Mining Security Risks from Massive Datasets

Fang Liu

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 & Application

Danfeng Yao, Chair
Wenjing Lou
Ali R. Butt
B. Aditya Prakash
Dongyan Xu

June 28, 2017
Blacksburg, Virginia

Keywords: Cyber Security, Big Data Security, Mobile Security, Data Leakage Detection

Mining Security Risks from Massive Datasets

Fang Liu

(ABSTRACT)

Cyber security risk has been a problem ever since the appearance of telecommunication and electronic computers. In the recent 30 years, researchers have developed various tools to protect the confidentiality, integrity, and availability of data and programs.

However, new challenges are emerging as the amount of data grows rapidly in the big data era. On one hand, attacks are becoming stealthier by concealing their behaviors in massive datasets. One the other hand, it is becoming more and more difficult for existing tools to handle massive datasets with various data types.

This thesis presents the attempts to address the challenges and solve different security problems by mining security risks from massive datasets. The attempts are in three aspects: detecting security risks in the enterprise environment, prioritizing security risks of mobile apps and measuring the impact of security risks between websites and mobile apps. First, the thesis presents a framework to detect data leakage in very large content. The framework can be deployed on cloud for enterprise and preserve the privacy of sensitive data. Second, the thesis prioritizes the inter-app communication risks in large-scale Android apps by designing new distributed inter-app communication linking algorithm and performing nearest-neighbor risk analysis. Third, the thesis measures the impact of deep link hijacking risk, which is one type of inter-app communication risks, on 1 million websites and 160 thousand mobile apps. The measurement reveals the failure of Google's attempts to improve the security of deep links.

Mining Security Risks from Massive Datasets

Fang Liu

(GENERAL AUDIENCE ABSTRACT)

Cyber security risk has been a problem ever since the appearance of telecommunication and electronic computers. In the recent 30 years, researchers have developed various tools to prevent sensitive data from being accessed by unauthorized users, protect program and data from being changed by attackers, and make sure program and data to be available whenever needed.

However, new challenges are emerging as the amount of data grows rapidly in the big data era. On one hand, attacks are becoming stealthier by concealing their attack behaviors in massive datasets. On the other hand, it is becoming more and more difficult for existing tools to handle massive datasets with various data types.

This thesis presents the attempts to address the challenges and solve different security problems by mining security risks from massive datasets. The attempts are in three aspects: detecting security risks in the enterprise environment where massive datasets are involved, prioritizing security risks of mobile apps to make sure the high-risk apps being analyzed first and measuring the impact of security risks within the communication between websites and mobile apps. First, the thesis presents a framework to detect sensitive data leakage in enterprise environment from very large content. The framework can be deployed on cloud for enterprise and avoid the sensitive data being accessed by the semi-honest cloud at the same time. Second, the thesis prioritizes the inter-app communication risks in large-scale Android apps by designing new distributed inter-app communication linking algorithm and performing nearest-neighbor risk analysis. The algorithm runs on a cluster to speed up the computation. The analysis leverages each app's communication context with all the other apps to prioritize the inter-app communication risks. Third, the thesis measures the impact of mobile deep link hijacking risk on 1 million websites and 160 thousand mobile apps. Mobile deep link hijacking happens when a user clicks a link, which is supposed to be opened by one app but being hijacked by another malicious app. Mobile deep link hijacking is one type of inter-app communication risks between mobile browser and apps. The measurement reveals the failure of Google's attempts to improve the security of mobile deep links.

Acknowledgments

I would like to express my deepest gratitude to my mentor Dr. Danfeng (Daphne) Yao for providing me the opportunity to be part of the Yao group! 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 have learned so much over the past five years from Dr. Yaos knowledge and expertise, and the knowledge and skills gained in this learning experience build the most solid foundation for my dissertation work. I also greatly appreciate Dr. Yaos kindness and generosity. She is always available to provide insights on my research when I am lost and offer support when I am discouraged. 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. Wenjing Lou, Dr. Ali R. Butt, Dr. B. Aditya Prakash, Dr. Dongyan Xu, and Dr. Babara Ryder who generously contributed much time and energy to help improve my dissertation. I have benefited a lot from their diverse perspectives and insightful comments which help both deepen and broaden my views in my research. I also want to thank Dr. Gang Wang for providing guidance on the mobile deep link project. It was such a valuable and enjoyable learning experience despite it being short in duration.

I would like to thank my friends and peers who I worked with over the past five years: Dr. Kui Xu, Dr. Xiaokui Shu, Dr. Hao Zhang, Dr. Karim Elish, Dr. Haipeng Cai, Daniel Barton, Ke Tian, Dr. Long Cheng, Sazzadur Rahaman, Stefan Nagy, Alex Kedrowitsch, Andres Pico, Bo Li, Yue Cheng, Kaixi Hou, Zheng Song, Tong Zhang, Qingrui Liu and Xinwei Fu. Their companionship has brought so much joy into the challenging and stressful graduate school life!

Finally, I would like to thank my parents, my sister, and my wife for always having faith in me and loving me. Their consistent support instills in me the strongest motivation to keep learning more, working harder, and getting better.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Cyber security has been a study topic ever since the appearance of telecommunication and electronic computer. A vulnerability could lead to the compromise of confidentiality (e.g., data leak), integrity (e.g., authentication bypass) and availability (e.g., denial of service) of data or program. Due to the prevalence of vulnerabilities, security risks are everywhere. For example, ComDroid [53] reported that over 97% of the studied apps are vulnerable to communication hijacking attacks. One of the direct consequence of the security risks is data leakage. In the year of 2014 only, over 1 billion data records have been leaked as reported by SafeNet[32], a data protection company. The leaked records include email addresses, credit card numbers, etc. A survey conducted by Kaspersky Lab on 4438 companies indicates that about 83% of the companies experienced data leak incidents [84].

Although various tools have been developed in the recent 30 years to detect and fix the security risks, new challenges are emerging in the big data era. First, new attacks are becoming stealthier. They conceal their behaviors in large datasets. It is almost impossible to detect their malicious behaviors without correlating the data from multiple sources. Second, the size of datasets for mining security risks are growing rapidly. Existing approaches running on one single host are not able to process huge datasets. New detection algorithms running on distributed systems are needed for mining security risks. Third, the exposure of large datasets poses threats to the confidentiality of private information. The attackers are more powerful and are able to correlate multiple data sources and compromise the confidentiality of sensitive data. What is even worse is that enterprises are moving their computation to cloud, which requires extra privacy-preserving techniques.

This thesis presents the attempts to address the challenges by *detecting, prioritizing and measuring* security risks from massive datasets. Specifically, it focuses on the detection of data leakage in enterprise environments, the prioritization of inter-app communication risks and the measurement of deep link hijacking impact on mobile apps and the web.

Data Leakage The exposure of sensitive data is a serious threat to the confidentiality of

organizational and personal data. Reports showed that over 800 million sensitive records were exposed in 2013 through over 2,000 incidents [32]. Reasons include compromised systems, the loss of devices, or unencrypted data storage or network transmission. While many data leak incidents are due to malicious attacks, a significant portion of the incidents is caused by unintentional mistakes of employees or data owners. In this thesis, I focus on the detection of accidental/unintentional data leaks from very large datasets.

Inter-app Communication Risks Inter-Component Communication (ICC) is a key mechanism for app-to-app communication in Android, where components of different apps are linked via messaging objects (or *Intents*). While ICC contributes significantly to the development of collaborative applications, it also becomes a predominant security attack surface. In the context of *inter-app* communication scenarios, individual apps often suffer from risky vulnerabilities such as Intent hijacking and spoofing, resulting in leaking sensitive user data [53]. In addition, ICC allows two or more malicious apps to collude on stealthy attacks that none of them could accomplish alone [36, 100]. According to a recent report from McAfee Labs [11], app collusions are increasingly prevalent on mobile platforms. In this thesis, I focus on the prioritization of the large number of ICC risks and detect the high risk apps.

Deep Link Hijacking Risks With the wide adoption of smartphones, mobile websites and native apps have become the two primary interfaces to access online content [12, 115]. Users can easily launch apps from websites with preloaded *context*, which becomes instrumental to many key user experiences. The key enabler of web-to-mobile communication is mobile deep links. Like web URLs, mobile deep links are universal resource identifiers (URI) for content and functions within apps [131]. The most widely used deep link is *scheme URL* supported by both Android [7] and iOS [8] since 2008. Despite the convenience, researchers have identified serious security vulnerabilities in scheme URLs [42, 53, 138]. The most significant one is *link hijacking*, where one app can register another app's scheme and induce the mobile OS to open the wrong app. This allows the malicious apps to perform phishing attacks (*e.g.*, displaying a fake Facebook login box) or to steal sensitive data carried by the link (*e.g.*, PII) [53]. To address the problem, Google has designed App Link and Intent URLs . The thesis measures whether the new mechanisms are helpful in addressing the problems and what is the current impact of hijacking on mobile phones and on web.

In the rest of the chapter, I briefly introduce the problems and my contributions in detect data leakage, prioritize inter-app communication risks and measure security impact of deep link hijacking.

1.1 Detect Data Leakage in Large Datasets

Several approaches have been proposed to detect data exfiltration [24, 29, 30, 122] either on a host or the network. The technique proposed by Shu and Yao [122] performs deep packet inspection to search for exposed outbound traffic that bears high similarity to sensitive data.

Set intersection is used for the similarity measure. This similarity-based detection is versatile, capable of analyzing both text and some binary-encoded context (e.g., Word or `.pdf` files).

However, a naive implementation of the similarity-base approach requires $O(nm)$ complexity, where n and m are sizes of the two sets A and B, respectively. If A and B are both very large (as in my data-leak detection scenario), It would not be practical due to memory limitation and thrashing. Distributing the computation to multiple hosts is nontrivial to implement from scratch and has not been reported in the literature.

In this thesis, I present a new MapReduce-based system to detect the occurrences of plaintext sensitive data in storage and transmission. The detection is distributed and parallel, capable of screening massive amount of content for exposed information.

In addition, I preserve the confidentiality of sensitive data for outsourced detection. In my privacy-preserving data-leak detection, MapReduce nodes scan content in data storage or network transmission for leaks without learning what the sensitive data is.

I implement the algorithms using the open source Hadoop framework. The implementation has very efficient intermediate data representations, which significantly minimizes the disk and network I/O overhead. I achieved 225 Mbps throughput for the privacy-preserving data leak detection when processing 74 GB of content.

1.2 Prioritize Inter-communication Risks in Large-scale Android Apps

To assess the ICC vulnerabilities, various approaches have been proposed to analyze each individual app [19, 36, 53, 103, 136]. The advantage of analyzing each app separately is that it can achieve high scalability. However, it may produce overly conservative risk estimations [53, 60, 103]. Manually investigating all the alerts would be a daunting task for security analysts.

Recently, researchers start to co-analyze ICCs of two or more apps simultaneously. This allows researchers to gain empirical contexts on the actual communications between apps and produce more relevant alerts [22, 90, 118]. However, existing solutions are largely limited in scale due to the high complexity of pair-wise components analyses ($O(N^2)$). They were either applied to a much smaller set of apps or small sets of inter-app links.

In this thesis, I present MR-Droid, a MapReduce-based parallel analytics system for *accurate* and *scalable* ICC risk detection. I empirically evaluate ICC risks based on an app's inter-connections with other real-world apps, and detect high-risk pairs.

To construct the communication context of each app, I construct a large-scale *ICC graph*, where each node is an app component and the edge represents the corresponding *inter-app* ICCs. To gauge the risk level of the ICC pairs (edge weight), I extract various features based on app flow analyses that indicate vulnerabilities. With the ICC graph, I then *rank* the risk

level of a given app by aggregating all its ICC edges connected to other apps.

The system is implemented with a set of new MapReduce [57] algorithms atop the Hadoop framework. The high-level parallelization from MapReduce allows us to analyze millions of app pairs within hours using commodity servers. My evaluation of 11,996 apps demonstrates the scalability of MR-Droid. The entire process took less than 25 hours for an average of only 0.0012 seconds per app pair. My prioritization result achieves over 90% of true positive in detecting high risk apps.

1.3 Measure Hijacking Risks of Mobile Deep Links

Scheme URLs are known to be vulnerable to various attacks [42, 53, 138]. The most significant one is *link hijacking* Recently, two new deep link mechanisms were proposed to address the security risks in scheme URLs: App link and Intent URL. 1) *App Link* [5, 10] was introduced to Android and iOS in 2015. It no longer allows developers to customize schemes, but exclusively uses HTTP/HTTPS scheme. To prevent hijacking, App links introduce a way to verify the app-to-link association. 2) *Intent URL* [6] is another solution introduced in 2013, which only works on Android. Intent URL defines how deep links should be called by websites. Web developer can specify the app's package name to avoid being hijacked. While the new mechanisms are secure in theory, little is known about how effective they are in practice.

In this thesis, I conduct the first empirical measurement on various mobile deep links across apps and websites. My analysis is based on the deep links extracted from two snapshots of 160,000+ top Android apps from Google Play (2014 and 2016), and 1 million webpages from Alexa top domains. I find that the new linking methods (particularly App links) not only failed to deliver the security benefits as designed, but significantly worsen the situation. First, App links apply link verification to prevent hijacking. However, only 194 apps (2.2% out of 8,878 apps with App links) can pass the verification due to incorrect (or no) implementations. Second, I identify a new vulnerability in App link's preference setting, which allows a malicious app to intercept arbitrary HTTPS URLs in the browser without raising any alerts. Third, I identify more hijacking cases on App links than existing scheme URLs among both apps and websites. Many of them are targeting popular sites such as online social networks. Finally, Intent URLs have little impact in mitigating hijacking risks due to a low adoption rate on the web.

1.4 Document Organization

In the rest of the thesis, in Chapter 2, I will discuss the literature and related works. Chapter 3 presents my approach to detect data leakage in enterprise environment. The prioritization

of inter-app communication risks are presented in Chapter 4. I further extend the thesis with the measurement of mobile deep links in Chapter 5. Chapter 6 concludes the thesis and discusses the future directions.

Chapter 2

Review of Literature

This chapter review the literature related to the thesis. In Section 2.1, I discuss the related works to MapReduce, detecting data leakages and the related techniques. Section 2.2 reviews previous work on inter-app communication risks and detection& prevention techniques.

2.1 Data Leakage Detection

Existing data leakage detection/prevention techniques Existing commercial products on data leak detection/prevention include GoCloudDLP [68], Symantec DLP [130], IdentityFinder [78] and GlobalVelocity [66]. However, most of them cannot be outsourced because they do not have the privacy-preserving feature. GoCloudDLP [68] allows outsourced detection only on fully honest DLD provider.

Borders *et al.* [29] presented a network-analysis approach for estimating information leak in the outbound traffic. The method identifies anomalous and drastic increase in the amount of information carried by the traffic. The method was not designed to detect small-scale data leak. In comparison, my technique is based on intersection-based pattern matching analysis. Thus, my method is more sensitive to small and stealthy leaks. In addition, my analysis can also be applied to content in data storage (e.g., data center), besides network traffic. Croft *et al.* [54] compared two logical copies of network traffic to control the movement of sensitive data. The work by Papadimitriou *et al.* [106] aims at finding the agents that leaked the sensitive data. Shu *et al.* [123] presented privacy-preserving methods for protecting sensitive data in a non-MapReduce based detection environment. They further proposed to accelerate screening transformed data leaks using GPU [124]. Blanton *et al.*[26] proposed a solution for fast outsourcing of sequence edit distance and secure path computation, while preserving the confidentiality of the sequence.

MapReduce for similarity computation MapReduce framework was used to solve problems in various areas such as data mining [71], data management [87] and security [41, 109, 150]. The security solution proposed by Caruana *et al.* [41] is designing parallel SVM algorithms for filtering spam. Zhang *et al.* [150] proposed to use MapReduce to perform data anonymization. Provos *et al.* [109] leveraged MapReduce to parse URLs for web-based malware. My data leak detection problem is new, which none of the existing MapReduce solutions addresses. In addition, my detection goal differs from the aforementioned solutions.

MapReduce algorithms for computing document similarity (e.g., [23, 61]) and similarity joins (e.g., [14, 55, 58, 134]) involve pairwise similarity comparison. Similarity measurements may be edit distance, cosine distance, etc. Their approaches are not efficient for processing very large content when performing data leak detection. Most importantly, they don't preserve the confidentiality of sensitive data. There exist MapReduce algorithms for computing the set intersection [25, 134]. They differ from my collection intersection algorithms, which are explained in Section 4.1.4. My collection intersection algorithm requires new intermediate data fields and processing for counting and recording duplicates in the intersection.

Privacy-preserving techniques Private string search over encrypted data (e.g., [88, 97, 126]) and privacy preserving record linkage [107, 121] are also different from my problem. For searching over encrypted data, the scale of the search queries is relatively small when compared with the content in data leak detection. Most of the privacy preserving record linkage approaches assume the party who computes the similarity is trustworthy, which is different from my threat model. Bloom filters used in record linkage approaches [107, 121] are also not scalable for very large scale content screening, demonstrated in Section 3.5.1. Most importantly, traditional string matching or key word search only applies when the size of sensitive data is small or the patterns of all sensitive data are enumerable. It is not practical for the query string to cover all sensitive data segments for data leak detection.

The problem of DNA mapping [43, 139] techniques for data leak detection is similar to that of string search. For example, Chen *et al.* [43] proposed to perform DNA mapping with the MapReduce cluster. To preserve the privacy of DNA sequences, they used SHA-1 to hash both the query sequences and the reference genome. MapReduce is used to sort hashed reference genomes and match the hashed results. The alignment (edit distance) is calculated locally afterwards. However, the edit distance and hash function in their approach incur high computation overhead especially on the local cluster, which makes it not suitable for large-scale content screening that comes with large volume of queries. My approach is specialized for data leak detection, making it different in several aspects:

- I use collection intersection instead of edit distance to measure the similarity. Collection intersection is more efficient and suitable for large-scale data leak detection.
- I perform similarity calculation on the MapReduce cluster instead of the local nodes, which is preferred because data owner's computation capacity is limited.

- I use rolling hash (Rabin fingerprint) instead of the traditional hash function (SHA-1). Rabin fingerprint helps reduce data owner's overhead especially when hashing large-scale n-grams.

Shingle and Rabin fingerprint [110] was previously used for collaborative spam filtering [89], worm containment [38], etc. There are also general purpose privacy-preserving techniques such as secure multi-party computation (SMC) [142]. SMC is a cryptographic mechanism that supports various functions such as private join operations [40], private information retrieval [143] and genomic computation [81]. However, the provable privacy guarantees provided by SMC also comes with high computational complexity in terms of large content screening.

Improvements over MapReduce Several techniques have been proposed to improve the integrity or privacy protection of MapReduce framework. Such solutions typically assume that the cloud provider is trustworthy. Roy *et al.* [116] integrated mandatory access control with differential privacy in order to manage the use of sensitive data in MapReduce computations. Yoon and Squicciarini [144] detected malicious or cheating MapReduce nodes by correlating different nodes' system and Hadoop logs. Squicciarini *et al.* [127] presented techniques that prevent information leakage from the indexes of data in the cloud. In comparison, my work has a different security model. I assume that MapReduce algorithms are developed by trustworthy entities, yet the MapReduce provider may attempt to gain knowledge of the sensitive information.

