the other hand, `FrameHanger` can be enhanced to deal with these cases, by augmenting dynamic analyzer with the support of other primitives, and including other DOM

objects of the same webpage.

`FrameHanger` uses a popular HTML parser, BeautifulSoup, to extract Iframe tag and script. Attacker can exploit the discrepancy between BeautifulSoup and the native parsers from browsers to launch parser confusion attack [23]. As a countermeasure, multiple parsers could be applied when an parser error is found. Knowing the features used by `FrameHanger`, attackers can tweak the attributes or position of Iframe for evasion (e.g., making Iframe visible). While they may succeed in evading our system, the Iframe would be more noticeable to web users and site developers.

Our dynamic analyzer launches multi-execution selectively to address the adversarial evasion leveraging environment profiling. We incorporate a set of indicators to identify profiling behaviors, which might be insufficient when new profiling techniques are used by adversary. That said, updating indicators is a straightforward task and we plan to make them more comprehensive. Multiple popular browser profiles are used for multi-execution. Attackers can target less used browsers or versions for evasion. However, it also means the victim base will be drastically reduced. `FrameHanger` aims to strike a balance between coverage and performance for dynamic analysis. We will keep improving `FrameHanger` in the future.

6.8 Conclusion

In this chapter, we present a study about Iframe inclusion and a detection system against adversarial Iframe injection. By measuring its usage in Alexa top 1M sites, our study sheds new light into this "aged" web primitive, like developers' inclination to offline inclusion, the concentration of

Iframe destination in terms of categories, and the simplicity of Iframe script in general. Based on these new observations, we propose a hybrid approach to capture both online and offline Iframe injections performed by web hackers. The evaluation result shows our system is able to achieve high accuracy and coverage at the same time, especially when trained with R.Forest algorithm. In the future, we will continue to improve the effectiveness and performance of our system.

Chapter 7

Conclusion

In this thesis, I present my attempts for learning based security and post security rewriting: mobile security in detection of Android repackaged malware, Android application rewriting with sink prioritization, rewriting with natural language processing and web security in malicious Iframe detection.

In Chapter 3, I address the problem of detecting repackaged malware through code heterogeneity analysis. We demonstrate its application in classifying semantically disjoint code regions. We design a partition-based detection to discover regions in an app, and a machine-learning-based classification to recognize different internal behaviors in regions. Our detection leverages security heterogeneity in the code segments of repackaged malware. Our algorithm aims to capture the semantic and logical dependence in portions of a program. Our experimental results showed that our prototype is very effective in detecting repackaged malware and Android malware in general.

In Chapter 4, I 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 quantitatively ranking sensitive data flows and sinks of Android apps and 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.

In Chapter 5, I present *IronDroid*, a new app customization tool with a natural language interface. Our approach enables automatic app rewriting with security-oriented natural language analysis. We propose a new mining algorithm to extract user security preferences from natural languages. The rewriting specification is extracted via a user-intention-guided taint flow analysis. In the experiment, our approach achieves 90.2% in understanding user intentions, a significant improvement over the state-of-the-art solution. Our framework supports more rewriting operations than existing solutions. Our rewriting successfully rewrites all benchmark apps. Our approach introduces negligible 3.3% overhead in rewriting.

In Chapter 6, I propose a new detection system named `FrameHanger` to mitigate the threat from Iframe injection. The system is composed of a static analyzer against offline injection and a

dynamic analyzer against online injection. To counter the obfuscation and environment profiling heavily performed by malicious Iframe scripts, we propose a new technique called *selective multi-execution*. After an Iframe is extracted by `FrameHanger`, a machine-learning model is applied to classify its intention. We consider features regarding Iframe's style, destination and context. The evaluation result shows the combination of these features enables highly accurate detection: 0.94 accuracy is achieved by the static analyzer and 0.98 by the dynamic analyzer.

Bibliography

[1] Amazon Alexa. https://developer.amazon.com/alexa.