Blass *et al.* [27] proposed a protocol for counting the frequency of Boolean formula expressed patterns with MapReduce. The protocol helps preserve the privacy of outsourced data records on the cloud. The privacy-preserving user matching protocol presented by Almishari *et al.* [17] provides a method for two parties to compare their review datasets. Asghar *et al.* [20] proposed to enforce role-based access control policies to preserve the privacy of data in outsourced environments. Hybrid cloud (e.g., [145]) is another general approach that secures computation with mixed sensitive data. Different from SMC, the MapReduce system in hybrid cloud is installed across public cloud and private cloud. It may require much more computation capacity for data owner when compared with my specific architecture. As enterprises are moving their services to the cloud, advanced techniques were also proposed to enhance the quality of service experience of data-intensive applications [45, 46, 47, 49] and reduce the cost for both cloud service providers and tenants [48, 50]. Some other approaches were also proposed to exploit the parallelism within computing nodes by using compiler-assisted methods [72], utilizing underlying resources efficiently [73, 75] and leveraging GPUs [74, 76, 77].

2.2 Mobile Inter-app Communication

Inter-app Attacks Researchers have discovered various vulnerabilities in the inter-app communication mechanism in Android [53] and iOS [135], which leads to potential hijacking and spoofing attacks. Chin *et al.* [53] analyzed the inter-app vulnerabilities in Android. They pointed out that the message passing system involves various inter-app attacks including broadcast theft, activity hijacking, etc. Davi *et al.* [56] conducted a permission escalation attack by invoking higher-privileged apps, which do not sufficiently protect their interfaces, from a non-privileged app. Ren *et al.* [113] demonstrated intent hijacking can lead to UI spoofing, denial-of-service and user monitoring attacks. Soundcomber [120] is a malicious app that transmits sensitive data to the Internet through an overt/covert channel to a second app. Android browsers also become the target of inter-app attacks via intent scheme URLs [132] and Cross-Application Scripting [69]. Inter-app attacks have become a serious security problem on smartphones. The fundamental issue is a lack of source and destination authentication [135]. Mobile deep links (*e.g.*, scheme URL) inherent some of these vulnerabilities when facilitating communications between websites and apps. My work on measuring deep links is complementary to existing work since I focus on large-scale empirical measurements, providing new understandings to how existing risks are mitigated in practice.

Mobile Browser Security. In web-to-app communications, mobile browsers play an important role in bridging websites and apps, which can also be the target of attacks. For example, malicious websites may attack the browser using XSS [69, 132] and origin-crossing [135]. The threat also applies to customized in-app browsers (called WebView) [51, 98, 102, 133]. In my work, I focus hijacking threats to apps, a different threat model where browser is the not target.

Defense and Detecting Techniques A significant amount of solutions have been proposed to perform single-app analysis and detect ICC vulnerabilities. DroidSafe [67] and AppIntent [141] analyzes sensitive data flow within an app and detects whether the sensitive information that flows out of the phone is user intended or not. ComDroid [52] and Epicc [103] identify Intents specifications on Android apps and detect ICC vulnerabilities. CHEX [96] discovers entry points and detects hijacking-enabling data flows in the app for component hijacking (Intent spoofing). Mutchler *et al.* [102] studied the security of mobile web apps. In particular, they detected inter-app URLs leakage in the mobile web apps. Zhang and Yin [149] proposed to automatically patch component hijacking vulnerabilities. Mulliner *et al.* [101] presented a scalable approach to apply third-party security patches against privilege escalation, capability leaks etc. PermissionFlow [118] detects inter-application permission leaks using taint analysis. On-device policies [37, 83, 111] were also designed to detect inter-app vulnerabilities and collusion attacks.

Inter-app analysis has also been proposed in the literature. One may combine multiple apps into one and perform single-app analysis on the combined one [90, 91]. These approaches

include flow-analysis and gain more detailed flow information from it. However, they do not scale and would be extremely expensive if applied to a large set of apps. In addition, in straightforward implementations, expensive program analysis is performed repetitively, which is redundant. In comparison, I perform data-flow analysis of an app *only once*, independent of how many its neighbors are.

COVERT [22] performs static analysis on multiple apps for detecting permission leakage. It extracts a model from each app. A formal analyzer is used to verify the safety of a given combination of apps. FUSE [112] statistically analyzes the binary of each individual app and merges the results to generate a multi-app information flow graph. The graph can be used for security analysis. Klieber *et al.* [85] performed static taint analysis on a set of apps with FlowDroid [19] and Epicc [103]. Jing *et al.* [82] proposed intent space analysis and a policy checking framework to identify the links among a small scale apps. The existing approaches mostly focus on the precision and analysis of in-depth flow information between apps. However, they are not designed to analyze large-scale apps. It is unclear how well they scale with real world apps. In addition, some solutions (such as Epicc and IC3) are for identifying ICCs, but do not have in-depth security classification as my work.

PRIMO [105] reported ICC analysis of a large pool of apps. Its analysis is on the likelihood of communications between two apps, not specifically on security. It does not provide any security classification. Nevertheless, PRIMO could potentially be used by us as a preprocessing step.

For related dynamic analysis, IntentFuzzer [140] detects capability leaks by dynamically sending Intents to the exposed interfaces. INTENTDROID [70] performs dynamic testing on Android apps and detects the vulnerabilities caused by unsafe handling of incoming ICC message. SCanDroid [65] checks whether the data flows through the apps are consistent with the security specifications from manifests. TaintDroid [62] tracks information flows with dynamic analysis and performs real-time privacy monitoring on Android. Similarly, FindDroid [151] associates each permission request with its application context thus protecting sensitive resources from unauthorized use. XManDroid [35] was proposed to prevent privilege escalation attacks and collusion attacks by analyzing the communications among apps and ensuring that the communications comply to a desired system policy. Network-based approach was also proposed for Android malware [146, 147, 148]. These dynamic analyses complement my static-analysis solution. However, their feasibility under a large number of app pairs is limited.

Generic Flow Analysis FlowDroid [19] is a popular static taint analysis tool for Android apps. The user can define the sources and sinks within one app and discover whether there are connections from the sources to the sinks. Amandroid [136] tracks the inter-component data and control flow of an app. The analysis can be leveraged to perform various types of security analysis. IC3 [104] focuses on inter-component communications and inferring ICC specifications. My work leverages IC3 [104] to extract the specifications of Intents. Although it is the most time-consuming step in my prototype, it is much more memory and

computation efficient than other static analysis tools. In addition, my approach scales well and has the potentials for market-scale security analysis.

Chapter 3

Detect Data Leakage

In this chapter, I propose to leverage MapReduce to perform data-leak detection [44, 93, 125]. MapReduce [57] is a programming model for distributed data-intensive applications. I present a new MapReduce-based system to detect the occurrences of plaintext sensitive data in storage and transmission. The detection is distributed and parallel, capable of screening the massive amount of content for exposed information.

Several existing approaches have been proposed to detect data exfiltration. The work by Papadimitriou and Garcia-Molina [106] aims to find the agents that leaked the sensitive data from unauthorized storage. The analysis proposed by Boarders *et al.* [29] detects changes in network traffic patterns by searching for the unjustifiable increase in HTTP traffic flow volume, that indicates data exfiltration. The technique proposed by Shu *et al.* [123] performs deep packet inspection to search for exposed outbound traffic that bears high similarity to sensitive data. Set intersection is used for the similarity measure. The intersection is computed between the set of n-grams from the content and the set of n-grams from the sensitive data. Other detection methods include enforcing strict data-access policies on a host (e.g., storage capsule [30]), watermarking sensitive data sets and tracking data flow anomalies (e.g., DBMS-layer [24]).

This similarity-based detection is versatile, capable of analyzing both text and some binary-encoded context (e.g., Word or *.pdf* files). A naive implementation requires $O(nm)$ complexity, where n and m are sizes of the two sets A and B, respectively. If the sets are relatively small, then a faster implementation is to use a hashtable or Bloom filter [28] to store set A and then testing whether items in B exist in the hashtable or not, giving $O(n+m)$ complexity. However, if A and B are both very large (as in my data-leak detection scenario), a naive hashtable may have hash collisions and slow down the computation. Although Bloom filter is more efficient in hash area and access time when compared with the naive hashtable, it is still not capable of processing very large data sets. Increasing the size of the hashtable may not

be practical due to memory limitation and thrashing.[1] One may attempt to distribute the dataset into multiple hashtables across several machines and coordinate the nodes to compute set intersections for leak scanning. However, such a system is nontrivial to implement from scratch and has not been reported in the literature.

I design and implement new MapReduce algoirithm computing the similarity between sensitive data and large public content. The advantage of my data leak solution is its scalability. Because of the intrinsic $\langle key, value \rangle$ organization of items in MapReduce, the worst-case complexity of my algorithms is correlated with the size of the leak (specifically a $\gamma \in [0, 1]$ factor denoting the size of the intersection between the content set and the sensitive data set). This complexity reduction brought by the γ factor is significant because the value is extremely low for normal content without leak. In my algorithm, items not in the intersection (non-sensitive content) are quickly dropped without further processing. Therefore, the MapReduce-based algorithms have a lower computational complexity when compared to the traditional set-intersection implementation. I experimentally verified the runtime deduction brought by γ in Section 3.5.4.

MapReduce can be deployed on nodes in the cloud (e.g., Amazon EC2) or in local computer clusters. It has been used to solve security problems such as spam filtering [41, 150], and malware detection [109]. However, the privacy of sensitive data is becoming an important issue that applications need to address when using MapReduce, especially when the sensitive data is outsourced to a third party or an untrusted cluster for analysis. The reason is that the MapReduce nodes may be compromised or owned by semi-honest adversaries, who may attempt to gain knowledge of the sensitive data. For example, Ristenpart *et al.* [114] demonstrated the possibility of exploring information leakage across VMs through side-channel attacks in third-party compute clouds (e.g., Amazon EC2).

Although private multi-party data matching and data retrieval methods [39, 43, 64] or more general purpose secure multi-party computation (SMC) framework exists [142], the high computational overhead is a concern for time-sensitive security applications such as data leak detection. For example, Chen *et al.* [43] proposed to perform DNA mapping using MapReduce while preserving the privacy of query sequences. However, the edit distance and hash function in their approach incur high computational overhead especially on the local cluster. My collection intersection based detection requires new MapReduce algorithms and privacy design, which their edit distance based approach does not apply.

I address the important data privacy requirement. In my privacy-preserving data-leak detection, MapReduce nodes scan content in data storage or transmission for leaks without learning what the sensitive data is.

Specifically, the data privacy protection is realized with a fast one-way transformation. This transformation requires the pre- and post-processing by the data owner for hiding and precisely identifying the matched items, respectively. Both the sensitive data and the content

[1] I experimentally validated this on a single host. The results are shown in Table 3.3 in Section 3.5.1.

need to be transformed and protected by the data owner, before it is given to the MapReduce nodes for the detection. In the meantime, such a transformation has to support the equality comparison required by the set intersection.

To summarize, my contributions in detecting data leakage are as follows.

- I present new MapReduce parallel algorithms for distributedly computing the sensitivity of content based on its similarity with sensitive data patterns. The similarity is based on collection intersection (a variant of set intersection that also counts duplicates). The MapReduce-based collection intersection algorithms are useful beyond the specific data leak detection problem.
- My detection provides the privacy enhancement to preserve the confidentiality of sensitive data during the outsourced detection. Because of this privacy enhancement, my MapReduce algorithms can be deployed in distributed environments where the operating nodes are owned by semi-honest service providers or untrusted clusters. Applications of my work include data leak detection in the cloud and outsourced data leak detection.
- I implement my algorithms using the open-source Hadoop framework. My prototype outputs the degree of sensitivity of the content, and pinpoints the occurrences of potential leaks in the content. I validate that my method achieves high detection rate and very low false positive rate for both plain text leaks and binary leaks.
- The pre-processing process can be handled without significant overhead. My implementation also has very efficient intermediate data representations, which significantly minimizes the disk and network I/O overhead. I performed two sets of experimental evaluations, one on Amazon EC2, and one on a local computer cluster, using large-scale email data. I achieved 225 Mbps throughput for the privacy-preserving data leak detection when processing 74 GB of content[2].

The significance of the work is the demonstration of the possibility of outsourced data leak detection. My architecture provides a complete solution of large-scale content screening for sensitive data leaks. The privacy guarantees, detection performance and the detection accuracy of my solution ensure its robustness for data leak detection in the enterprise environment.

3.1 Threat Model, Security and Computation goals

There are two types of input sequences in my data-leak detection model: content sequences and sensitive data sequences.

[2]74 GB content will be transformed into 74*8 GB intermediate data as input for my MapReduce system.

- *Content* is the data to be inspected for any occurrences of sensitive data patterns. The content can be extracted from file system or network traffic. The detection needs to partition the original content stream into **content segments**.

- *Sensitive data* contains the sensitive information that cannot be exposed to unauthorized parties, e.g., customers' records, proprietary documents. Sensitive data can also be partitioned to smaller **sensitive data sequences**.

3.1.1 Threat Model and Security Goal

In my model, two parties participate in the large-scale data leak detection system: data owner and data-leak detection (DLD) provider.

- *Data owner* owns the sensitive data and wants to know whether the sensitive data is leaked. It has the full access to both the content and the sensitive data. However, it only has limited computation and storage capability and needs to authorize the DLD provider to help inspect the content for the inadvertent data leak.

- *DLD provider* provides detection service and has much larger computation and storage power when compared with the data owner. However, the DLD provider is honest-but-curious (aka semi-honest). That is, it follows the prescribed protocol but may attempt to gain knowledge of sensitive data. The DLD provider is not given the access to the plaintext content. It can perform the dictionary attack on the digests of sensitive data records. Typical DLD providers include the untrusted local cluster, the public cloud (e.g., Amazon EC2) and the organizations that provide detection services.

My goal is to offer DLD provider solutions to scan massive content for sensitive data exposure and minimize the possibility that the DLD provider learns about the sensitive information.

- **Scalability**: the ability to process content at a variety of scales, e.g., megabytes to terabytes, enabling the DLD provider to offer on-demand content inspection.

- **Privacy**: the ability to keep the sensitive data confidential, not disclosed to the DLD provider or any attacker breaking into the detection system.

- **Accuracy**: the ability to identify all leaks and only real leaks in the content, which implies low false negative/positive rates for the detection.

My framework is not designed to detect intentional data exfiltration, during which the attacker may encrypt or transform the sensitive data.

3.1.2 Computation Goal

My detection is based on computing the similarity between content segments and sensitive data sequences, specifically the intersection of two collections of n-grams. One collection consists of n-grams obtained from the content segment and the other collection consists of n-grams from the sensitive sequence. n-grams captures local features of a sequence and have also been used in other sequence similarity methods (e.g., web search duplication [34]). Following the terminology proposed by Broader [34], n-gram is also referred to as *shingle*.

Collection intersection differs from set intersection, in that it also records duplicated items in the intersection, which is illustrated in Fig. 3.1. Recording the frequencies of intersected items achieves more fine-grained detection. Thus, collection intersection is preferred for data leak analysis than set intersection.

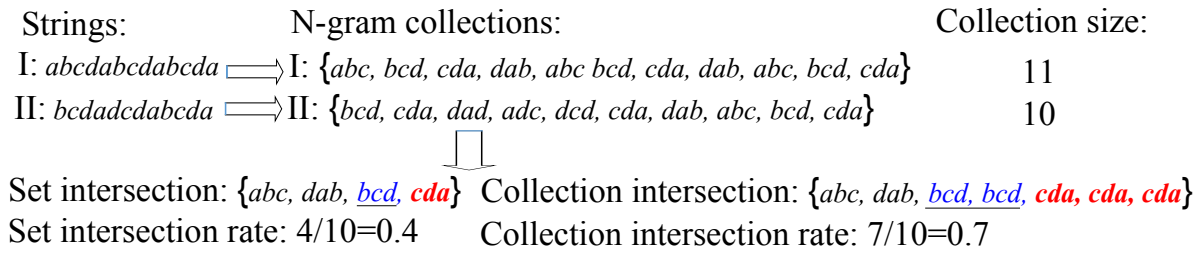


Figure 3.1: An example illustrating the difference between set intersection and collection intersection in handling duplicates for 3-grams.

Notation used in my algorithms is shown in Table 3.1, including collection identifier CID, size CSize (in terms of the number of items), occurrence frequency *Snum* of an item in one collection, occurrence frequency *Inum* of an item in an intersection, and intersection rate *Irate* of a content collection with respect to some sensitive data. Formally, given a content collection C_c and a sensitive data collection C_s, my algorithms aim to compute the intersection rate $Irate \in [0, 1]$ defined in Equation 3.1, where *Inum* is the occurrence frequency of an item i in the intersection $C_s \cap C_c$ (defined in Table 3.1). The sum of frequencies of all items appeared in the collection intersection is normalized by the size of the sensitive data collection or the size of content data collection, whichever is smaller. That yields the intersection rate *Irate*. The rate represents the percentage of sensitive data that appears in the content. *Irate* is also referred to as the *sensitivity score* of a content collection.

$$Irate = \frac{\sum_{i \in \{C_s \cap C_c\}} Inum_i}{min(|C_s|, |C_c|)} \tag{3.1}$$

Table 3.1: Notations used in my MapReduce algorithms.

Syntax	Definition
CID	An identifier of a collection (content or sensitive data)
CSize	Size of a collection
Snum	Occurrence frequency of an item
Inum	Occurrence frequency of an item in an intersection
CSid	A pair of CIDs $\langle CID_1, CID_2 \rangle$, where CID_1 is for a content collection and CID_2 is for a sensitive data collection
Irate	Intersection rate between a content collection and a sensitive data collection as defined in Equation 3.1. Also referred to as the sensitivity score of the content
ISN	A 3-item tuple of a collection ⟨identifier *CID*, size *CSize*, and the number of items in the collection⟩
CSS	An identifier for a collection intersection, consisting of an ID pair *CSid* of two collections and the size of the sensitive data collection *CSize*

3.1.3 Confidentiality of Sensitive Data

Naive collection-intersection solutions performing on *shingles* provide no protection for the sensitive data. The reason is that MapReduce nodes can easily reconstruct sensitive data from the shingles. My detection utilizes several methods for the data owner to transform shingles before they are released to the MapReduce nodes. These transformations, including specialized hash function, provide strong-yet-efficient confidentiality protection for the sensitive information. In exchange for these privacy guarantees, the data owner needs to perform additional data pre- and post-processing operations.

In addition to protecting the confidentiality of sensitive data, the pre-processing operations also need to satisfy the following requirements:

- Equality-preserving: the transformation operation should be deterministic so that two identical shingles within one session are always mapped to the same item for comparison.
- One-wayness: the function should be easy to compute given any shingle and hard to invert given the output of a sensitive shingle.
- Efficiency: the operation should be efficient and reliable so that the data owner is able to process very large content.

My collection intersection (in Section 3.3) is computed on one-way hash values of n-grams, specifically Rabin fingerprints. Rabin fingerprint is a fast one-way hash function, which is

computationally expensive to invert. In addition, Rabin fingerprint is a rolling hash function. It can be computed in linear time [33] when hashing n-grams, much faster than traditional hash functions.

Specifically, Rabin fingerprint of a n-bit shingle is based on the coefficients of the remainder of the polynomial modular operation with an irreducible polynomial $p(x)$ as the modulo as shown in Equation 3.2, where c_{n-i+1} is the i-th bit in the shingle C.

$$f(C) = c_1 x^{n-1} + c_2 x^{n-2} + ... + c_{n-1} x + c_n \ \ mod \ \ p(x) \tag{3.2}$$

Rabin fingerprint is efficient in hashing n-grams because the fingerprint of a shingle can be derived from the fingerprint of its preceding shingle. Suppose the current shingle I am hashing is $C_j = [c_1, c_2, \ldots, c_n]$. Its preceding shingle is $C_{j-1} = [c_0, c_1, \ldots, c_{n-1}]$. How the fingerprint of C_j is computed is shown in Equation 3.3.

$$f(C_j) = f(C_{j-1})x - c_0 x^n + c_n \ \ mod \ \ p(x) \tag{3.3}$$

The computation can also be implemented with fast XOR, shift and table lookup operations. This makes Rabin fingerprints the perfect choice for hashing very large n-gram data set.

Section 3.4 presents the security analysis of my approach especially on the confidentiality of sensitive data.

3.2 Technical Requirements and Design Overview

In this section, I introduce MapReduce and the specific challenges when performing data leak detection with MapReduce. I further present the workflow of my detection framework and the overview of my collection intersection algorithm.

3.2.1 MapReduce

MapReduce is a programming model for processing large-scale data sets on clusters. With an associated implementation (e.g., Hadoop), MapReduce frees programmers from handling program's execution across a set of machines. It takes care of tasks scheduling, machine failures and inter-machine communications. A MapReduce algorithm has two phases: *map* that supports the distribution and partition of inputs to nodes, and *reduce* that groups and integrates the nodes' outputs. MapReduce data needs to be in the format of $\langle$*key, value*$\rangle$ pair, where *key* serves as an index and the *value* represents the properties corresponding to the key/data item. A programmer usually only needs to specify the *map* and *reduce* function to process the $\langle$*key, value*$\rangle$ pairs. Fig. 3.2 illustrates the process of a MapReduce program execution.

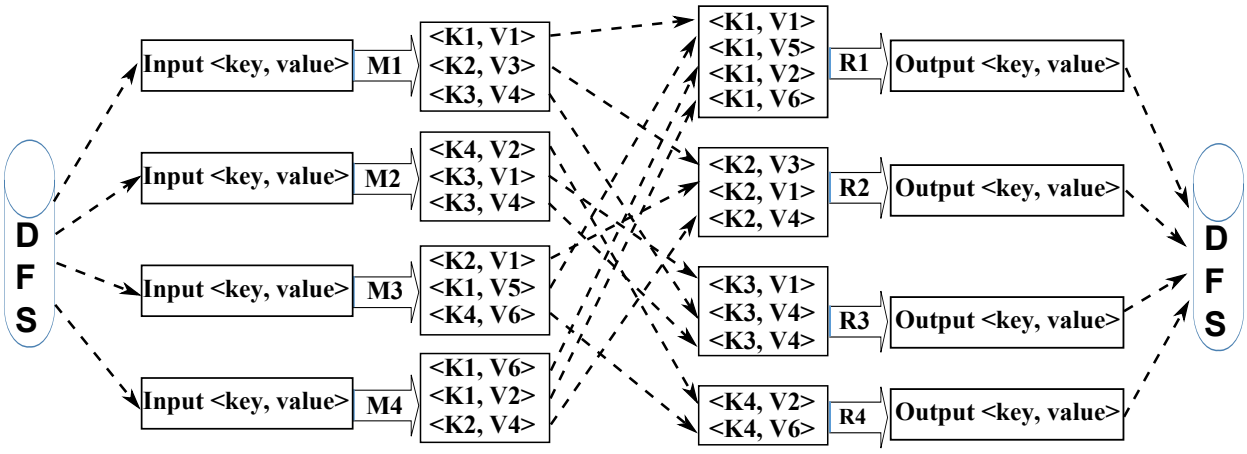


Figure 3.2: Overview of MapReduce execution process. *Map* takes each ⟨*key, value*⟩ pair as input and generates new ⟨*key, value*⟩ pairs. The output ⟨*key, value*⟩ pairs are redistributed according to the *key*s. Each *reduce* processes the list of *value*s with the same key and writes results back to DFS.

The input data in distributed file system is split and pre-processed by RecordReader. The output of RecordReader is a set of ⟨*key, value*⟩ pairs, which are sent to *map*. Each programmer specified *map* processes a ⟨*key, value*⟩ pair and generates a list of new ⟨*key, value*⟩ pairs. In Fig. 3.2, the first *map* generated three ⟨*key, value*⟩ pairs. The output ⟨*key, value*⟩ pairs are redistributed with the keys as indexes. All the pairs with key *K1* in Fig. 3.2 is processed by the first *reduce*. *Reduce* analyzes the group of values with the same key and writes the result back to distributed file system.

A significant of real world problems are able to be expressed by this model. A complex problem may require several rounds of map and reduce operations, requiring redefining and redistributing ⟨*key, value*⟩ pairs between rounds. New large-scale processing models are also proposed. For example, Google's Percolator [108] focuses on incrementally processing updates to a large data set. Muppet [86] provides a MapReduce-style model for streaming data. My collection intersection problem cannot be represented by the Percolator model. Although Muppet is able to perform collection intersection with streaming content, I do not use it due to its memory-heavy feature.

3.2.2 MapReduce-Based Design and Challenges

There exist several MapReduce-specific challenges when realizing collection-intersection based data leak detection.

1. **Complex data fields** Collection intersection with duplicates is more complex than set intersection. This requires the design of complex data fields for $\langle key,\ value \rangle$ pairs and a series of *map* and *reduce* operations.

2. **Memory and I/O efficiency** The use of multiple data fields (e.g., collection size and ID, shingle frequency) in $\langle key,\ value \rangle$ pairs may cause frequent garbage collection and heavy network and disk I/O.

3. **Optimal segmentation of data streams** While larger segment size allows the full utilization of CPU, it may cause insufficient memory problem and reduced detection sensitivity.

My data-leak detection algorithms in MapReduce addresses these technical challenges in MapReduce framework and achieves the security and privacy goals. I design structured-yet-compact representations for data fields of intermediate values, which significantly improves the efficiency of my algorithms. My prototype also realizes an additional post-processing partitioning and analysis, which allows one to pinpoint the leak occurrences in large content segments. I experimentally evaluate the impact of segment sizes on the detection throughput and identify the optimal segment size for performance.

3.2.3 Workload Distribution

The details of how the workload is distributed between data owner and DLD provider is as follows and shown in Fig. 3.3:

1. Data owner has m sensitive sequences $\{S_1, S_2, \cdots, S_m\}$ with a average size of $\mathcal{S}'$ and n content segments $\{C_1, C_2, \cdots, C_n\}$ with a average size of $\mathcal{C}'$. It obtains shingles from the content and sensitive data respectively. Then it chooses the parameters $(n, p(x), K, L)$, where n is the length of a shingle, $p(x)$ is the irreducible polynomial and L is the fingerprint length. The data owner computes Rabin fingerprints using Equation 3.2 and releases the sensitive collections $\{C_{S1}, C_{S2}, \cdots, C_{Sm}\}$ and content fingerprint collections $\{C_{C1}, C_{C2}, \cdots, C_{Cn}\}$ to the DLD provider.

2. DLD provider receives both the sensitive fingerprint collections and content fingerprint collections. It deploys MapRecuce framework and compares the n content collections with the m sensitive collections using my two-phase MapReduce algorithms. By computing the intersection rate of each content and sensitive collections pair, it outputs whether the sensitive data was leaked and reports all the data leak alerts to the data owner.

3. Data owner receives the data leak alerts with a set of tuples $\{(C_{Ci}, C_{Sj}), (C_{Ck}, C_{Sl}), \cdots\}$. The data owner maps them to suspicious content segments and the plain sensitive

sequences tuples $\{(C_i, S_j), (C_k, S_l), \cdots\}$. The data owner consults plaintext content to confirm that true leaks (as opposed to accidental matches) occur in these content segments and further pinpoints the leak occurrences.

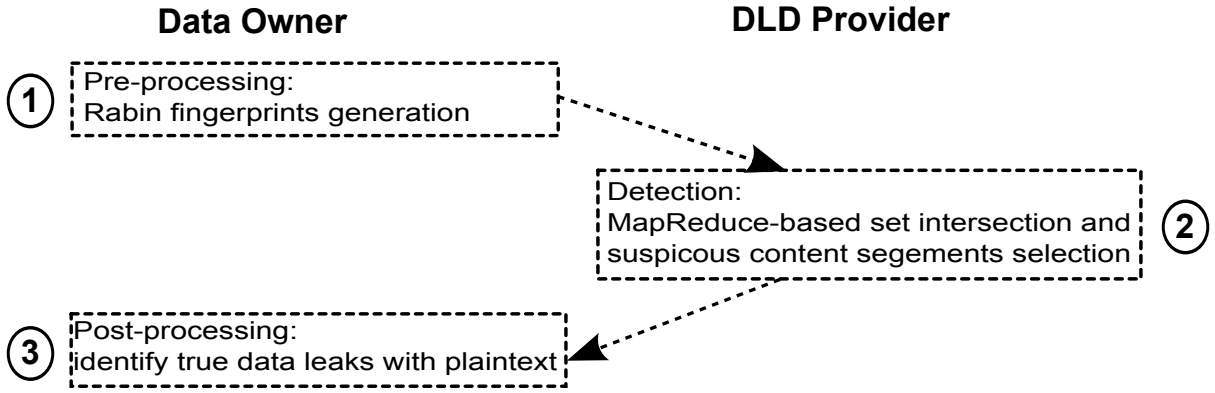


Figure 3.3: Workload distribution of DLD provider and data owner.

3.2.4 Detection Workflow

To compute the intersection rate of two fingerprint collections *Irate*, I design two MapReduce algorithms, Divider and Reassembler, each of which has a *map* and a *reduce* operation. Map and reduce operations are connected through a redistribution process. During the redistribution, outputs from map (in the form of $\langle key, value \rangle$ pairs) are sent to reducer nodes, as the inputs to the reduce algorithm. The key value of a record decides to which reducer node the record is forwarded. Records with the same key are sent to the same reducer.

1. Divider takes the following as inputs: fingerprints of both content and sensitive data, and the information about the collections containing these fingerprints. Its purpose is to count the number of a fingerprint's occurrences in a collection intersection (i.e., *Inum* in Equation 3.1) for all fingerprints in all intersections.

   In *map* operation, it re-organizes the fingerprints to identify all the occurrences of a fingerprint across multiple content or sensitive data collections. Each *map* instance processes one collection. This reorganization traverses the list of fingerprints. Using the fingerprint as the key, it then emits (i.e., redistributes) the records with the same key to the same node.

   In *reduce*, for each fingerprint in an intersection the algorithm computes the *Inum* value, which is its number of occurrences in the intersection. Each *reduce* instance processes one fingerprint. The algorithm outputs the tuple $\langle CSS, Inum \rangle$, where *CSS*

is the identifier of the intersection (consisting of IDs of the two collections and the size of the sensitive data collection[3]). Outputs are written to MapReduce file system.

2. REASSEMBLER takes as inputs $\langle CSS, Inum \rangle$ (outputs from Algorithm DIVIDER). The purpose of this algorithm is to compute the intersection rates (i.e., *Irate* in Equation 3.1) of all collection intersections $\{C_{c_i} \cap C_{s_j}\}$ between a content collection C_{c_i} and a sensitive data collection C_{s_j}.

   In *map*, the inputs are read from the file system and redistributed to reducer nodes according to the identifier of an intersection *CSS* (key). A reducer has as inputs the *Inum* values for all the fingerprints appearing in a collection intersection whose identifier is *CSS*. At *reduce*, it computes the intersection rate of *CSS* based on Equation 3.1.

In the next section, I present my algorithms for realizing the collection intersection workflow with one-way Rabin fingerprints. In Section 3.4, I explain why my privacy-preserving technique is able to protect the sensitive data against semi-honest MapReduce nodes and discuss the causes of false alarms and the limitations.

3.3 Collection Intersection in MapReduce

I present my collection-intersection algorithm in the MapReduce framework. The algorithm computes the intersection rate of two collections as defined in Equation 3.1. Each collection consists of Rabin fingerprints of n-grams generated from a sequence (sensitive data or content).

RECORDREADER is a (standard) MapReduce class. I customize it to read initial inputs into my detection system and transform them into the $\langle key, value \rangle$ format required by the *map* function. The initial inputs of RECORDREADER are content fingerprints segments and sensitive fingerprints sequences. For the DIVIDER algorithm, the RECORDREADER has two tasks: *i)* to read in each *map* split (e.g., content segment) as a whole and *ii)* to generate $\langle CSize, fingerprint \rangle$ pairs required by the *map* operation of DIVIDER algorithm.

3.3.1 DIVIDER Algorithm

DIVIDER is the most important and computational intensive algorithm in the system. Pseudocode of DIVIDER is given in Algorithm 1. In order to count the number of a fingerprint's occurrences in a collection intersection, the map operation in DIVIDER goes through the input $\langle CSize, fingerprint \rangle$ pairs, and reorganizes them to be indexed by fingerprint values. For each fingerprint in a collection, map records its origin information (e.g., *CID, CSize* of

[3]The sensitive data collection is typically much smaller than the content collection.

Algorithm 1 Divider: To count the number of a fingerprint's occurrences in a collection intersection for all fingerprints in all intersections.

Input: Output of RecordReader in a format of ⟨*CSize, Fingerprint*⟩ as ⟨*key, value*⟩ pair.
Output: ⟨*CSS, Inum*⟩ as ⟨*key, value*⟩ pair, where CSS contains content collection ID, sensitive data collection ID and the size of the sensitive data collection. $Inum$ is occurrence frequency of a fingerprint in the collection intersection.

```
1: function Divider::Mapper(CSize, Fingerprint)
2:     ▷ Record necessary information for the collection.
3:     ISN←CID, CSize and Snum
4:     Emit⟨Fingerprint, ISN⟩
5: end function
1: function Divider::Reducer(Fingerprint, ISNlist[c_1, ..., c_n])
2:     j = 0, k = 0
3:     ▷ Divide the list into a sensitive list and a content list
4:     for all c_i in ISNlist do
5:         if c_i belongs to sensitive collections then
6:             SensList[++j]← c_i
7:         else
8:             ContentList[++k] ← c_i
9:         end if
10:    end for
11:    ▷ Record the fingerprint occurrence in the intersection
12:    for all sens in SensList do
13:        for all content in ContentList do
14:            Size ←sens.CSize
15:            Inum ←Min(sens.Snum, content.Snum)
16:            CSS← ⟨content.CID, sens.CID, Size⟩
17:            Emit ⟨CSS, Inum⟩
18:        end for
19:    end for
20: end function
```

the collection) and *Snum* (fingerprint's frequency of occurrence in the collection). These values are useful for later intersection-rate computation. The advantage of using the fingerprint as index (key) in the map's outputs is that it allows the reducer to quickly identify non-intersected items.

After redistribution, entries having the same fingerprint are sent to the same reducer node as inputs to the reduce algorithm.

Reduce algorithm is more complex than map. It partitions the occurrences of a fingerprint into two lists, one list (*ContentList*) for the occurrences in content collections and the other

for sensitive data (*SensList*). It then uses a double for-loop to identify the fingerprints that appear in intersections. Non-intersected fingerprints are not analyzed, significantly reducing the computational overhead. This reduction is reflected in the computational complexity analysis in Table 3.2, specifically the $\gamma \in [0, 1]$ reduction factor representing the size of the intersection.

The for-loops also compute the occurrence frequency *Inum* of the fingerprint in an intersection. The output of the algorithm is the $\langle CSS,\ Inum \rangle$ pairs, indicating that a fingerprint occurs *Inum* number of times in a collection intersection whose identifier is *CSS*.

3.3.2 REASSEMBLER Algorithm

Algorithm 2 REASSEMBLER: To compute the intersection rates *Irate* of all collection intersections $\{C_{c_i} \cap C_{s_j}\}$ between a content collection C_{c_i} and a sensitive data collection C_{s_j}.

Input: Output of DIVIDER in a format of $\langle CSS,\ Inum \rangle$ as $\langle key,\ value \rangle$ pairs.
Output: $\langle CSid,\ Irate \rangle$ pairs where *CSid* represents a pair of a content collection ID and a sensitive collection ID, while *Irate* represents the intersection rate between them

```
1: function REASSEMBLER::MAPPER(CSS, Inum)
2:     Emit⟨CSS,Inum⟩
3: end function
```

```
1: function REASSEMBLER::REDUCER(CSS, Inum[n_1, ..., n_n])
2:     intersection ← 0
3:     ▷ Add up all the elements in Inum[]
4:     for all n_i in Inum[] do
5:         intersection ← intersection + n_i
6:     end for
7:     CSid ← CSS.CSid
8:     ▷ Compute intersection rate
9:     Irate ← |intersection| / CSS.CSize
10:    Emit ⟨CSid, Irate⟩
11: end function
```

The purpose of REASSEMBLER is to compute the intersection rates *Irate* of all collection-and-sensitive-data intersections. Pseudocode of REASSEMBLER is in Algorithm 2. The map operation in REASSEMBLER emits (i.e., redistributes) inputs $\langle CSS,\ Inum \rangle$ pairs according to their key *CSS* values to different reducers. The reducer can then compute the intersection rate *Irate* for the content and sensitive data collection pair. I.e., this redistribution sends all the intersected items between a content collection C_{c_i} and a sensitive data collection C_{s_j} to the same reducer.

3.3.3 Example of the Algorithms

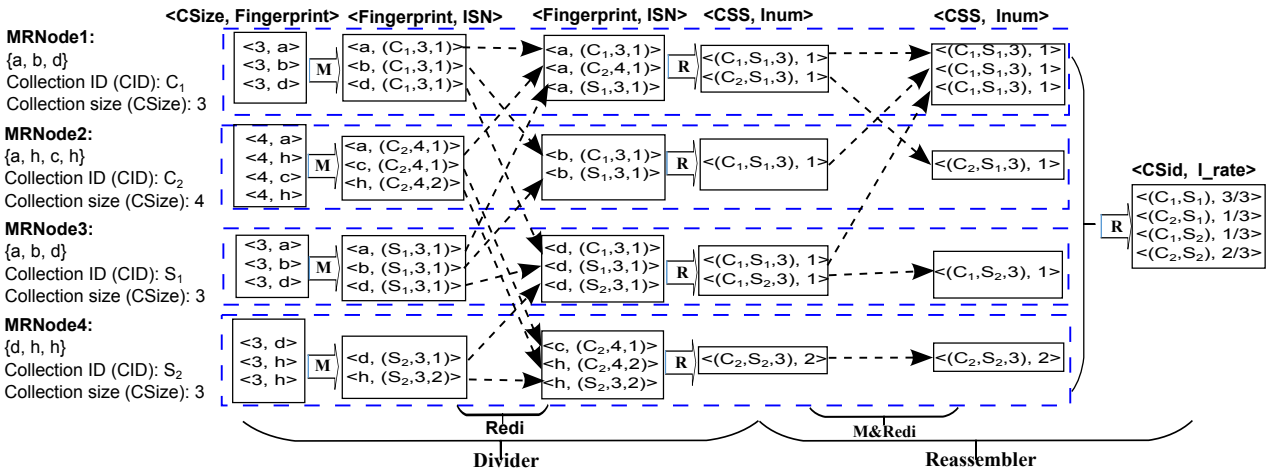


Figure 3.4: An example illustrating Divider and Reassembler algorithms, with four MapReduce nodes, two content collections C_1 and C_2, and two sensitive data collections S_1 and S_2. **M**, **R**, **Redi** stand for map, reduce, and redistribution, respectively. ⟨key, value⟩ of each operation is shown at the top.