[2] Android dynamic permission. https://developer.android.com/training/permissions/requesting.html.

[3] Apple Siri. http://www.apple.com/ios/siri/.

[4] The easylist filter lists. https://easylist.to/. Accessed: 2017-10-10.

[5] The easyprivacy filter lists. https://easylist.to/easylist/easyprivacy.txt. Accessed: 2017-10-10.

[6] Framehanger released version. https://github.com/ririhedou/FrameHanger.

[7] Google tag manager quick start. https://developers.google.com/tag-manager/quickstart. Accessed: 2017-10-10.

[8] A javascript minifier written in python. https://github.com/rspivak/slimit. Accessed: 2017-10-10.

[9] Malvertising campaigns involving exploit kits. https://www.fireeye.com/blog/threat-research/2017/03/still_getting_served.html. Accessed: 2017-10-10.

[10] Obfuscation service. https://javascriptobfuscator.com/. Accessed: 2017-10-10.

[11] People don't trust mobile wallets and apps. https://www.computerworld.com/article/2880546/people-dont-trust-mobile-wallets-and-apps.html.

[12] Rsa shadow fall. https://www.rsa.com/en-us/blog/2017-06/shadowfall. Accessed: 2017-10-10.

[13] Same original policy. https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy. Accessed: 2017-10-10.

[14] Scrapy cralwer framework. https://scrapy.org/. Accessed: 2017-10-10.

[15] Selenium automates browsers. http://www.seleniumhq.org/.

[16] Smartphone owners worry about mobile app security. https://mashable.com/2012/09/05/smartphone-security/#fC3Z3vZ2vGq2.

[17] Tree-based importance score. http://scikit-learn.org/stable/auto_examples/ensemble/plot_forest_importances.html. Accessed: 2017-10-10.

[18] X-frame-options or csp frame-ancestors? https://oxdef.info/csp-frame-ancestors/. Accessed: 2017-10-10.

[19] Y. Aafer, W. Du, and H. Yin. DroidAPIMiner: Mining API-level features for robust malware detection in Android. In Proc. SecureComm, 2013.

[20] A. V. Aho, M. R. Garey, and J. D. Ullman. The transitive reduction of a directed graph. SIAM Journal on Computing, pages 131–137, 1972.

[21] A. Ali-Gombe, I. Ahmed, G. G. Richard III, and V. Roussev. AspectDroid: Android app analysis system. In Proc. of CODASPY, 2016.

[22] B. Andow, A. Nadkarni, B. Bassett, W. Enck, and T. Xie. A study of grayware on google play. In Proc. of MoST, 2016.

[23] G. Argyros, I. Stais, S. Jana, A. D. Keromytis, and A. Kiayias. Sfadiff: Automated evasion attacks and fingerprinting using black-box differential automata learning. In Proc. of CCS, 2016.

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

[25] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. 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.

[26] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In Proc. PLDI, 2014.

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

[28] N. Bach and S. Badaskar. A review of relation extraction. In Literature review for Language and Statistics II, 2007.

[29] A. Banerjee, B. Livshits, A. V. Nori, and S. K. Rajamani. Merlin: Specification inference for explicit information flow problems. In Proc. of PLDI, 2009.

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

[31] A. Blum, B. Wardman, T. Solorio, and G. Warner. Lexical feature based phishing url detection using online learning. In Proc. of AISec, 2010.

[32] K. Borgolte, C. Kruegel, and G. Vigna. Delta: automatic identification of unknown web-based infection campaigns. In Proc. of CCS, 2013.

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

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

[35] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. Crowdroid: behavior-based malware detection system for Android. In Proc. SPSM, 2011.

[36] S. Calzavara, A. Rabitti, and M. Bugliesi. Content security problems?: Evaluating the effectiveness of content security policy in the wild. In Proc. of CCS, 2016.