Steps of my MapReduce algorithms are illustrated with an example (with four MapReduce nodes) in Fig. 3.4. The example has two content collections C_1 and C_2, and two sensitive data collections S_1 and S_2. The four data collections are generated by the data owner and sent to DLD provider. The sizes of the corresponding collections are 3, 4, 3 and 3, respectively. Each element (e.g., a) in the collections represents a fingerprint. The items below the steps indicate how the operations compute.

Step 1 Before the algorithms, the customized RecordReader reads the collections and sends ⟨*key, value*⟩ pairs to maps. In node 1, RecordReader parses collection C_1 by generating a ⟨*key, value*⟩ whenever it encounters an element. The key is the collection size 3 for C_1 and the value is the element it encounters.

- Node1: $\{a, b, c\} \Rightarrow \{\langle 3, a\rangle, \langle 3, b\rangle, \langle 3, c\rangle\}$
- Node2: $\{a, h, c, h\} \Rightarrow \{\langle 4, a\rangle, \langle 4, h\rangle, \langle 4, c\rangle, \langle 4, h\rangle\}$
- Node3: $\{a, b, d\} \Rightarrow \{\langle 3, a\rangle, \langle 3, b\rangle, \langle 3, d\rangle\}$
- Node4: $\{d, h, h\} \Rightarrow \{\langle 3, d\rangle, \langle 3, h\rangle, \langle 3, h\rangle\}$

Step 2 For the element a in node 1, *map* in Divider outputs the pair $\langle a, (C_1, 3, 1)\rangle$, indicating that fingerprint (key) a is from content collection C_1 of size 3 and occurs once in C_1. The

outputs are redistributed according the key values. All occurrences of fingerprint a are sent to node 1, including two occurrences from content collections C_1 and C_2, one occurrence from sensitive data collection S_1. Similar process applies to all the other fingerprints. Below shows how a is manipulated in different nodes.

- a in node1: $\langle 3, a\rangle \Rightarrow \langle a, (C_1, 3, 1)\rangle$
- a in node2: $\langle 4, a\rangle \Rightarrow \langle b, (C_2, 4, 1)\rangle$
- a in node3: $\langle 3, a\rangle \Rightarrow \langle c, (S_1, 3, 1)\rangle$

Step 3 Reduce in DIVIDER computes the intersection of content and sensitive collections for each fingerprint. In node 1, given the list of collections that fingerprint a exists, reduce algorithm uses a double for-loop and identifies that a appears in intersection $C_1 \cap S_1$ and intersection $C_2 \cap S_1$. The intersections are set as keys. The occurrence frequencies of fingerprint a are set as values. In node 1, a appears once in $C_1 \cap S_1$ and once in $C_2 \cap S_1$.

- Node1: $\langle a, \{(C_1, 3, 1), (C_2, 4, 1), (S_1, 3, 1)\}\rangle \Rightarrow$
  $\{\langle (C_1, S_1, 3), 1\rangle, \langle (C_2, S_1, 3), 1\rangle\}$
- Node2: $\langle b, \{(C_1, 3, 1), (S_1, 3, 1)\}\rangle \Rightarrow \{\langle (C_1, S_1, 3), 1\rangle\}$
- Node3: $\langle d, \{(C_1, 3, 1), (S_1, 4, 1), (S_2, 3, 1)\}\} \Rightarrow$
  $\{\langle (C_1, S_1, 3), 1\rangle, \langle (C_1, S_2, 3), 1\rangle\}$
- Node4: $\langle c, \{(C_2, 4, 1)\}\rangle \Rightarrow NULL;$
  $\langle h, \{(C_2, 4, 2), (S_2, 3, 2)\}\rangle \Rightarrow \{\langle (C_2, S_2, 3), 2\rangle\}$

Step 4 In REASSEMBLER, the outputs of DIVIDER are redistributed. All the pairs with the same intersection are sent to the same node. In node 1, all the occurrence frequencies of fingerprints in intersection $C_1 \cap S_1$ are collected. The total number of fingerprints shared by C_1 and S_1 is 3. The intersection rate is 1.

- Node1: $\langle (C_1, S_1, 3), \{1, 1, 1\}\rangle \Rightarrow \langle (C_1, S_1), 3/3\rangle$
- Node2: $\langle (C_2, S_1, 3), \{1\}\rangle \Rightarrow \langle (C_2, S_1), 1/3\rangle$
- Node3: $\langle (C_1, S_2, 3), \{1\}\rangle \Rightarrow \langle (C_1, S_2), 1/3\rangle$
- Node4: $\langle (C_2, S_2, 3), \{2\}\rangle \Rightarrow \langle (C_2, S_2), 2/3\rangle$

With the outputs of all the nodes, I get the intersection rates of all the intersections. The intersection rates are used to determine which content collections are suspicious.

3.3.4 Complexity Analysis

The worst-case computational and communicational complexities of various operations of my algorithm are shown in Table 3.2. I denote the average size of a sensitive data collection by $\mathcal{S}$, the average size of a content collection by $\mathcal{C}$, the number of sensitive data collections by m, the number of content collections by n, and the average intersection rate by $\gamma \in [0, 1]$. Without loss of generality, I assume that $|\mathcal{S}| < |\mathcal{C}|$ and $|\mathcal{S}m| < |\mathcal{C}n|$. I do not include post-processing in complexity analysis. The total communication complexity $O(\mathcal{C}n + \mathcal{S}mn\gamma)$ covers the number of records ($\langle key,\ value \rangle$ pairs) that all operations output. For a hashtable-based (non-MapReduce) approach, where each content collection is stored in a hashtable (total n hashtables of size $\mathcal{C}$ each) and each sensitive data item (total $\mathcal{S}m$ items) is compared against all n hashtables, the computational complexity is $O(\mathcal{C}n + \mathcal{S}mn)$.

Compared with the hashtable-based approach, the γ in my approach brings significant performance improvement, because it is extremely low for normal content without leak. In my algorithm, items not in the intersection (non-sensitive content) are quickly dropped without further processing. I experimentally verified the runtime deduction brought by γ in Section 3.5.4. In addition, the hashtable-based approach is not scalable of processing very large sensitive and content data sets.

Table 3.2: Worst-case computation and communication complexity of each phase in my MapReduce algorithm and that of the conventional hashtable-based approach. I denote the average size of a sensitive data collection by $\mathcal{S}$, the average size of a content collection by $\mathcal{C}$, the number of sensitive data collections by m, the number of content collections by n, and the average intersection rate by $\gamma \in [0, 1]$.

Algorithm	**Computation**	**Communication**
My Pre-processing	$O(\mathcal{C}n + \mathcal{S}m)$	$O(\mathcal{C}n + \mathcal{S}m)$
My Divider:Map	$O(\mathcal{C}n + \mathcal{S}m)$	$O(\mathcal{C}n + \mathcal{S}m)$
My Divider:Reduce	$O(\mathcal{C}n + \mathcal{S}mn\gamma)$	$O(\mathcal{S}mn\gamma)$
My Reassembler:Map	$O(\mathcal{S}mn\gamma)$	$O(\mathcal{S}mn\gamma)$
My Reassembler:Reduce	$O(\mathcal{S}mn\gamma)$	$O(mn)$
My Total	$O(\mathcal{C}n + \mathcal{S}mn\gamma)$	$O(\mathcal{C}n + \mathcal{S}mn\gamma)$
Hashtable	$O(\mathcal{C}n + \mathcal{S}mn)$	N/A

3.4 Security Analysis and Discussion

MapReduce nodes that perform the data-leak detection may be controlled by honest-but-curious providers (aka semi-honest), who follow the protocol, but may attempt to gain knowledge of the sensitive information (e.g., by logging the intermediate results and making

inferences). I analyze the security and privacy guarantees provided by my MapReduce-based data leak detection system.

3.4.1 Privacy Guarantee

The privacy goal of my system is to prevent the sensitive data from being exposed to DLD provider or untrusted nodes. Let f_s be the Rabin fingerprint of sensitive data shingle s. Using the algorithms in Section 3.3, a MapReduce node knows fingerprint f_s but not shingle s of the sensitive data. I assume that attackers are not able to infer s in polynomial time from f_s. This assumption is guaranteed by the one-way Rabin fingerprinting function [110]. This indicates that dictionary attack is not able to succeed in polynomial time.

In addition, the data owner chooses a different irreducible polynomial $p(x)$ for each session. Under this configuration, the same shingle is mapped to different fingerprints in multiple sessions. The advantage of this design is the increased randomization in the fingerprint computation, making it more challenging for the DLD provider to correlate values and infer preimage. This randomization also increases the difficulty of dictionary attacks. Other cryptographic mechanisms, e.g., XORing a secret session key with content and sensitive data before fingerprinting, achieve similar security improvement. Because the transformation needs to preserve equality comparison (in Section 3.1.3), the configuration needs to be consistent within a session.

3.4.2 Collisions

Two types of collisions are involved in my detection framework: fingerprint collisions and coincidental matches. Fingerprint collisions rarely happen as long as the length of fingerprint is sufficiently long. In my case, 64 bits fingerprints are sufficient long with the collision probability being less than 10^{-6}, according to the study by Broder [34]. Coincidental match occurs when some shingles in content happen to match some in sensitive data. These shingle matches may be due to shorter shingles, large content segment or sensitive data containing widely used patterns. With proper shingle length and threshold setting, the detection accuracy would not be affected by the coincidental matches. I perform accuracy experiment in Section 3.5.5, which shows that the intersection rate for normal leak is much higher than that caused by coincidental matches.

My data leak detection framework is designed to detect accidental data leaks in content, instead of intentional data exfiltration. In intentional data exfiltration, which cannot be detected by deep packet inspection, an attacker may encrypt or transform the sensitive data.

3.4.3 Discussion

I consider two data leak detection scenarios in my model: leakage in storage and leakage in transmission.

- Leakage in storage is the case when private data has already been leaked. The organizations may want to know whether their private data exists in unauthorized file systems.
- Leakage in transmission is the case when private data is being leaked through the outbound data transmission. The organizations may want to know whether their employees are sending sensitive documents to unauthorized parties through the network (e.g., emails).

One may think of using string matching in network intrusion detection system [92] to detect data leakage in transmission or keyword search [15] to detect leakage in storage. However, they only apply when the size of sensitive data is small or the patterns of all sensitive data are enumerable. It is not practical for the query string to cover all sensitive data segments for data leak detection.

My architecture also provides privacy guarantee for the organizations who need to perform data leak detection in untrusted clusters. For example, a government agency may need a contractor to perform detection. The contractor has a large amount of computing resources but a lower clearance level. The computation and storage resources that the DLD provider has are not limited to third party clusters (e.g., Amazon EC2). They could also be local, belonging to the same organization as the data owner. However, it is highly possible that the people who have access to the computation resources do not have the privilege to access sensitive data. My privacy-preserving data leak detection design also applies to this scenario.

Another concern is that the leaked data in content may span across two or more blocks after content segmentation thus affecting the detection accuracy. Actually, when the leaked sensitive data is much larger than the size of segment block, at least one block is filled with sensitive data. My detection algorithm as shown in Equation 3.1 is still able to get high intersection rate for that block thus being able to detect data leak. For the cases when small sensitive data spans across two large blocks, I propose to use a sliding window for segmentation. Suppose the threshold is set to be γ. The window slides $1 - \gamma/2$ of the block size to generate a new one. The two conjunctive blocks overlap $\gamma/2$ of the block size. This guarantees at least one of the block gets detectable intersection rate (the intersection rate being higher than the threshold γ). According to my experiment in Section 3.5.5, setting γ to be 0.2 is enough for high detection rate and low false positive rate, thus only incurring 10% of overhead. The data owner can also adjust the size of segment according to the size of sensitive data.

Stream Processing Hadoop is the most popular implementation of the MapReduce model. I

discuss my prototype of the architecture in Hadoop for processing offline content in Section 3.5. However, the MapReduce-based architecture I provide is not limited to processing offline content in storage. The design and algorithms can also be implemented in Spark [9], which is another large scale data processing tool especially for machine learning and stream processing. Spark supports transformations of continuous stream data with functions $flatMap()$ and $reduceByKey()$, which could be implemented with my algorithms in Section 3.3. In addition, the DLD provider needs to specify the batch interval for the best performance. The data owner and the DLD provider also need to coordinate about the length of subsession so that one batch data would not cross two subsessions. I plan to discuss more details of stream processing in my future work.

Limitations My data leak detection approach is based on the similarity of two sets with each element in the sets has its frequency information. Therefore, the system is vulnerable to frequency analysis if the DLD provider has enough background frequency information of the n-grams. However, one of the most important requirements for a success frequency analysis is that the DLD provider has to know the specific type of sensitive data sets. For example, the n-gram frequencies in emails are very different from that in natural language. According to my experiments with Enron emails, one of the most frequently used 8-gram is ".com/xt_", which is not possible to exist in natural language. Frequency analysis also cannot succeed if the sensitive data does not have obvious frequency patterns. For example, if the sensitive data is an Excel file with SSN records in it, the attacker is not able to guess them.

There exist approaches that increase the difficulty of frequency analysis for sensitive data sets that have obvious frequency patterns. One may try to leverage the existing search over encrypted data techniques (e.g., [88, 97]), which are independent of my set-intersection based approach. However, these approaches incur significant communication overhead or computational overhead. As the consequence, they are suitable for very large scale data leak detection. A more practical way is to the divide both the sensitive data and content data into smaller blocks, with same shingles in different block transformed into different fingerprints with different irreducible polynomial. The frequency patterns are not able to be easily made use of if the block size is small enough. Existing techniques that can be applied to data leak detection is not completely secure against frequency analysis [80]. For now, complete secure privacy-preserving solution for data leak detection is still an open problem.

Another limitation of my approach is the bandwidth requirement of outsourced detection. Outsourcing large-scale pre-processed content data may require more time than the detection if the transmission speed is slow. The data owner may need to setup a local cluster or use other data transfer solutions such as Amazon Snowball [18] to share the workload or speedup the transmission.

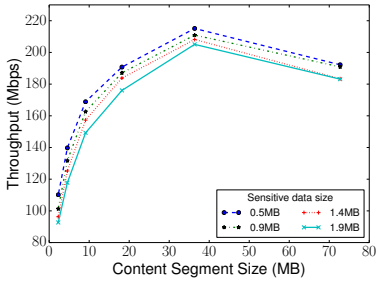


Figure 3.5: DLD provider's throughput with different sizes of content segments. For each setup (line), the size of the content analyzed is 37 GB.

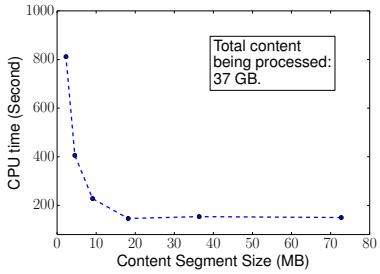


Figure 3.6: Data owner's pre-processing overhead on a workstation with different sizes of content segments. The total size of content analyzed is 37 GB.

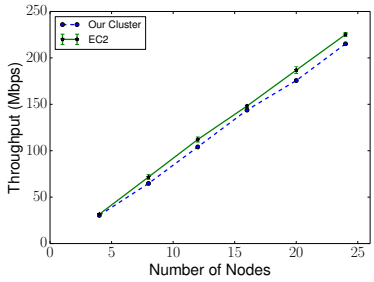


Figure 3.7: Throughputs with different numbers of nodes on a local cluster and Amazon EC2.

3.5 Implementation and Evaluation

I implement the algorithms with Java in Hadoop, which is an open-source software implementing MapReduce. I set the length of fingerprint and shingle to 8 bytes (64 bits). This length was previously reported as optimal for robust similarity test [34], as it is long enough to preserve some local context and short enough to resist certain transformations. My prototype implements an additional IP-based post-processing analysis and partition focusing on the suspicious content. It allows the data owner to pinpoint the IPs of hosts where leaks occur. I output the suspicious content segments and corresponding hosts.

I make several technical measures to reduce disk and network I/O. I use (structured) SequenceFile format as the intermediate data format. I minimize the size of $\langle key, value \rangle$ pairs. E.g., the size of *value* after *map* in Divider is 6 bytes on average. I implement Combination classes that significantly reduce the amount of intermediate results written to the distributed file systems (DFS). This reduction in size is achieved by aggregating same $\langle key, value \rangle$ pairs. This method reduces the data volume by half. I also enable Hadoop compression, which gives as high as 20-fold size reduction.

I deploy the algorithms in two different 24-node Hadoop systems, a local cluster and Amazon Elastic Compute Cloud (EC2). For both environments, I set one node as the master node and the rest as slave nodes.

- *Amazon EC2:* 24 nodes, each having a c3.2xlarge instance with 8 CPUs and 15 GB RAM.
- *Local cluster:* 24 nodes, each having two quad-core 2.8 GHz Xeon processors and 8 GB RAM.

I use the Enron Email Corpus, including both email header and body to perform the performance experiments. The entire dataset is used as content and a small subset of it as the sensitive data. For the accuracy experiments, I use two kinds of datasets. For plain text data leak, I use 10 academic research papers as the sensitive data and transformed/non-transformed Enron emails as content (insert `</br>` at each new line). For binary (e.g., pdf) data leak, I use four kinds of binaries (i.e., images, pdfs, zips and Windows Office documents) as sensitive data and emails that have part of the binaries as attachments as content.

My experiments aim to answer the flowing research questions (RQ).

1. *RQ1: Whether one host can perform large-scale data leak detection? (Section 3.5.1)*
2. *RQ2: How does the size of content segment affect the analysis throughput? (Section 3.5.2)*
3. *RQ3 What are the throughputs of the analysis on Amazon EC2 and the local clusters? (Section 3.5.3)*
4. *RQ4: How does the size of sensitive data and intersection rate γ affect the detection performance? (Section 3.5.4)*
5. *RQ: What is the detection accuracy of my data leak detection system? (Section 3.5.5 and Section 3.5.6)*

Suppose I have n content segments with s of them containing sensitive sequences ($s < n$). During the detection, the content segments with intersection rates above the predefined sensitivity threshold t raise alerts in my detection algorithms. Suppose m content segments raise the alerts and s' of them are true leak instances. Then, I define the following metrics:

- detection rate as $\frac{s'}{s}$.
- false positive rate as $\frac{m-s'}{n-s}$.

3.5.1 Performance of A Single Host

To verify that one host alone cannot perform large-scale data leak detection, a single host version of the similarity-based detection algorithm was implemented and tested on one machine containing two quad-core 2.8 GHz Xeon processors and 8 gigabytes of RAM.

I tested the performance with different sizes of content and sensitive data and monitored the system. The performance is shown in Table 3.3. It has the same detection accuracy with my MapReduce approach in terms of collision rate.