[37] D. Canali, M. Cova, G. Vigna, and C. Kruegel. Prophiler: A fast filter for the large-scale detection of malicious web pages. In Proc. of WWW, 2011.

[38] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen. Edgeminer: Automatically detecting implicit control flow transitions through the Android framework. In Proc. NDSS, 2015.

[39] O. Catakoglu, M. Balduzzi, and D. Balzarotti. Automatic extraction of indicators of compromise for web applications. In Proc. of WWW, 2016.

[40] D. M. Cer, M.-C. De Marneffe, D. Jurafsky, and C. D. Manning. Parsing to stanford dependencies: Trade-offs between speed and accuracy. In Proc. of LREC, 2010.

[41] K. Chen, P. Wang, Y. Lee, X. Wang, N. Zhang, H. Huang, W. Zou, and P. Liu. Finding unknown malice in 10 seconds: Mass vetting for new threats at the google-play scale. In Proc. USENIX Security, 2015.

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

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

[44] H. Choi, B. B. Zhu, and H. Lee. Detecting malicious web links and identifying their attack types. In Proc. of USENIX Conference on Web Application Development, 2011.

[45] M. Cova, C. Kruegel, and G. Vigna. Detection and analysis of drive-by-download attacks and malicious javascript code. In Proc. of WWW, 2010.

[46] J. Crussell, C. Gibler, and H. Chen. Attack of the clones: Detecting cloned applications on Android markets. In Proc. ESORICS, 2012.

[47] J. Crussell, C. Gibler, and H. Chen. Andarwin: Scalable detection of semantically similar Android applications. In Proc. ESORICS, 2013.

[48] C. Curtsinger, B. Livshits, B. G. Zorn, and C. Seifert. Zozzle: Fast and precise in-browser javascript malware detection. In Proc. of USENIX Security, 2011.

[49] B. Davis and H. Chen. RetroSkeleton: Retrofitting Android Apps. In Proc. of MobiSys, 2013.

[50] B. Davis and H. Chen. Retroskeleton: Retrofitting Android apps. In Proc. MobiSys, 2013.

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

[52] L. Deshotels, V. Notani, and A. Lakhotia. Droidlegacy: automated familial classification of Android malware. In Proc. PPREW, 2014.

[53] K. O. Elish, X. Shu, D. Yao, B. G. Ryder, and X. Jiang. Profiling user-trigger dependence for Android malware detection. Computers & Security, 2014.

[54] K. O. Elish, D. D. Yao, and B. G. Ryder. On the need of precise inter-app icc classification for detecting Android malware collusions. In Proc. of IEEE Mobile Security Technologies (MoST), 2015.

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

[56] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. ACM Transactions on Computer Systems (TOCS), 2014.

[57] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of android application security. In Proc. of USENIX Security, 2011.

[58] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In Proc. of CCS, 2009.

[59] S. Englehardt and A. Narayanan. Online tracking: A 1-million-site measurement and analysis. In Proc. of CCS, 2016.

[60] E. Fernandes, A. Aluri, A. Crowell, and A. Prakash. Decomposable trust for Android applications. In Proc. of DSN, 2015.

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

[62] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck. Structural detection of Android malware using embedded call graphs. In Proc. AISec, 2013.

[63] C. Gibler, J. Crussell, J. Erickson, and H. Chen. AndroidLeaks: Automatically detecting potential privacy leaks in Android applications on a large scale. In Proc. TRUST, 2012.

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

[65] S. Hanna, L. Huang, E. Wu, S. Li, C. Chen, and D. Song. Juxtapp: A scalable system for detecting code reuse among Android applications. In Proc. DIMVA, 2013.

[66] K. Hou, W. c. Feng, and S. Che. Auto-tuning strategies for parallelizing sparse matrix-vector (spmv) multiplication on multi- and many-core processors. In 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), May 2017.

[67] K. Hou, W. Liu, H. Wang, and W.-c. Feng. Fast Segmented Sort on GPUs. In Proceedings of the 2017 International Conference on Supercomputing, ICS '17. ACM, 2017.

[68] K. Hou, H. Wang, and W. c. Feng. Delivering Parallel Programmability to the Masses via the Intel MIC Ecosystem: A Case Study. In 2014 43rd International Conference on Parallel Processing Workshops, pages 273–282, Sept 2014.

[69] K. Hou, H. Wang, W. Feng, J. Vetter, and S. Lee. Highly Ef?cient Compensation-based Parallelism for Wavefront Loops on GPUs. In IEEE Int. Parallel and Distrib. Process. Symp. (IPDPS), 2018.

[70] K. Hou, H. Wang, and W.-c. Feng. ASPaS: A Framework for Automatic SIMDization of Parallel Sorting on x86-based Many-core Processors. In Proceedings of the 29th ACM on International Conference on Supercomputing, ICS '15, pages 383–392, New York, NY, USA, 2015. ACM.

[71] K. Hou, H. Wang, and W. C. Feng. AAlign: A SIMD Framework for Pairwise Sequence Alignment on x86-Based Multi-and Many-Core Processors. In 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pages 780–789, May 2016.

[72] K. Hou, H. Wang, and W.-c. Feng. Gpu-unicache: Automatic code generation of spatial blocking for stencils on gpus. In Proceedings of the ACM Conference on Computing Frontiers, CF '17. ACM, 2017.

[73] K. Hou, H. Wang, and W.-c. Feng. A Framework for the Automatic Vectorization of Parallel Sort on x86-based Processors. IEEE Trans. Parallel Distrib. Syst. (TPDS), 2018.

[74] W. Hu and D. Gu. AppSpear: Bytecode decrypting and DEX reassembling for packed android malware. In Proc. RAID, 2015.

[75] W. Hu, J. Tao, X. Ma, W. Zhou, S. Zhao, and T. Han. MIGDroid: Detecting app-repackaging Android malware via method invocation graph. In Proc. ICCCN, 2014.

[76] X. Hu, Y. Cheng, Y. Duan, A. Henderson, and H. Yin. Jsforce: A forced execution engine for malicious javascript detection. CoRR, abs/1701.07860, 2017.

[77] S. Kaplan, B. Livshits, B. Zorn, C. Seifert, and C. Curtsinger. ”nofus: Automatically detecting” + string.fromcharcode(32) + ”obfuscated ”.tolowercase() + ”javascript code”. Technical Report MSR-TR-2011-57, Microsoft Research, May 2011.

[78] K. Kim, I. L. Kim, C. H. Kim, Y. Kwon, Y. Zheng, X. Zhang, and D. Xu. J-Force: Forced execution on javascript. In Proc. of WWW, 2017.

[79] D. Klein and C. D. Manning. Accurate unlexicalized parsing. In Proc. of ACL, 2003.

[80] C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert. Rozzle: De-cloaking internet malware. In Proc. of Security and Privacy (Oakland), 2012.

[81] A. Kovacheva. Efficient code obfuscation for android. In Proc. International Conference on Advances in Information Technology, 2013.

[82] D. Kumar, Z. Ma, Z. Durumeric, A. Mirian, J. Mason, J. A. Halderman, and M. Bailey. Security challenges in an increasingly tangled web. In Proc. of WWW, 2017.

[83] P. Lam, E. Bodden, O. Lhoták, and L. Hendren. The soot framework for java program analysis: a retrospective. In Proc. of CETUS, 2011.

[84] T. Lauinger, A. Chaabane, S. Arshad, W. Robertson, C. Wilson, and E. Kirda. Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web. In Proceedings of NDSS, 2017.

[85] A. Le, A. Markopoulou, and M. Faloutsos. Phishdef: Url names say it all. In Proc. of INFOCOM, 2011.