This single machine-based detection system crashes with large content or sensitive data because of lacking sufficient memory. Thus, when the content or sensitive data is large,

Table 3.3: Detection throughputs (Mbps) on a single host with different content sizes and sensitive data sizes. * indicates that the system crashes before the detection is finished. Note that, for shingle size to be 8, the size of input for the hashtable is 8 times the size of content.

Content \ Sensitive	0.9 MB	1.4 MB	1.9 MB
588 MB	22.08	20.81	*
1229 MB	24.21	*	*
2355 MB	25.7	*	*
4710 MB	*	*	*

a single host is not capable of completing the detection due to memory limitation and thrashing. Restricting the size of hashtable may cause collisions and incur false positives. This experiment confirms the importance of my parallel data leak detection framework with MapReduce.

3.5.2 Optimal Size of Content Segment

Content volume is usually overwhelmingly larger than sensitive data, as new content is generated continuously in storage and in transmission. Thus, I evaluate the throughput of different sizes and numbers of content segments in order to find the optimal segment size for scalability on DLD provider's side. A content segment with size $\hat{C}$ is the original sequence that is used to generate the n-gram content collection. A sensitive sequence with size $\hat{S}$ is the original sequence that is used to generate the n-gram sensitive collection.

The total size of content analyzed is 37 GB[4], which consists of multiple copies of Enron data. Detection performance under different content segment sizes (from 2 MB to 80 MB) is measured. I vary the size of sensitive data from 0.5 MB to 2 MB. The results are shown in Fig. 3.5.

I observe that when $\hat{C} < 37$ MB, the throughput of my analysis increases with the size $\hat{C}$ of content segment. When $\hat{C}$ becomes larger than 37 MB, the throughput begins to decrease. The reason for this decrease is that more computation resources are spent on garbage collection with larger $\hat{C}$. There are over 16 processes running at one node at the same time. I assign 400 MB memory for each process to analyze 37x8 MB shingles. One can adjust the optimal segment size by configuring the size of memory allocated for each process.

I also evaluate data owner's overhead of generating fingerprints with the same datasets. I performed the experiment on a quad-core 3.00 GHz Xeon processor and 8 GB RAM machine

[4]The content generates 37*8 GB shingles for shingle size to be 8. The 37*8 GB shingles are the input of the MapReduce system.

running Ubuntu 14.04.1 . The results are shown in Fig. 3.6. I observe that the CPU time of generating fingerprints falls off quickly with the content segment size increasing. When $\hat{C} > 20$ MB, the CPU time is less than 100 seconds, meaning that the throughput is more than 3000 Mbps. The total time of pre-processing is over 40 minutes due to the speed limit of I/O. However, pre-processing can be easily parallelized. The time of generating fingerprints is linearly decreased with multiple machines processing the content at the same time. This fact indicates that the data owner can handle fingerprint generation process without significant overhead incurred.

Thus, I set the size of content segments to 37 MB for the rest of the experiments. This size also allows the full utilization of the Hadoop file system (HDFS) I/O capacity without causing out-of-memory problems.

3.5.3 Scalability

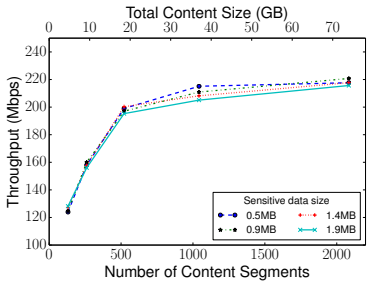


Figure 3.8: Throughput with different amounts of content workload. Each content segment is 37 MB.

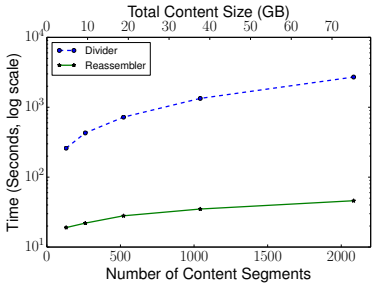


Figure 3.9: Runtime of DIVIDER and REASSEMBLER algorithms. The DIVIDER operation takes 85% to 98% of the total runtime. The Y-axis is in 10 based log scale.

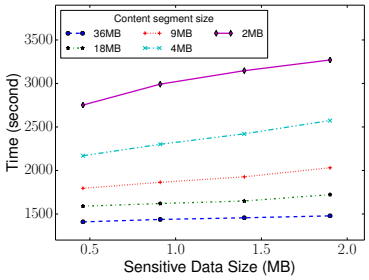


Figure 3.10: Runtime with a varying size of sensitive data. The content volume is 37.5 GB for each setup. Each setup (line) has a different size for content segments.

For scalability evaluation, I processed 37 GB content with different numbers of nodes, 4, 8, 12, 16, 20, and 24. The experiments were deployed on both on the local cluster and on Amazon EC2. My results are shown in Fig. 3.7. The system scales well, as the throughput linearly increases with the number of nodes. The peak throughput observed is 215 Mbps on the local cluster and 225 Mbps on Amazon EC2. EC2 cluster gives 3% to 11% performance improvement. This improvement is partly due to the larger memory. The standard error bars of EC2 nodes are shown in Fig. 3.7 (from three runs). Variances of throughputs on the local cluster are negligible.

I also evaluate the throughput under a varying number of content segments n, i.e., workload. The results are shown in Fig. 3.8, where the total size of content analyzed is shown at the

top X-axis (up to 74 GB). Throughput increases as workload increases as expected.

In the experiments above, the size of sensitive data is small enough to fit in one collection. Larger size of sensitive data increases the computation overhead, which explains the slight decrease in throughputs in both Figures 3.5 and 3.8.

I break down the total overhead based on the Divider and Reassembler operations. The results are shown in Fig. 3.9 with the runtime (Y-axis) in a log scale. Divider algorithm is much more expensive than Reassembler, accounting for 85% to 98% of the total runtime. With increasing content workload, Divider's runtime increases, more significantly than that of Reassembler.

These observations are expected, as Divider algorithm is more complex. Specifically, both *map* and *reduce* in Divider need to touch *all* content items. Because of the large content volume, these operations are expensive. In comparison, Reassembler algorithm only touches the intersected items, which is substantially smaller for normal content without leaks. These experimental results are consistent with the complexity analysis in Table 3.2.

3.5.4 Performance Impact of Sensitive Data

I reorganize the performance results in Fig. 3.5 so that the sizes of sensitive data are shown at the X-axis. The new figure is Fig. 3.10, where each setup (line) processes 37 GB content data and differs in their sizes for content segments. There are a few observations. First, smaller content segment size incurs higher computational overhead, e.g., for keeping tracking the collection information (discussed in Section 3.5.2).

The second observation is that the runtime increases as the size of sensitive data increases, which is expected. Experiments with the largest content segment (bottom line in Fig. 3.10) have the smallest increase, i.e., the least affected by the increasing volume of sensitive data.

This difference in intercept is explained next. The total computation complexity is $O(\mathcal{C}n + \mathcal{S}mn\gamma)$ (Table 3.2). In $O(\mathcal{C}n + \mathcal{S}mn\gamma)$, $n\gamma$ serves as the coefficient (i.e., intercept), as the total size $\mathcal{S}m$ of sensitive data increases, where n is the number of content segments. When 37 GB content is broken into small segments, n is large. A larger coefficient magnifies the increase in sensitive data, resulting in more substantial overhead increase. Therefore, the line at the bottom of Fig. 3.10 represents the recommended configuration with a large 37 MB content segment size.

I further verify the impact of γ on performance in Fig. 3.11. In this experiment, I manually change the value of γ by manipulating the percent of sensitive data leaked in content. I measure the runtime of Divider phase and Reassembler phase on 9 GB content and 9 GB sensitive data with segment size as 37 MB. Fig. 3.11 shows that the runtimes of both the two jobs drop linearly when the average intersection rate γ goes small. When γ is almost 0, which means there isn't any sensitive data leaked in content, the runtime of Reassembler

is almost 0 second and the runtime of Divider is about 1500 seconds. This observation is consistent with the fact that Reassembler's computation complexity is $O(\mathcal{C}n + \mathcal{S}m + \mathcal{S}mn\gamma)$ and Divider's computation complexity is $O(\mathcal{S}mn\gamma)$.

3.5.5 Plain Text Leak Detection Accuracy

I simulated two kinds of data leak cases that may possibly happen in the real world: plain text data leak and binary data leak. In the case of plain text data leak, a careless person may paste sensitive data on blogs or in emails. I used Enron emails as content and 10 academic papers as sensitive data. I pre-processed the content into four types: content without containing sensitive data (C_c), content containing sensitive data (C_s), transformed content without containing sensitive data (T_c) and transformed content containing sensitive data (T_s). I transformed the content data by inserting `</br>` at each new line. Detection rates are computed from C_s and T_s. If alerts are raised by the segments from these two types of data, the alerts are true positives. The false positive rates are computed from C_c and T_c. If alerts are are raised by segments from these two types of data, the alerts are false positives. The results on false positive rates and detection rates are shown in Fig. 3.12.

With sensitivity threshold t ranging from 0.18 to 0.82, the detection rate is 1, with 0 false positive rates for both the transformed leak and the non-transformed leak. The false positive rates of transformed data leak and non-transformed data leak are very close (the two lines mostly overlap). The false positive rate of the transformed data leak is on average 2.55% lower than that of the non-transformed leak, when the sensitivity threshold is smaller than 0.18. The reason is that transformation reduces the number of coincidental matches. When the sensitivity threshold ranges from 0.3 to 0.7, the detection rate is 1 and false positive rate is 0.

3.5.6 Binary Leak Detection Accuracy

In binary data leak, sensitive data is in binary format (e.g., .pdf). The difference between binary leak and plain text leak is that binary data is usually represented in ASCII string format when transferred. For example, the binary attachments of an email are encoded with Base64 scheme. I used four categories of binaries as sensitive data: Images, PDFs, Zips, Microsoft Office documents. Each has 20 files leaked and other 20 files not leaked. The emails with the leaked files as attachments are content. I transformed the sensitive datasets with Base64 encoding scheme and performed the detection with the transformed datasets. The details of how the experiment is set up are shown in Table 3.4 The results on false positive rates and detection rates are shown in Fig. 3.13.

With sensitivity threshold ranging from 0.35 to 0.91, the detection rate is 1, with 0 false positive rates for all the four binaries. The false positive rates of images and compressed

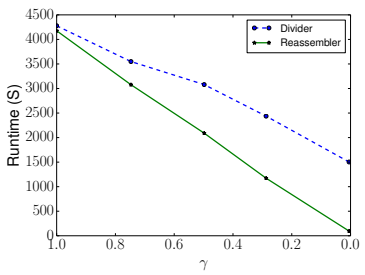


Figure 3.11: Runtimes of Divider and Reassembler with different values of γ. The runtimes drop linearly when the value of γ goes small. The result is consistent with the complexity analysis in Table 3.2.

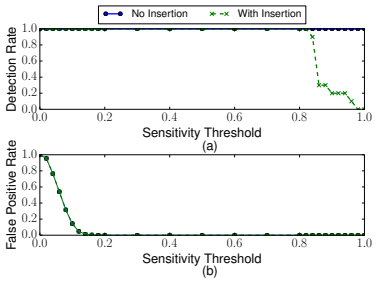


Figure 3.12: Plain text leak detection accuracy with different thresholds. The detection rate is 1 with very low false positive rate when the threshold ranges from 0.18 to 0.8. In Figure (b), the false positive rates of the two lines are very close.

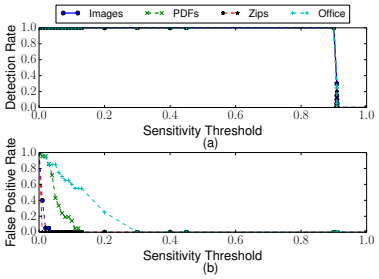


Figure 3.13: Binary leak detection accuracy with different thresholds. The detection rate is 1 with very low false positive rate when the threshold ranges from 0.35 to 0.9. In Figure (a), the detection rates of the four lines are very close.

binaries are both very low when compared with PDFs and Office documents. The reason is that compression may reduce the repeated patterns that happen to exist in the content. While images may contain fewer common patterns than documents. Another observation is that Office documents leak has higher false positive rate than plain text leak. One of the reasons is that the ratio of output bytes to input bytes of Base64 is 4:3. As the consequence, the effective shingle length is reduced to 6 bytes from 8 bytes, which makes more coincidental matches.

To summarize, binary sensitive data needs to be transformed in pre-processing step based on what content data to detect. In email content, the binaries need to be encoded with Base64 scheme. I am capable of detecting binary leaks with detection rate to be 1 and false positive rate to be 0. Base64 encoding increases accidental matches thus increasing false positive rate. Compression may reduce repeated patterns thus reducing false positive rate. If a sensitive plain text file is encoded into .pdf or .zip file before being leaked, the sensitive plain text file also needs to be encoded accordingly before detection.

Summary I summarize my experimental findings below.

1. My MapReduce-based data leak detection algorithms linearly scale with the number of nodes. I achieved 225 Mbps throughput (2.3 TB/day) on Amazon EC2 cluster and a similar throughput on the local cluster with only 24 nodes. *Divider* algorithm accounts for 85% to 98% of the total runtime.

2. I observed that larger content segment size $\hat{C}$ (up to 37 MB) gives higher performance. This observation is due to the decreased amount of bookkeeping information for keeping

Table 3.4: Setup of binary leak detection.

Sensitive data type	Images	PDFs	Zips	Office
Sensitive file number	40 jpgs with 20 leaked	40 pdfs with 20 leaked	40 compressed pdfs with 20 leaked.	24 docs, 8 excels and 8 ppts. Half of them are leaked.
Average sensitive file size	456 KB	845 KB	742 KB	479 KB
Content	Gmails attaching the leaked images	Gmails attaching the leaked pdfs	Gmails attaching the leaked zips	Gmails attaching the leaked office files
False positive rates	The percentage of non-leak files whose intersection rates are above the threshold.			
Detection rates	The percentage of leaked files whose intersection rates are below the threshold.			

track of collections, which results in significantly reduced I/O overhead associated with intermediate results. Data owner can also handle fingerprint generation process without significant overhead incurred.

When the content segment size $\hat{C}$ is large (37 MB), I observed that the increase in the amount of sensitive data has a relatively small impact on the runtime. Given the content workload, larger $\hat{C}$ means fewer number of content segments, resulting in a smaller coefficient.

3. I validated that my detection system has high detection accuracy for both transformed and non-transformed plain text data leaks. By setting the threshold to be a proper value, my algorithms can detect the leaks with low false alerts.

4. The detection of binary data leaks has different false positive rates from plain text data leaks due to the binary encoding schemes. My detection system has high detection accuracy for common binary data leaks with proper threshold setting.

5. *Limitations* My method is designed for detecting accidental data exposure in content, but not for intentional data exfiltration, which typically uses strong encryption. Detecting malicious data leaks, in particular those by insiders, is still an active research area. Coincidental matches may generate false positives in my detection, e.g., an insensitive phone number in the content happens to match part of a sensitive credit card number. A possible mitigation is for the data owner to further inspect the context surrounding the matching shingles and interpret its semantics. For detecting data leak in network, the throughput with 24 nodes is able to keep up with the average outbound traffic of a medium organization (less than 2 TB outbound traffic per day). However, it is not true for peak load. This problem could be addressed by introducing more nodes into the cluster.

Chapter 4

Prioritize Inter-app Communication Risks

In this chapter, I present MR-Droid, a MapReduce-based parallel analytics system for *accurate* and *scalable* ICC risk detection [94]. The goal is to empirically evaluate ICC risks based on an app's inter-connections with other real-world apps, and detect high-risk pairs. My intuition is that an ICC is of high-risk not only because it has a proof-of-concept vulnerability, but more importantly the app is actually communicating with other apps through this ICC interface. This intuition is similar to a recent work that prioritizes proof-of-concept vulnerabilities in CVE [117]. Ideally, all vulnerabilities should be addressed, but my priority needs to be on those that are causing a real-world impact.

To achieve this goal, I construct a large-scale *ICC graph*, where each node is an app component and the edge represents the corresponding *inter-app* ICCs. To gauge the risk level of the ICC pairs (edge weight), I extract various features based on app flow analyses that indicate vulnerabilities. For instance, I examine whether the ICC pair is used to pass sensitive data, or escalate permissions for another app. With the ICC graph, I then *rank* the risk level of a given app by aggregating all its ICC edges connected to other apps.

My system allows app market administrators to accurately pinpoint market-wise high-risk apps and prioritize risk migration. Individual app developers can also leverage my results to assess their own apps. In addition, the ICC graph provides rich contexts on how an app vulnerability is exploited, and by (or with) which external apps. For the purpose of this work, I customize my features to capture three major ICC risks: app collusion (malicious apps working together on stealth attacks), intent hijacking and intent spoofing (vulnerabilities that allow unauthorized apps to eavesdrop or manipulate inter-app communications).

Several existing work have been proposed to assess the ICC vulnerabilities, ranging from to inter-app Intent analysis [31, 52, 103] to static data flow tracking [19, 36, 136]. Yet, most of these approaches focus one *individual* app at a time, ignoring its feasible communication

context, *i.e.*, how the vulnerable interface can be exploited by other real-world apps. In addition, single-app analyses produce overly conservative risk estimations, leading to a high number of (false) alarms [52, 60, 103]. Manually investigating all the alerts would be a daunting task for security analysts.

Recently, researchers start to co-analyze ICCs of two or more apps simultaneously. This allows researchers to gain empirical contexts on the actual communications between apps and produce more relevant alerts, *e.g.*, identifying privacy leak [90] and permission escalations [22, 118]. However, existing solutions are largely limited in scale due to the high complexity of pair-wise components analyses ($O(N^2)$). They were either applied to a much smaller set of apps (only a few hundreds, versus a few thousands in single-app analyses), or small sets of inter-app links. The only system that successfully processed a large number of apps (*e.g.*, 11K apps) is PRIMO [105]. However, PRIMO is an ICC mapping tool for app pairs, which does not provide security and risk analysis. Moreover, PRIMO is designed to run on a single workstation. The large memory consumption makes it challenging for market scale analysis.

My approach is scalable for analyzing large-scale Android apps. To scale up the system, I implement MR-Droid with a set of new MapReduce [57] algorithms atop the Hadoop framework. I carefully design the $\langle key,\ value \rangle$ pairs for each MapReduce job and balance the workload among MapReduce nodes. Instead of performing pair-wise ICC mapping, I leverage the hash partition functionality of MapReduce and achieve near-linear complexity. The high-level parallelization from MapReduce allows us to analyze millions of app pairs within hours using commodity servers.

I evaluate my systems on a large set of 11,996 Android apps collected from 24 major Google Play categories (13 million ICC pairs). My manual post-analysis confirms the effectiveness of my approach in reducing false, excessive alerts. For apps labeled as high-risk, I obtain a 100% true positive rate in detecting collusion, broadcast injection, activity- and service-launch based intent spoofing, and a 90% true positive rate for activity hijacking and broadcast theft detection. For app pairs labeled as medium- or low-risk, manual analysis show their actual risks are substantially lower, indicating the effectiveness of risk prioritization. My system is also highly scalable. With 15 commodity servers, the entire process took less than 25 hours for an average of only 0.0012 seconds per app pair. More importantly, my runtime experiment shows the computation time grows near-linearly with respect to the number of apps.