[86] L. Li, T. F. Bissyandé, D. Octeau, and J. Klein. Droidra: Taming reflection to support whole-program analysis of android apps. In Proc. ISSTA, 2016.

[87] Z. Li, S. Alrwais, X. Wang, and E. Alowaisheq. Hunting the red fox online: Understanding and detection of mass redirect-script injections. In Proc. of Security and Privacy (Okaland), 2014.

[88] F. Liu, H. Cai, G. Wang, D. D. Yao, K. O. Elish, and B. G. Ryder. MR-Droid: A scalable and prioritized analysis of inter-app communication risks. In Proc. MoST, in conjunction with the IEEE Symposium on Security and Privacy, 2017.

[89] K. Lu, Z. Li, V. P. Kemerlis, Z. Wu, L. Lu, C. Zheng, Z. Qian, W. Lee, and G. Jiang. Checking more and alerting less: Detecting privacy leakages via enhanced data-flow analysis and peer voting. In Proc. NDSS, 2015.

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

[91] T. C. H. Lux, L. T. Watson, T. H. Chang, J. Bernard, B. Li, X. Yu, L. Xu, G. Back, A. R. Butt, K. W. Cameron, D. Yao, and Y. Hong. Novel meshes for multivariate interpolation and approximation. In Proceedings of the ACMSE 2018 Conference, ACMSE '18, pages 13:1–13:7, New York, NY, USA, 2018. ACM.

[92] J. Ma, L. K. Saul, S. Savage, and G. M. Voelker. Learning to detect malicious urls. ACM Transactions on Intelligent Systems and Technology (TIST), 2011.

[93] D. Nadeau and S. Sekine. A survey of named entity recognition and classification. Lingvisticae Investigationes, 2007.

[94] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. You are what you include: large-scale evaluation of remote javascript inclusions. In Proc. of CCS, 2012.

[95] M. Nourian, X. Wang, X. Yu, W.-c. Feng, and M. Becchi. Demystifying automata processing: Gpus, fpgas or micron's ap? In Proceedings of the International Conference on Supercomputing, ICS '17. ACM, 2017.

[96] D. Octeau, S. Jha, M. Dering, P. McDaniel, A. Bartel, L. Li, J. Klein, and Y. Le Traon. Combining static analysis with probabilistic models to enable market-scale Android inter-component analysis. In Proc. of POPL, 2016.

[97] D. Octeau, S. Jha, and P. McDaniel. Retargeting android applications to java bytecode. In Proc. of FSE, 2012.

[98] T. Pedersen, S. Patwardhan, and J. Michelizzi. Wordnet:: Similarity: measuring the relatedness of concepts. In Demonstration papers at HLT-NAACL, 2004.

[99] H. Peng, C. Gates, B. Sarma, N. Li, Y. Qi, R. Potharaju, C. Nita-Rotaru, and I. Molloy. Using probabilistic generative models for ranking risks of Android apps. In Proc. of CCS, 2012.

[100] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna. Execute this! analyzing unsafe and malicious dynamic code loading in android applications. In Proc. of NDSS, 2014.

[101] R. Potharaju, A. Newell, C. Nita-Rotaru, and X. Zhang. Plagiarizing smartphone applications: attack strategies and defense techniques. In Proc. ESSoS, 2012.

[102] N. Provos, M. Panayiotis, M. A. Rajab, and F. Monrose. All your iframes point to us. In Proc. of USENIX Security, 2008.

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

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

[105] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for javascript. In Proc. of Security and Privacy (Okaland), 2010.

[106] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for javascript. In Proc. of ESEC/FSE, 2013.

[107] Y. Shao, X. Luo, C. Qian, P. Zhu, and L. Zhang. Towards a scalable resource-driven approach for detecting repackaged Android applications. In Proc. ACSAC, 2014.

[108] R. Socher, A. Perelygin, J. Y. Wu, J. Chuang, C. D. Manning, A. Y. Ng, C. Potts, et al. Recursive deep models for semantic compositionality over a sentiment treebank. In Proc. of EMNLP, 2013.

[109] K. Soska and N. Christin. Automatically detecting vulnerable websites before they turn malicious. In Proc. of USENIX Security, 2014.

[110] W. Y. A. Z. Soteris Demetriou, Whitney Merrill and C. A. Gunter. Free for all! assessing user data exposure to advertising libraries on Android. In Proc. of NDSS, 2016.

[111] B. Stock, B. Livshits, and B. Zorn. Kizzle: A signature compiler for exploit kits. In International Conference on Dependable Systems and Networks (DSN), June 2016.

[112] M. Sun and G. Tan. NativeGuard: Protecting Android applications from third-party native libraries. In Proc. WiSec, 2014.

[113] X. Sun, Y. Zhongyang, Z. Xin, B. Mao, and L. Xie. Detecting code reuse in Android applications using component-based control flow graph. In Proc. IFIP SEC, 2014.

[114] J. B. T. H. C. B. L. X. Y. L. X. G. B. A. R. B. K. W. C. Y. H. D. Y. T. C. H. Lux, L. T. Watson. Nonparametric distribution models for predicting and managing computational performance variability. In IEEE SoutheastCon 2018, pages 1–7, April 2018.

[115] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro. CopperDroid: Automatic reconstruction of Android malware behaviors. In Proc. NDSS, 2015.

[116] K. Tian, G. Tan, D. D. Yao, and B. G. Ryder. Redroid: Prioritizing data flows and sinks for app security transformation. In Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation, pages 35–41. ACM, 2017.

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

[118] K. Tian, D. D. Yao, B. G. Ryder, G. Tan, and G. Peng. Detection of repackaged android malware with code-heterogeneity features. IEEE Transactions on Dependable and Secure Computing, 2017.

[119] K. Tian, L. Zhou, K. Bowers, and D. Yao. Framehanger: Evaluating and classifying iframe injection at large scale. In n the 14th EAI International Conference on Security and Privacy in Communication Networks (SecureComm), 2018.

[120] H. Wang, W. Liu, K. Hou, and W.-c. Feng. Parallel Transposition of Sparse Data Structures. In Proceedings of the 2016 International Conference on Supercomputing, ICS '16, pages 33:1–33:13, New York, NY, USA, 2016. ACM.

[121] L. Weichselbaum, M. Spagnuolo, S. Lekies, and A. Janc. Csp is dead, long live csp! on the insecurity of whitelists and the future of content security policy. In Proc. of CCS, 2016.

[122] M. Weissbacher, T. Lauinger, and W. Robertson. Why is csp failing? trends and challenges in csp adoption. In International Workshop on Recent Advances in Intrusion Detection, pages 212–233. Springer, 2014.

[123] B. Wolfe, K. Elish, and D. Yao. Comprehensive behavior profiling for proactive Android malware detection. In Proc. ISC, 2014.

[124] B. Wolfe, K. Elish, and D. Yao. High precision screening for Android malware with dimensionality reduction. In Proc. ICMLA, 2014.

[125] T. Wüchner, M. Ochoa, and A. Pretschner. Malware detection with quantitative data flow graphs. In Proc. of AsiaCCS, 2014.

[126] T. Wuchner, M. Ochoa, and A. Pretschner. Robust and effective malware detection through quantitative data flow graph metrics. In Proc. of DIMVA, 2015.

[127] t. y. X. Yu and K. Hou and H. Wang and W.-c. Feng , booktitle=2017 IEEE International Conference on Big Data (Big Data).

[128] K. Xu, K. Tian, D. Yao, and B. G. Ryder. A sharper sense of self: Probabilistic reasoning of program behaviors for anomaly detection with context sensitivity. In Proc. DSN, 2016.

[129] K. Xu, D. D. Yao, B. G. Ryder, and K. Tian. Probabilistic program modeling for high-precision anomaly classification. In Proc. of CSF, 2015.