The empirical analysis also reveals new insights to app collusion and hijacking attacks. I find previously unknown real-world colluding apps that leak user data (*e.g.*, passwords) and escalate each other's permission (*e.g.*, location or network access).

In addition, I find a more stealth way of collusion using *implicit intents*. Instead of "explicitly" communicating with each other (easy to detect), the colluding apps using high customized (rare) actions to mark their implicit intent to achieve the same effect of explicit intent. This type of collusion cannot be detected by analyzing each app independently. Finally, I find third-party libraries and the automatically-generated apps have contributed

greatly to today's hijacking vulnerabilities.

I have four key contributions in this work.

- I propose an empirical analytics method to identify risks within inter-app communications (or ICC). By constructing a large *ICC graph*, I assess an app's ICC risks based on its empirical communications with other apps. Using a risk ranking scheme, I accurately identify high-risk apps.
- I design and implement a highly scalable MapReduce framework for my inter-app ICC analyses. With carefully designed MapReduce cycles, I avoid full pair-wise ICC analyses and achieve near-linear complexity.
- I evaluate my system on 11,996 top Android apps under 24 Google Play categories (13 million ICC pairs). My evaluation confirms the effectiveness of my approach in reducing false alerts (90%-100% true positive rate) and its high scalability.
- My empirical analysis reveals new types of app collusion and hijacking risks (e.g., transferring user's sensitive information including password[1] and leveraging rarely used implicit intents[2]). Based on my results, I provide practical recommendations for app developers to reduce ICC vulnerabilities.

4.1 Models & Methodology

In this section, I describe the threat model and the security insights of large-scale inter-app ICC risk analysis. Then I introduce the high-level methodology of my approach.

4.1.1 Threat Model

My work focuses on security risks caused by inter-app communications realized through ICCs , covering three most important classes of inter-app ICC security risks.

- **Malware collusion.** Through inter-app ICCs, two or more apps may collude to perform malicious actions [99, 100, 119] that none of the participating apps alone would be able to. The collusion can be realized either by passing Intents to exported components or by using the same `sharedUserId` among the malicious apps. Malware collusion can result in disguised data leak and system abuse.

[1] `uda.onsiteplanroom` and `uda.projectlogging`

[2] `org.geometerplus.fbreader.plugin.local_opds_sanner` and `com.vng.android.zingbrowser.labanbo-okreader`

- **Intent hijacking.** An Intent sent by an app via an implicit ICC may not reach the intended recipient. Instead, it may be intercepted (hijacked) by an unauthorized app. This threat scenario, referred to as Intent hijacking, can be categorized into three subclasses according to the type of the sending component: broadcast theft, activity hijacking, and service hijacking, as introduced in [52]. Intent hijacking may lead to data/permission leakage and phishing attack.

- **Intent spoofing.** By sending Intents to exported components of a vulnerable app, an attacker can spoof the vulnerable (receiving) app to perform malicious actions, including those that are to be triggered only by the Android system. Intent spoofing can be classified into three subclasses by the type of the receiving component: malicious broadcast injection, malicious activity launch, and malicious service launch [52].

4.1.2 Security Insights of Large-Scale Inter-app Analysis

In this section, I use an example to illustrate the security differences between single-app analysis and pairwise app analysis. I explain why large-scale pairwise analysis is useful and necessary.

Listing 4.1 and 4.2 show parts of two apps, A and B, between which inter-app ICC introduces risks. A has access to location information and sends it out through an implicit Intent. The manifest of B defines an Intent filter that restricts the types of Intents to accept. Once A's Intent passes through the Intent filter, B then sends the received data out through SMS to a phone number (`phoneNo`).

With single-app analysis (*e.g.*, ComDroid [52]), B would be identified as *always* vulnerable to Intent spoofing attacks because its `SendSMSService` component is exported without permission protection (other apps can send SMS through B). Likewise, A would be detected as *always* vulnerable too, as other apps could hijack the implicit Intent it sends out and get the location information. However, single-app analysis does not consider specific communication contexts. It cannot track the destination of the implicit sensitive Intent `i` in A, and unable to identify which communicating peers may hijack `i` and abuse the sensitive data. The Android design of ICC makes this type of vulnerabilities prevalent. ComDroid [52] reported that over 97% of the studied apps are vulnerable. However, reporting an app as generically vulnerable or malicious is overly-conservative, and lead to insufficient precision and excessive alerts.

In practice, the risk becomes real when an app has actual communication contexts with other specific apps. For example, A's security risk increases if malicious apps are found being able to leak A's location information. More malicious hijacking apps trigger higher security risk because users have larger possibilities to install A and the malicious apps at the same time. In addition, A and B also may collude to leak the location information through the implicit Intent, which existing single-app analyses cannot detect both of the apps.

Listing 4.1: Code snippet of the app A which creates and sends out external ICC Intent carrying location info.

```
public void onClick(View v) {
 Intent i = new Intent();
 Location loc = locationManager.getLastKnownLocation();
 i.setAction ("android.Intent.action.SENDTO");
 i.setCategory ("android.Intent.category.DEFAULT");
 /*Set the intent data field as location*/
 i.setData(Uri.parse(loc.toString()));
 i.putExtra("PhoneNo.", "*********")
 startService(i);
}
```

Listing 4.2: The manifest and code snippet of the app B which processes received Intents and sends out intent data.

```
<service android:name=".SendSMSService" android:enabled="true">
<Intent-filter>
 <action android:name="android.Intent.action.SENDTO"/>
 <category android:name="android.Intent.category.DEFAULT"/>
</Intent-filter>
</service>
public void onStart(Intent Intent, int startId) {
 String pn = Intent.getStringExtra("phoneNo.");
 String msg = "Location: " + Intent.getData();
 /*Send intent data out through text message*/
 smsManager.sendTextMessage(pn, null, msg, null, null);
}
```

To these ends, I need to evaluate security risks for a given app by not just checking its own ICC properties, but also by carefully examining its external communications with its peers in a neighbor-aware manner. For example, the analysis should be able to track which app may hijack the Intent `i` of A and may collude with this app to abuse the location information; the analysis should give a detailed list of apps (*e.g.*, B) and their corresponding components (e.g., `B.SendSMSService`) involved in the attacks.

The need for the large-scale risk analyses arises from the two main limitations in existing approaches:

- **Large Threats to Validity.** Results are reported only with respect to small sets of sampled apps, and thus suffer potentially large threats to validity (biases) in their findings and/or conclusions.
- **Lesser Security Guarantee.** There is a potentially higher risk of missing true risk warnings because of the limited size of communication context considered (e.g., a highly vulnerable app may be declared safe when only a few external apps are analyzed).

In the rest of this chapter, I demonstrate how such a solution can be realized by a highly-scalable distributed ICC graph and, a neighbor-aware security analysis. My solution not only seeks to detect vulnerable apps but also label them with risk levels and rank accordingly. This provides a big picture for the app market to identify the most vulnerable apps to prioritize risk mitigation. Users can be aware of the risk of an app even before installing. The developers can be better guided to fix vulnerabilities when they are aware of the trade-off between functionality and vulnerability.

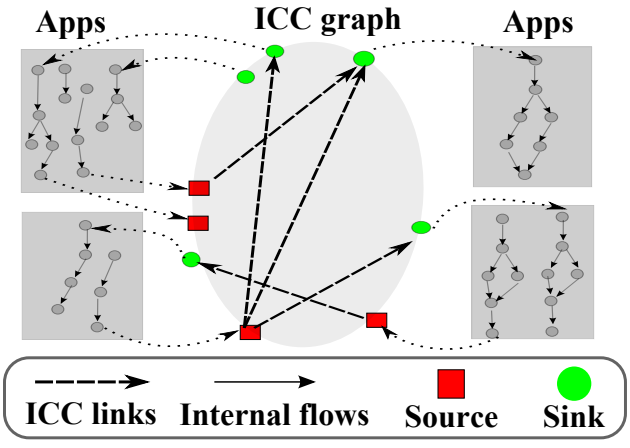


Figure 4.1: Illustration of data flows, inter-app Intent matching, and ICC graph. My MapReduce algorithms compute the complete ICC graph information for a set of apps. internal ICCs are excluded.

4.1.3 Computational Goal

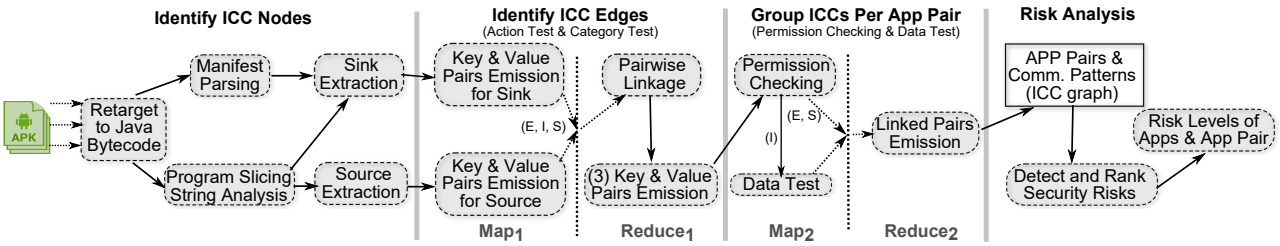


Figure 4.2: The workflow for analyzing Android app pairs with MapReduce. The dashed vertical lines indicate redistribution processes in MapReduce. E, I, S represent explicit edge, implicit edge, and sharedUserId edge respectively.

The computational goal is two-fold.

1. Build a complete inter-app ICC graph and identify all communication app pairs for a set of apps to provide the communication context (i.e., the neighbor set) for each one.

2. Further perform neighbor-aware inter-app security analysis on top of the ICC graph and rank the apps and app pairs with respect to their risk levels.

The definition for an ICC graph is given in Definition 4.1.

Definition 4.1. Inter-app ICC graph is a directed bipartite graph $G = (V_S, V_D, E_{V_S \to V_D})$, where each node $v \in V_S \cup V_D$ represents the specifications of an entry or exit point of external ICC of an app, and each edge $e \in E_{V_S \to V_D}$ represents the Intent-based communication between one app's ICC exit point $v_s \in V_S$ to another app's ICC entry point $v_d \in V_D$. A node in the set V_S in G is referred to as an *ICC source* or simply *source*. A node in the set V_D is referred to as an *ICC sink* or simply *sink*.

Figure 4.1 illustrates these definitions. In Table 4.1, I list some of the attribute details for sources and sinks. Sources represent the outbound Intents. Sinks represent Intent filter and/or public components.

Given apps, the construction of ICC graph requires *i)* identifying communication nodes (Intent, Intent filter, or component) and *ii)* identifying edges (properties of ICCs). Identifying edges requires attributes matching and testing between sources and sinks. Both nodes and edges information should be extracted from apps.

Table 4.1: Nodes in ICC graph and their attributes. `type` is either "Activity", "Service" or "Broadcast". `pointType` is either "Explicit", "Implicit" or "sharedUserId".

Node	Attributes
Source	`ID, type, destPackage, destComponent, action, categoryList, mimeType, uri, extraDataList, Permission, sharedUserId, pointType`
Sink	`ID, type, componentName, actionList, categoryList, permission, priority, mimeTypeList, schemeList, hostList, portList, pathList, pointType`

4.1.4 The Workflow

As shown in Figure 4.2, the workflow involves a preprocessing step 1. IDENTIFY ICC NODES, two MapReduce jobs referred to as 2. IDENTIFY ICC EDGES, 3. GROUP ICCS PER APP PAIR, and a 4. RISK ANALYSIS operation. Each MapReduce job consists of a *Map* and a *Reduce* algorithm.

1. In IDENTIFY ICC NODES, I extract the attributes of sources and sinks from apps. Sources are extracted from the attributes in outbound Intents. Sinks are extracted from the exported components in the manifest or from dynamic receivers created in the code. (Details are in Section 4.2.1.)

2. IDENTIFY ICC EDGES is the first MapReduce job which identifies edges between communicating sources and sinks. The MapReduce job transforms the source and sinks into $\langle key, value \rangle$ pairs, which enable parallel edge finding. (Details are in Section 4.2.2.)

3. GROUP ICCS PER APP PAIR is the second MapReduce job which performs the data test and permission checking, and identifies and groups edges belonging to the same pair.(Details are in Section 4.2.3.)

4. RISK ANALYSIS utilizes the ICC analysis results from previous phases to compute key ICC link features, and then uses the features to rank security risks for each app (for hijacking and spoofing attacks) and app pair (for collusion attacks). (Details are in Section 4.3.)

4.2 Distributed ICC Mapping

I now present the distributed market-wide ICC mapping algorithms in MR-Droid. The purpose of the algorithms is to build the ICC graph. My distributed ICC mapping algorithms involve three major operations: IDENTIFY ICC NODES, IDENTIFY ICC EDGES, and GROUP ICCS PER APP PAIR, where the last two operations are performed in MapReduce

framework. The entire workflow has many steps and is technically complex. For efficiency, my prototype integrates the tests (on action, category, data, and permission) with other edge-related operations in the last two operations. I describe details of each operation next.

4.2.1 Identify ICC Nodes

The purpose of *Identify ICC Nodes* is to extract all the sources and sinks from all available apps. I customized the Android ICC analysis tool IC3 [104] for this purpose.

Sink Extraction. I analyze each app's decoded manifest generated with IC3 [104] and retrieve the attributes for sinks listed in Table 4.1. I also parse the dynamic receivers of each app as exported receiver components.[3] A public component[4] may generate multiple sinks. The number is determined by the number of the component's Intent filters, as each Intent filter is extracted into a sink in ICC graph. A component is extracted into a sink only when it has no Intent filters.

Source Extraction. The attributes of sources are obtained by propagating values in fields of outbound Intents. I utilize IC3 [104] to perform program slicing and string analysis. String analysis may generate multiple possible values for one Intent field, which is due to multiple paths from where the string value is defined. I track all the possible values, and label them with the same identifier. Thus, there may be multiple values associated with an attribute of a source.

4.2.2 Identify ICC Edges and Tests

Identify ICC Edges and Action/Category Tests[5] operations are performed together in MapReduce. The purpose is to identify all matching source and sink pairs, which are connected with edges in the ICC graph. There are three types of edges: *explicit edge*, *implicit edge*, and *sharedUserId edge*, which are shown in Table 4.2. The explicit edge and implicit edge correspond to explicit and implicit intent. The sharedUserId edge corresponds to the private ICC communication using sharedUserId.

The Map_1 algorithm transforms sources or sinks into the $\langle key, value \rangle$ pairs. Table 4.3 shows how three types of edges in ICC graph are transformed into $<key, value>$ pairs in Map_1.

[3]The main difference between a dynamic receiver and the components in the Android manifest is that the former can only receive Intents when it is in pending or running status.

[4]A public component is a component that is available for access by other apps via ICCs.

[5]https://developer.android.com/guide/components/intents-filters.html

Table 4.2: Tests completed at different MapReduce phases and the edges that the tests are applicable to.

Name	**Phase**	**Apply to**
Action Test	Map_1, $Reduce_1$	Implicit Edges
Category Test	Map_1, $Reduce_1$	Implicit Edges
Data Test	Map_2	Implicit Edges
Permission Checking	Map_2	All Edges

Table 4.3: Summary of how three types of edges in ICC graph are obtained based on source and sink information and transformed into <*key, value*> pairs in Map_1.

	Key	**Value**
Explicit Edge	EFlag, source.type (sink.type), source.destComponent (sink.componentName);	hostPackage, source (sink);
Implicit Edge	IFlag, source.type (sink.type), source.action (sink.action), source.categoryList (sink.subCategoryList);	hostPackage, source (sink);
SharedUserId Edge	SFlag. sharedUserId, source.type (sink.type), source.destComponent (sink.componentName);	hostPackage, source (sink);

During the redistribution phase (after Map_1), the $\langle key, value \rangle$ pairs that have the same *key* are sent to the same reducer. $Reduce_1$ algorithm identifies qualified edges from the redistributed records. Qualification is based on action test and category test for implicit edges and exact string match for explicit and sharedUserId edges. These tests are efficiently performed in the redistribution phase with only the edges that pass the action test and category test sent to $Reduce_1$.

Outputs at the end of $Reduce_1$ phase are $\langle key, value \rangle$ pairs, where *key* consists of package names of a communicating app pair (corresponding to one ICC edge), and *value* contains properties of the edge.

4.2.3 Multiple ICCs Per App Pair

An app pair may have multiple ICC data flows between them. For the subsequent risk analysis (in Section 4.3), I need to identify and cluster all the inter-app ICCs that belong

to an app pair. Therefore, the main purpose of Group ICCs Per App Pair performed in MapReduce is to group together ICCs that belong to the same app pair. In addition, I perform permission checking on all the three types of edges and data test on implicit edges as shown in Table 4.2. The *key* is the package names of an app pair. A reducer in the $Reducer_2$ algorithm records the complete set of inter-app ICCs between two apps that pass the tests.

4.2.4 Workload Balance

The types of actions and categories are unevenly distributed with very different frequencies. This property leads to the unbalanced workload at different nodes in $Reduce_1$. It highly impacts the performance because most of the nodes are idle while waiting for the nodes with heavy workload. To address this problem, I add a tag before each key emitted by the Map_1. The tag helps to divide the large amount of key-value pairs, which should be sent to one reducer, into m parts feeding m reducers. The tags incur additional communication and disk storage overhead. The optimal parameter m is selected to maximize performance and achieve high scalability. I compare the operation time and computational complexity of my approach with other available inter-app analyses in Section 4.2.5.

My distributed ICC mapping conservatively matches the sources with all the potential sinks even if the links are of low likelihood. The full set of links provides security guarantees for the risk analysis. For example, collusion apps may leverage rarely used implicit intent specifications for targeted communication. Ignoring any low-likelihood links would miss detecting such attacks. My following security analysis incorporates all the possible links and gives a prioritized risk result to minimize security analysts' manual investigation.

4.2.5 Complexity Analysis

Table 4.4: Worst-case complexity of different phases of my approach

Approaches	Operation Time	Complexity
Identify ICC Nodes	$\mathcal{T}n$	$O(n)$
Identify ICC Edges	$t_l mn$	$O(mn)$
Group ICCs Per App Pair	$t_g mn$	$O(mn)$
Total (MapReduce)	$\mathcal{T}n + (t_l + t_g)mn$	$O(mn)$

I show operation times and computational complexities of different phases of my approach in Table 4.4. The average time for identifying ICC nodes for each app is denoted as $\mathcal{T}$, the average time for identifying ICC edges of two apps as t_l, the average time for grouping ICCs per app pair as t_g, the number of available apps as n and the average number of linked apps

for each app is m. Note that $t_l + t_g \ll \mathcal{T}$ and $m \ll n$.[6] I also assume that each app has a constant number of entry and exit points on average.

Other inter-app analyses usually run static analysis on two or more apps together (e.g., IccTA coupled with ApkCombiner). Assuming two apps are analyzed each time, the time complexity for parsing n apps is $O(n^2)$. The operation time is $k\mathcal{T}\binom{n}{2}$ with $2\mathcal{T}$ being the average analysis time of each two apps.