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

[131] W. Xu, F. Zhang, and S. Zhu. Jstill: mostly static detection of obfuscated malicious javascript code. In Proc. of AsiaCCS, 2013.

[132] C. Yang, Z. Xu, G. Gu, V. Yegneswaran, and P. Porras. DroidMiner: Automated mining and characterization of fine-grained malicious behaviors in Android applications. In Proc. ESORICS, 2014.

[133] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck. AppContext: Differentiating malicious and benign mobile app behaviors using context. In Proc. ICSE, 2015.

[134] X. Yu. Deep packet inspection on large datasets: algorithmic and parallelization techniques for accelerating regular expression matching on many-core processors. University of Missouri-Columbia, 2013.

[135] X. Yu and M. Becchi. Exploring different automata representations for efficient regular expression matching on gpus. SIGPLAN Not., 2013.

[136] X. Yu and M. Becchi. Gpu acceleration of regular expression matching for large datasets: Exploring the implementation space. In Proceedings of the ACM International Conference on Computing Frontiers, CF '13. ACM, 2013.

[137] X. Yu, W.-c. Feng, D. D. Yao, and M. Becchi. O3fa: A scalable finite automata-based pattern-matching engine for out-of-order deep packet inspection. In Proceedings of the

2016 Symposium on Architectures for Networking and Communications Systems, ANCS '16. ACM, 2016.

[138] X. Yu, K. Hou, H. Wang, and W. c. Feng. A framework for fast and fair evaluation of automata processing hardware. In 2017 IEEE International Symposium on Workload Characterization (IISWC), pages 120–121, Oct 2017.

[139] X. Yu, B. Lin, and M. Becchi. Revisiting state blow-up: Automatically building augmented-fa while preserving functional equivalence. IEEE Journal on Selected Areas in Communications, 2014.

[140] X. Yu, H. Wang, W. C. Feng, H. Gong, and G. Cao. cuart: Fine-grained algebraic reconstruction technique for computed tomography images on gpus. In 2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), 2016.

[141] X. Yu, H. Wang, W.-c. Feng, H. Gong, and G. Cao. An enhanced image reconstruction tool for computed tomography on gpus. In Proceedings of the Computing Frontiers Conference, CF'17. ACM, 2017.

[142] X. Yu, H. Wang, W.-c. Feng, H. Gong, and G. Cao. Gpu-based iterative medical ct image reconstructions. Journal of Signal Processing Systems, Mar 2018.

[143] F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu. ViewDroid: Towards obfuscation-resilient mobile application repackaging detection. In Proc. WiSec, 2014.

[144] H. Zhang, D. Yao, and N. Ramakrishnan. Causality-based sensemaking of network traffic for Android application security. In Proceedings of the 9th ACM Workshop on Artificial Intelligence and Security (AISec'16), pages 47–58, 2016.

[145] H. Zhang, D. Yao, N. Ramakrishnanvt, and Z. Zhang. Causality reasoning about network events for detecting stealthy malware activities. Computers & Security, 2016.

[146] H. Zhang, D. D. Yao, and N. Ramakrishnan. Detection of stealthy malware activities with traffic causality and scalable triggering relation discovery. In Proceedings of the 9th ACM symposium on Information, computer and communications security, pages 39–50. ACM, 2014.

[147] M. Zhang, Y. Duan, H. Yin, and Z. Zhao. Semantics-aware Android malware classification using weighted contextual api dependency graphs. In Proc. of CCS, 2014.

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

[149] Y. Zhang, X. Luo, and H. Yin. DexHunter: toward extracting hidden code from packed android applications. In Proc. ESORICS, 2015.

[150] W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou. Fast, scalable detection of piggybacked mobile applications. In Proc. CODASPY, 2013.

[151] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. Detecting repackaged smartphone applications in third-party Android marketplaces. In Proc. CODASPY, 2012.

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