4.3 Neighbor-based Risk Analysis

Table 4.5: Features in my security risk assessment

Feature	Definition	Hijacking	Spoofing	Collusion
Data linkage	Number of outbound/inbound links that carry data	✓		✓
Permission leakage	Number of apps connected to via links that involve permission leaks	✓	✓	✓
Priority distribution	Mean and standard deviation of priority set for outbound/inbound links	✓	✓	
Link ambiguity	Number of outbound explicit links missing package prefix in the target component	✓		
Connectivity	Number of outbound/inbound links and/or number of connected apps	✓	✓	✓

I present my neighbor-aware security risk analysis of inter-app communications, covering the three common types defined in the threat model. Neighbors of each app are the apps connected to it in the ICC graph. The input of the risk analysis is the ICC graph produced by the MapReduce pipeline previously described, and the output is the risk assessment per communicating app pair in the graph for malware collusion threats, or per individual app for (Intent) hijacking and spoofing threats.

For the latter two types of risks for a single app, the neighbor-aware security analysis differs from previous approaches (e.g., ComDroid [52], Epicc [103]), because I examine ICC sinks and sources in the app and all of its inter-app ICC links. My approach adds more semantic and contextual information to the risk assessment than previous work. Including the neighbor sets as communication contexts reduces false warnings. In addition, the risk assessment identifies the presence of a risk (i.e., detection) and reports how serious that risk may be (i.e., risk ranking). In this section, I explain how I compute the security risks associated with app vulnerabilities and collusions.

4.3.1 Features

To leverage the neighbor sets for a more effective security risk analysis, I extract five key ICC-link features and utilize varying combinations of them for assessing different types of

[6] The ICC graph is sparsely connected as demonstrated in Section 4.4.

risks, as defined in Table 4.5. Note that the five features are all expressed as numerical values.

I further explain the definition of each feature as follows: (1) an ICC link carries data if the ICC Intent contains a non-empty data field; (2) a permission leak occurs when an unauthorized app gains access to the resources when it does not have the required permissions for them itself; (3) the priority of a link is a property of the Intent filter for the corresponding ICC Intent; I use median and standard deviation of priority values set for all inbound/outbound links to characterize the distribution; (4) when specifying the target component for an explicit ICC call, the name of the enclosing package of that component may be missing; (5) the number of ICC links and the number of connected apps are considered to quantify *link-connectivity* and *app-level connectivity*, respectively.

4.3.2 Hijacking/Spoofing Risk

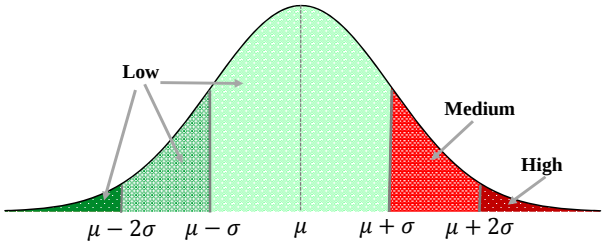


Figure 4.3: Feature value distribution and classification.

For a given app, my analysis first computes its risk with respect to individual features, and then aggregates them to obtain an overall risk value. Meanwhile, some features (e.g., *priority distribution*) are of high importance and thus are used to determine the overall risk level directly without involving other features.

Specifically, for hijacking and spoofing risks, given a feature f, the feature-wise risk of f for the subject app a is evaluated based on the distribution of all f values in the ICC graph, and classified into three risk levels from `low` to `high` as illustrated in Figure 4.3. The rationale is that the risk level is proportional to the feature value and all feature values in the ICC graph are *normally* distributed. I normalize the categorical value of each feature f by mapping (`low, medium, high`) to (.1, .5, 1.0) respectively. To incorporate the varying effects of different features on the overall risk, I assign customizable feature weights based on heuristics. The eventual risk value is computed as the weighted sum of feature-wise risk values. I upgrade the risk level if no apps are detected at the higher levels in order to normalize the risk levels. Next, I give details on the risk-evaluation procedures for each risk type.

Hijacking

Since the Intent-sending app is the subject of evaluation (i.e., the app for which risks are assessed) in a hijacking scenario, I consider outbound links only for all relevant features. Also, since an Intent is much less likely to be hijacked if the target is specified explicitly than implicitly, only implicit links are considered. The only exception is for *link ambiguity*, computing which involves just explicit links. This special feature captures the possibility of hijacking via ambiguous target-component specification—ignoring the package prefix in the target component name. Using these five features, the overall risk value is computed as follows.

- If *link ambiguity* ≥ 1 or the sum of the two *priority distribution* parameters is high (mean+standard deviation$\geq$500), report the risk as `high` and exit.

- Compute the numerical feature-wise risk values for *data linkage*, *permission leakage*, *app-level connectivity*, and *link-level connectivity*, and sum them up with weights .3, .4, .2, .1, respectively. Then, cast the resulting numerical risk value to its categorical risk level, according to where the value falls in the three equal-length subintervals of [0.1,1.0].

Spoofing

In a spoofing scenario, the Intent-receiving app is the subject of evaluation, thus I consistently consider inbound links only for the three features involved (see Table 4.5). Also, except for the *priority distribution* which counts implicit links only because the priority can only be set for implicit Intents, the other two features consider both implicit and explicit links. Nevertheless, as the spoofing attack is more likely to succeed when using explicit Intents than using implicit ones, I weigh explicit links higher in the composite quantification of the relevant features. Using these three features, the overall risk value is computed as follows.

- If the sum of the two *priority distribution* parameters is high (mean+STD$\geq$500), report the risk as `high` and exit.

- Compute the numerical feature-wise risk values for *permission leakage*, *app-level connectivity*, and *link-level connectivity* with respect to explicit links, and sum up them with weights .4, .2, and .2 (total of .8), respectively. Then, compute the same three risk numbers with respect to implicit links, and sum up them with weights 0.1, 0.05, and 0.05 (total of 0.2). Finally, add the six risk numbers up and cast the sum back to its categorical risk level, according to where it falls in the three equal-length subintervals of [0.1,1.0].

Table 4.6: Numbers of apps vulnerable to Intent spoofing/hijacking attacks and potentially colluding app pairs reported by my technique, and percentages (in parentheses) manually validated as indeed vulnerable or colluding, per risk category and level. The DroidBench test result is not included in this table.

	Intent Hijacking			Intent Spoofing			
	Activity Hijacking	Service Hijacking	Broadcast Theft	Activity Launch	Service Launch	Broadcast Injection	Collusion Pairs
High	94 (90%)	10 (70%)	15 (90%)	17(100%)	4(100%)	7(100%)	6 (100%)
Med.	790 (80%)	32 (60%)	303 (70%)	9(90%)	8(100%)	0	169(8.3%)
Low	11,112(20%)	11,954(0%)	11,678(10%)	11,970(2%)	11,984(0%)	11,989(0%)	12,986,079(0%)

4.3.3 Collusion Risk

In collusion attacks, all communicating pairs are analyzed together. The three features involved (see Table 4.5) count both inbound and outbound links. Also, both implicit and explicit links are used for feature extraction.

MR-Droid looks for the existence of links between the app pair, ignoring the number of links but examining whether the links are explicit or implicit. The analysis uses higher values for features due to explicit links than for those due to implicit ones. The overall risk level is computed as follows.

- *Connectivity* is quantified as 5 for unidirectional explicit connection (i.e., one app is connected to the other via explicit ICCs yet the other connects back via implicit ones), as 10 for bidirectional explicit connection (i.e., explicit ICCs exist in both directions), and as 3 for bidirectional implicit connection (i.e., no explicit ICC exists between the pair). The *data linkage* feature is quantified as 3 and 2 for data transfer through explicit and implicit links, respectively. *permission leakage* is evaluated as 3 if the leak potentially exists or 0 otherwise.

- Calculate as the aggregate risk value the sum of the three numerical feature values above, and cast the sum into its categorical risk level ranging from `low` to `high`, according to where it falls in the three equal-length subintervals of [1,16].

- Identify the apps with low implicit connectivity (empirically, the connectivity is *low* if the number of inbound or outbound implicit apps is less than 20), and retrieve the corresponding pairs with bidirectional implicit connections. Pairs with both apps having a low implicit connectivity are set as having a `high` risk of collusion. Pairs with only one app having a low implicit connectivity are set to have a `medium` risk of collusion.

4.4 Evaluation

I implemented the system with native Hadoop MapReduce framework. The input is the ICC sources and sinks extracted from individual apps using IC3 [104], the most precise single-app ICC resolution in the literature. I modified IC3 to accommodate the MapReduce paradigm. The Hadoop system is deployed on a 15-node cluster, with one node as master and the rest as slaves. Each node has two quad-core 2.8GHz Xeon processors and 8GB RAM.[7]

Datasets. For the evaluation, I apply my system to 11,996 most popular free apps from Google Play. I select the top 500 apps from each of the 24 major app categories (4 apps were unavailable due to bugs in program analysis). The detailed category list is shown in Table 4.7. I downloaded the apps in December 2014 with an Android 4.2 client. I ran the inter-app ICC analysis and security risk assessment on 12,986,254 app pairs.

In addition to this empirical dataset, I also test my system on DroidBench, the most comprehensive Android app benchmark for evaluating Android taint analysis tools. The latest suite DroidBench 3.0[8] consists of 8 app-pair test cases for evaluating inter-app collusions.

Validation. Because of the lack of ground truth on the empirical data, I devote substantial efforts to manually inspect the apps for validation.[9] In addition, I will evaluate the performance of the distributed ICC analyses by gauging the running time of each phase of the pipeline. The evaluation seeks to answer the following questions.

- *Q1: What are the risk levels of app pairs?* (Section 4.4.1)
- *Q2: How accurate is the risk assessment and ranking?* (Section 4.4.2)
- *Q3: What do detected attacks look like?* (Section 4.4.3)
- *Q4: What is MR-Droid's runtime, including per-app ICC resolution and ICC graph analyses?* (Section 4.4.4)

4.4.1 Q1: Results of Risk Assessment

I apply the system to the collected app dataset. The resulting ICC graph contains 38,134,207 source nodes, 26,227,430 sink nodes and 75,123,502 edges. On the per-app level, there are in total 12,986,254 app pairs that have at least one ICC link. Each app averagely connects with 1185 external apps (9.9% of all apps), confirming the overall sparsity of the graph. For non-connected app pairs, I can safely exclude them during the security analysis. The security

[7] The algorithms can also be implemented with Spark [9] with faster in-memory processing. It will require much larger RAMs to hold all the data.

[8] https://github.com/secure-software-engineering/DroidBench/tree/develop

[9] I plan to share these manually labeled datasets as benchmarks to the Android security community.

Table 4.7: App categories in the evaluation dataset

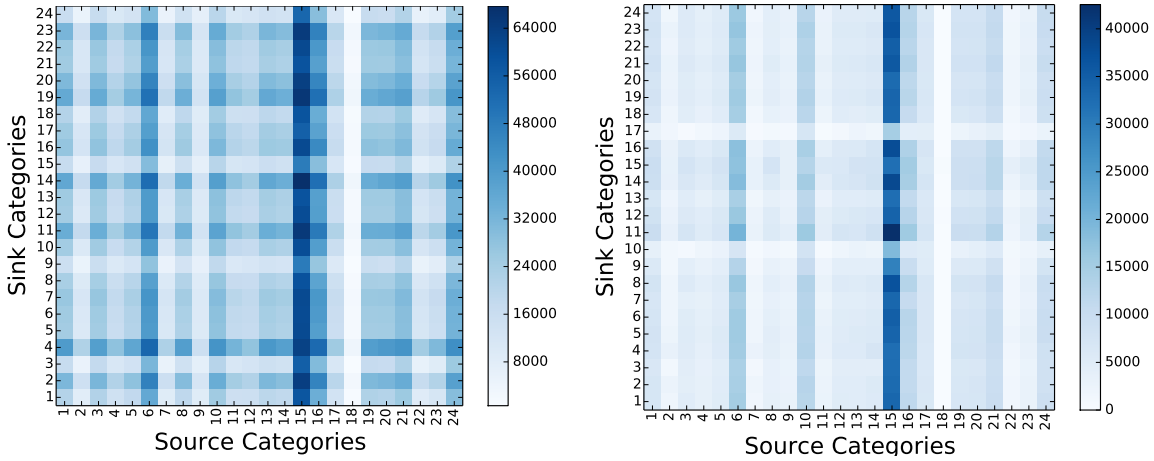
Index	Category	# of Apps
1	BOOKS_AND_REFERENCE	500
2	BUSINESS	498
3	COMICS	500
4	COMMUNICATION	500
5	EDUCATION	500
6	ENTERTAINMENT	500
7	FINANCE	500
8	HEALTH_AND_FITNESS	500
9	LIBRARIES_AND_DEMO	500
10	LIFESTYLE	499
11	MEDIA_AND_VIDEO	500
12	MEDICAL	500
13	MUSIC_AND_AUDIO	500
14	NEWS_AND_MAGAZINES	500
15	PERSONALIZATION	500
16	PHOTOGRAPHY	500
17	PRODUCTIVITY	500
18	SHOPPING	500
19	SOCIAL	499
20	SPORTS	500
21	TOOLS	500
22	TRANSPORTATION	500
23	TRAVEL_AND_LOCAL	500
24	WEATHER	500
	Total:	**11996**

analysis focuses on all potential security risks related to Intent hijacking, Intent spoofing and app collusion (Section 4.1.1). I quantify and rank security risks into as categorical risk levels following the procedures detailed in Section 4.3. In total, my system identified 150 high-risk apps, 1,021 medium and 10,825 low risk apps.

Table 4.6 summarizes the results, highlighting the numbers of apps or app pairs vulnerable to the *high* and *medium* level of risks. Prominently, stealthy attacks such as *activity hijacking* and *broadcast theft* dominate medium and high risks of any type. These attacks often involve passively steal user data.

The more intrusive types of attacks such as *service launch* and *broadcast injection* are less prevalent. In addition, considerably collusion-attacks are revealed by my analyses. There are six colluding app pairs are of high risk, and 169 are of medium risk.

Through risk ranking, I successfully prioritize alerts. Single-app analysis detects the vul-



(a) All communicating pairs (b) High & medium risk pairs

Figure 4.4: The heat maps of the number of app pairs across app categories. The tick label indexes represent 24 categories. Detailed index-category mapping in Table 4.7.

nerabilities based on whether the app has exposed its (component) interfaces. In contrast, my approach further examines an app's empirical neighbor set and determines whether it is currently subject to real threats.

App Categories. Figure 4.4 depicts the number of app pairs that communicate within or across categories. Figure 4.4(a) includes all the 13 million app pairs and Figure 4.4(b) shows pairs with at least one app in the high & medium risk lists. I observe that PERSONALIZATION (#15) apps have the most significant contribution to high-risk pairs (Figure 4.4(b)). These apps usually help users to download themes or ringtones using vulnerable implicit intents. For example, the app `com.aagroup.topfunny` provides options for user to download apps and ringtones from the web. A malicious app can easily hijack the implicit intent and redirect user to downloading malicious apps or visiting phishing websites. ENTERTAINMENT (#6) apps are also heavily involved high-risk ICCs in similar ways. For example, app `com.rayg.soundfx` also offers downloading ringtones via vulnerable implicit intents. Another high-risk category is LIFESTYLE (#10). These apps often require sensor data (*e.g.*, GPS, audio, camera) for their functionalities. One example vulnerable app is `com.javielinux.andando`. It is a location tracking app but it broadcasts GPS information through an implicit broadcast intent. Other apps can eavesdrop the broadcasted intent and acquire location data even if they don't have location permissions.

I find that categories such as FINANCE (#7) are less involved in high-risk ICCs. Since these apps often deal with banking and financial payments, it is likely that these apps have put more efforts on security. Even so, I still find high-risk apps under these categories. For example the app `com.ifs.androidmobilebanking.fiid3383` has links in its app to visit its website. However, they trigger implicit intents, which can be easily redirected to phishing

websites by malicious apps.

Validation with DroidBench. I have used DroidBench to confirm the effectiveness of the analysis. The latest suite DroidBench 3.0 consists of 8 app-pair test cases for evaluating inter-app collusion. My approach detected all 8 of them: 6 were labeled as high risk under *collusion*, and 2 were labeled as high risk under *intent hijacking*. My system put the two apps under hijacking category because the way they collude leads to a hijacking vulnerability. They use implicit intents with the default Category and "ACTION_SEND" Action. The experiment shows that many other apps can receive this type of implicit intents and acquire sensitive information.

4.4.2 Q2: Manual Validation

To further assess the usefulness of MR-Droid, I manually examined over 200 apps to verify the validity of the detection results. More specifically, I randomly select 10 apps from all 7 attack categories at all 3 risk-levels — for category/level with fewer than 10 apps, I chose all of them. I carefully inspected the selected apps in two ways to check their behavior: (1) static inspection, by which I examine relevant Intent attributes of each app and manually match them against peer apps (in its neighbor set); (2) dynamic verification, in which I run individual apps and app pairs on an Android emulator and observe suspicious behaviors in the activity logs.

The validation result is presented in (the parentheses of) Table 4.6. For each selected set of apps or app pairs, I report the percentage that was verified to be indeed vulnerable. Overall, this work justified my risk detection and ranking approach. I find apps labeled as high-risk have a much higher rate to be actually vulnerable. I have a 100% true positive rate in detecting collusion, broadcast injection, activity- and service-launch based intent spoofing. I have a 90% true positive rate for activity hijacking and broadcast theft detection. For apps labeled as low risks, the true positive rate is much lower: 5 out of 7 categories have a true positive rate of 0%. These results suggest that the rankings produced by my approach can help users and security analysts prioritize their inspection efforts.

Sources of Errors. My approach still have a few false alerts (at high-risk level). I find that most of them were caused by unresolved attributes in relevant Intent objects. In those cases, I conservatively match unresolved source points to all the sinks in order to cover possible ICCs. This matching leads to false positives. One improvement this is to combine the static program analysis with probabilistic models to better resolve ICC attributes [105]. Regarding false negatives, I cannot give a reliable estimation since I did not scan all apps within the market and due to the lack of ground truth.

4.4.3 Q3: Attack Case Studies

Based on my manual analysis, I present a few case studies to discuss empirical insights on ICC-based attacks.

Stealthy Collusion via Implicit Intents. During my analysis, I find some colluding apps also use *implicit* intents in an effort to avoid detection. Colluding apps usually use explicit intent, since they know "explicitly" which app(s) they are colluding with. However, using explicit intent makes the collusion easier to detect, even by single-app analyses. The new collusion uses highly customized "action" and "category" to mark the implicit intent, hoping no other apps will accidentally interrupt their communication. For example, the app `org.geometerplus.fbreader.plugin.local_opds_scanner` has open interfaces with a customized action name `android.fbreader.action.ADD_OPDS_CATALOG` in its intent filter. Another app `com.vng.android.zingbrowser.labanbookreader` sends implicit intent with the same action and leverages the first app to scan the local WiFi network (the second app itself does not have the permission). This type of collusion cannot be detected if each app was analyzed individually.

Risks of Automatically Generated Apps. I found many of the high-risk apps were automatically generated by app-generating websites (e.g., www.appsgeyser.com). These apps send a large number of implicit Intents, attempting to reuse other apps' functionality as much as possible. For example, `com.conduit.app_39b8211270ff4593ad85daa15cbfb9c6.app` is an automatically generated app and it contains a number of unprotected interfaces including those for viewing social-media feeds from Facebook and Twitter. It has 20,805 connections with other apps, *five* times more than the average.

Hijacking Vulnerabilities in Third-Party Libraries. I observed that a significant amount of vulnerable exit points are from third-party libraries. For example, a flash light app `com.ihandysoft.ledflashlight.mini.apk` bundles multiple third-party libraries for Ads and analytics. One of the libraries `com.inmobi.androidsdk` sends implicit intents to access external websites (*e.g.*, connecting to Facebook). A malicious app can hijack the Intent and redirect the user to a phishing website to steal the user's Facebook password.

Colluding Apps by the Same Developers. I find that colluding apps were usually developed by the same developers. One example is the `org.geometerplus.zlibraryui.android` and `org.geometerplus.fbreader.plugin.local_opds_scanner` app pair. The first app is a book reader app with 167,625 reviews and 10 million – 50 million installs. It leverages the later app (100K – 500K installs) to scan the user's local network interface. The first app itself does not have the permission to do so. Both of the two apps were developed by "FBReader.ORG Limited".

Another example is the pair of `uda.onsiteplanroom` and `uda.projectlogging`. The first app is for document sharing in collaborative projects, and the second app is for project

progress logging. With a click of a button, users will be redirected from the first app to the second app. User's sensitive information in the first app is also sent to the second app without user knowledge. Such information includes user's name, email address, password, etc. These two apps were written by the same developer "UDA Technologies, Inc.".

The practical risk of app collisions varies from case to case. The bottom line is, different apps written by the same developer should not open backdoors for each other to exchange user data and access privileges without the user knowledge.

Insecure Interfaces for Same-developer Apps. Usually, apps developed by the same developer have specialized interface to communicate with each other. The secure way to do this is to use *sharedUserID* mechanism to protect the data and interface from exposing to other apps. In my empirical analysis, I find 560 app pairs are developed by the same developer without using `sharedUserID` links. Instead, they use explicit intents to implement the communication interface, which is exposed to all other apps, leaving hijacking vulnerabilities.

4.4.4 Q4: Runtime of MR-Droid

Finally, I analyze the runtime performance of MR-Droid. Figure 4.5 depicts the time cost of the MapReduce pipeline (y axis) as the number of apps increases (x axis). Overall, the result shows that my approach is readily scalable for large-scale inter-app analysis. The running time of ICC node identification appears to dominate the total analysis cost, yet its growth is linear with the number of apps. In addition, given the sparse nature of the ICC graph (rarely does an app communicate to all apps), I manage to achieve near-linear complexity for edge identification and grouping ICCs. In total, It takes 25 hours to perform the complete analysis on 13 million ICC pairs for 12K apps.

Noticeably, grouping ICCs for all app pairs only took 44 minutes, rendering 0.0012 seconds per app pair. This speedup benefits from the reduced input size — a large portion of (unmatched) ICC links have been excluded in the previous phase. The load balancing design also contributes to speeding up the process. Currently, the cluster has 15 nodes. I anticipate that increasing the cluster size would further speed up the inter-app ICC analysis.

As a baseline comparison, I evaluated the performance of IccTA [90], a non-distributed inter-app ICC analysis system. IccTA needs to first combine two or more apps into one app and then perform ICC analytics. I evaluate IccTA with 57 randomly selected real world apps on a workstation (64GB RAM). It took IccTA over 200 hours to analyze all the apps. I estimate that processing 200 apps with IccTA would take about 18 thousand hours, making it impractical for analyzing market-scale apps.

Summary of Findings. The manual verification confirms the accuracy of my system. For app pairs at high-risk level, I obtain a 100% TP rate for the detection of collusion,

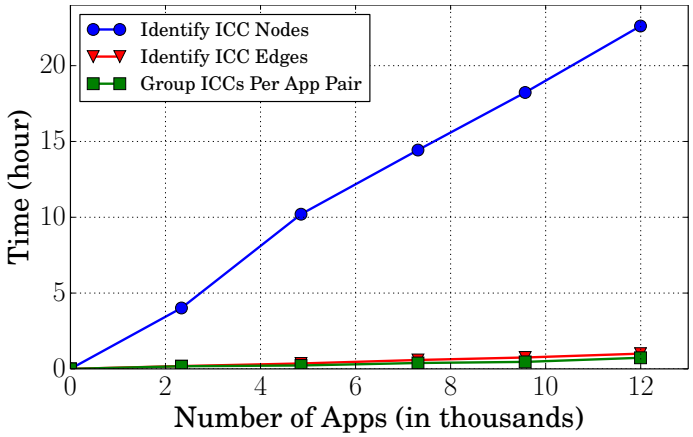


Figure 4.5: Analysis time of the three phases in my approach.

broadcast injection, activity- and service-launch based intent spoofing; I have a 90% TP rate for broadcast theft, activity- and service hijacking detection. On the other hand, app pairs at low-risk level indeed have substantially lower TP rate. This result indicates that the risk prioritization is effective. My empirical analysis reveals new types of app collusion and hijacking risks (e.g., leveraging rarely implicit intents for more stealthy collusion). My runtime experiments demonstrate that MapReduce pipeline scales well to a large number of apps. Analyzing 11,996 apps and performing ICCs matching took less than 25 hours. More importantly, the runtime cost has a near-linear increase with respect to the number of apps.

4.5 Security Recommendations

Based on my empirical study and manual verification, I make the following recommendations for app developers to reduce potential security risks.

- When carrying out sensitive operations, developers are encouraged to use *explicit Intents* instead of implicit ones. For example, to access Facebook account, communicating to the Facebook app via explicit Intents is preferred over a `https` URL as an implicit Intent to the Facebook webpage.
- When possible, it is recommended to integrate all necessary functionalities into a single app. If the developer has to develop multiple apps that communicate with each other, it is encouraged to use the safe communication via *sharedUserID* to restrict other apps accessing the interfaces.
- Developers are encouraged to use customized actions and to enforce data and permission restrictions. These customized configurations (compared to using the default) will greatly reduce the chance to accidentally communicate with other apps, hence reducing risks.

- It is recommended to avoid automatically generating apps using tools or services. Developers should be aware that third-party libraries could be vulnerable or leak user's sensitive information.

4.6 Discussion

The risk analysis is primarily based on Intent attributes, while dismissing data flows that involve the Intents. It is possible that this approach misses true vulnerabilities. Alternatively, one can perform more in-depth and fine-grained data-flow analysis [19, 67]. However, they conflict with efficiency and scalability, and thus less practical for market-wide app analyses. Compared to static analysis, dynamic approaches, either single app analyses [62] or inter-app analyses [70] are typically more precise. The issue is that dynamic analysis is much slower, and it often misses true threats that are not triggered during the analyzed executions.

My current analysis focuses on app pairs. It is possible for attacks to span across three or more apps. One way to generalize my approach to cluster app pairs into app groups. One needs to efficiently parallel the clustering process.

Alternative Implementations. The ICC mapping algorithm is designed under the universal MapReduce programming model. Any platforms that are built on the MapReduce programming model can implement my algorithm. For example, my system can easily fit in Spark [9] with "flatMap" and "groupByKey" functions for faster in-memory processing. However, it will require the cluster to have very large RAMs to batch process all the apps in memory.

One way to further speed up the system is to use GPU (CUDA). In practice, GPU (CUDA) is for computation only and orthogonal to MapReduce. GPU alone is not enough for large-scale data processing, but could be integrated with MapReduce to achieve better performance and scalability [129].

My system is not very suitable for MPI clusters. MPI is network-bounded when processing large-scale dataset. Developers have to work on the parallelization and data distribution themselves. In addition, MPI lacks the fault tolerance feature, which is essential for large-scale long-time processing tasks.

Chapter 5

Measure Mobile Deep Link Risks

In this chapter, I present the first large-scale measurement on the current ecosystem of mobile deep links [95]. The goal is to detect and measure link hijacking vulnerabilities across the web and mobile apps, and understand the effectiveness of new linking mechanisms in battling hijacking attacks.

Recently, two new deep link mechanisms were proposed to address the security risks in scheme URLs: App link and Intent URL. While most existing works focus on vulnerabilities in scheme URLs [42, 53, 138], little is known about how widely App links and Intent URLs are adopted, and how effective they are in mitigating the threats in practice.

I perform extensive measurements on a large collection of mobile apps and websites. To measure the adoption of different mobile deep links, I collect two snapshots of 160,000+ most popular Android apps from Google Play in 2014 and 2016, and crawled 1 million web pages (using a dynamic crawler) from Alexa top domains. I primarily focus on Android for its significant market share (87%) [79] and availability of apps. I also perform a subset of analysis on iOS deep links. At the high-level, my method is to extract the link registration entries (URIs) from apps, and then measure their empirical usage on websites. To detect hijacking attacks, I group apps that register the same URIs as link collision groups. I find that not all link collisions are malicious — certain links are expected to be shared such as links for common functionality (*e.g.*, "`tel`") or third-party libraries (*e.g.*, "`zxing`"). I develop methods to identify malicious hijacking attempts.

Findings. My study has four surprising findings, which lead to one overall conclusion: the newly introduced deep link solutions not only fail to improve security, but significantly increased hijacking risks for users.

First, App links' verification mechanism fails in practice. Surprisingly, among 8,878 Android apps with App links, only 194 (2.2%) correctly implemented link verification. The reasons are a combination of the lack of motivation from app developers and various developer mistakes.

I confirm a subset of mistakes in iOS App links too: 1925 out of 12,570 (15%) failed the verification due to server misconfigurations, including popular apps such as Airbnb.

Second, I uncover a new vulnerability in App links, which allows malicious apps to *stealthily* intercept HTTP/HTTPS URLs in the browser. The root cause is that Android grants excessive permissions to unverified App links through the preference setting. For an unverified App link, Android by default will prompt users to choose between the app and the browser. To disable promoting, users may set a "preference" to always use the app for this link. This preference is overly permissive, since it not only disables prompting for the current link, but all other unverified links registered by the app. A malicious app, once received preference, can hijack any sensitive HTTP/HTTPS URLs (*e.g.*, to a bank website) without alerting users. I validate this vulnerability in the latest Android 7.1.1.

Third, I detect more malicious hijacking attacks on App links (1593 apps) than scheme URLs (893 apps). Case studies show that popular websites (*e.g.*, "`google.com`") and apps (*e.g.*, Facebook) are common targets for traffic hijacking. In addition, I identify suspicious apps that act as man-in-the-middle between websites and the original app to record sensitive URLs and the parameters (*e.g.*, "`https://paypal.com`").

Finally, Intent URLs have very limited impact in mitigating hijacking risks due to the low adoption rate among websites. Only 452 websites out of the Alexa top 1 million contain Intent URLs (0.05%), which is a much lower ratio than that of App links (48.0%) and scheme URLs (19.7%). Meanwhile, among these websites, App links drastically increase the number of links that have hijacking risks compared to existing vulnerable scheme URLs

To the best of my knowledge, my study is the first empirical measurement on the ecosystem of mobile deep links across web and apps. I find the new linking methods not only failed to deliver the security benefits as designed, but significantly worsen the situation. There is a clear mismatch between the security design and practical implementations due to the lack of incentives of developers, developer mistakes, and inherent vulnerabilities in the link mechanism. Moving forward, I propose a list of suggestions to mitigate the threat. I have reported the over-permission vulnerability to the Google Android team. The detailed plan for further notification and risk mitigation is described in §5.7.

5.1 Background and Research Goals

Mobile deep links are URIs that point to specific locations within mobile apps. Through deep links, websites can initiate useful interactions with apps, which is instrumental to many key user experiences, for example, opening apps, sharing and bookmarking in-app pages [131], and searching in-app content using search engines [3]. In the following, I briefly introduce how deep links work and the related security vulnerabilities. Then I describe my research goals and methodology.

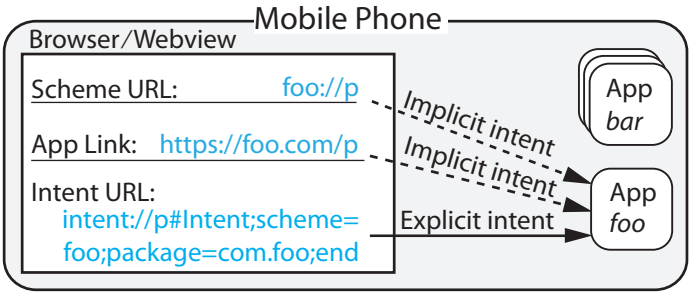


Figure 5.1: Three types of mobile deep links: Scheme URL, App Link and Intent URL.

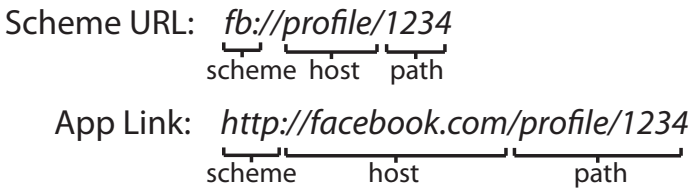


Figure 5.2: URI syntax for Scheme URLs and App links.

5.1.1 Mobile Deep Links

To understand how deep links work, I first introduce inter-app communications on Android. An Android app is essentially a package of software *components*. One app's components can communicate with another app's components through *Intent*, a messaging object characterized "action", "category" and "data". By sending an intent, one app can communicate with the other app's front-end *Activities*, or background *Services*, *Content Providers* and *Broadcast Receivers*.

Mobile deep links trigger a particular type of intent to enable communications between the web and mobile apps. As shown in Figure 5.1, after users click on a link in the browser (or in-app webview), the browser sends an intent to invoke the corresponding component in the target app. Unlike app-to-app communication, mobile deep link can only launch front-end Activity in the app.

Mobile deep links work in two simple steps: 1) Registration: an app "`foo`" should first register its URIs ("`foo://`" or "`https://foo.com`") to the mobile OS during installation. The URIs are declared in the in the "data" field of *intent filters*. 2) Addressing: when "`foo://`" is clicked, mobile OS will search all the intent filters for a potential match. Since the link matches the URI of app "`foo`", mobile OS will launch this app.

5.1.2 Security Risks of Deep Linking

Hijacking Risk in Scheme URL. Scheme URL is the first generation of mobile deep

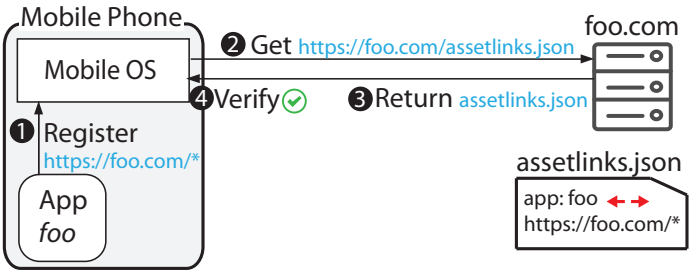


Figure 5.3: App link verification process.

links, and is the least secure one. It was introduced since Android 1.0 [7] and iOS 2.0 [8] in 2008. Figure 5.2 shows the syntax of a scheme URL. App developers can customize any schemes and URIs for their app without restriction.

Prior research has pointed out key security risks in scheme URLs [53, 138], given that any app can register other apps' schemes. For example, apps other than Facebook can also register "`fb://`". When a deep link is clicked, it triggers an "implicit intent" to open any app with a matched URI. This allows a malicious app to hijack the request to the Facebook app to launch itself, either for phishing (*e.g.*, displaying a fake Facebook login box), or stealing sensitive data in the request [53].

With an awareness of this risk, Android lets users be the security guard. When multiple apps declare the same URI, users will be prompted (with a dialog box) to select/confirm their intended app. However, if the malicious app is installed but the victim app is not, the malicious app will automatically skip the prompting and hijack the link without user knowledge. Even when both apps are installed, the malicious app may trick users to set itself as the "preference" and disable prompting. Historically speaking, relying on end-users as the sole security defense is risky since users often fail to perceive the nature of an attack, leading to bad decisions [16, 59, 137].

Solution1: App Link. App Link was introduced recently in October 2015 to Android 6.0 [5] as a more secure version of deep links. App link is designed to prevent hijacking with two mechanisms. First, the authentic app can build an association with the corresponding website, which allows the mobile OS to open the App link exclusively using the authentic app. Second, App link no longer allows developers to custom their own schemes, but exclusively uses the `http` or `https` scheme.

Figure 5.3 shows the App link association process. Suppose app "*foo*" wants to register "`http://foo.com/*`". Mobile OS will contact the server at "`foo.com`" for verification. The app's developer needs to set up an association file "assetlinks.json" beforehand under the root directory ("/.well-known/") of the `foo.com` server. This file must be hosted on an HTTPS server. If the file contains an entry that certifies that app "*foo*" is associated with the link "`http://foo.com/*`", the mobile OS will confirm the association. The association file contains a field called "`sha256_cert_fingerprints`", which is the SHA256 fingerprint of

the associated app's signing certificate. The mobile OS is able to verify the fingerprint and prevent hijacking because only the authentic app has the corresponding signing certificate. Suppose a malicious app "`bar`" also wants to register "`http://foo.com/*`", the verification will fail, assuming the attacker cannot access the root of `foo.com` server to modify the association file and the fingerprint.

The iOS version of App links is called universal link, introduced at iOS 9.0 [10], which has the same verification process. The association file for iOS is "apple- app-site-association". However, iOS and Android have different policies to handle *failed verifications*. iOS prohibits opening unverified universal links in apps. Android, however, leaves the decision to users: if an unverified link is clicked, Android prompts users to choose if they want to open the link in the app or browser.

Solution 2: Intent URL. Intent URL is introduced in 2013 and only works on Android [6]. Intent URLs prevent hijacking by changing how the deep link is called on the website. As shown in Figure 5.1, instead of calling "`foo://p`", Intent URL is structured as "`intent://p/#Intent;scheme=foo;package=com.foo;end`" where the package name of the target app is explicitly specified. Package name is a unique identifier for an Android app. Clicking an intent URL will launch an "explicit intent" to open the specified app.

Compared to scheme URLs and App links, Intent URL does not need special URI registration on the app. Intent URL can invoke the same interfaces defined by the URIs of scheme URLs or App links, as well as other exposed components [6].

5.1.3 Research Questions

While the hijacking risk of scheme URLs has been reported by existing research [42, 53, 138], little is known about how prevalently this risk exists among apps, and how effective the new mechanisms (App links and Intent URLs) are in reducing this risk in practice. I hypothesize that upgrading from scheme URL to App link/Intent URL is a non-trivial task, considering that scheme URLs may already have significant footprints on the web. Mobile platforms may be able to enforce changes to apps through OS updates, but their influence on the web is likely less significant. I conduct the first large-scale measurement on the mobile deep link ecosystem to understand the adoption of different linking methods and their effectiveness in battling hijacking threats.

Threat Model. My study focuses on *link hijacking threat* since this is the security issue that App Links and Intent URLs aim to address. Link hijacking happens when a malicious app registers the URI that belongs to the victim app. If mobile OS redirects the user to the malicious app, it can lead to phishing (*e.g.*, the malicious app displays forged UI to lure user passwords) or data leakage (*e.g.*, the deep link may carry sensitive data in the URL

Link Type	Conditions			Prompt User?
	> 1 Apps	Set As Preference	Link Verified	
Scheme URL	✓	✗	/	✓
	✓	✓	/	✗
	✗	✗	/	✗
	✗	✓	/	✗
App Link*	/	✗	✗	✓
	/	✓	✗	✗
	/	✗	✓	✗
	/	✓	✓	✗
Intent URL	/	/	/	✗

Table 5.1: Conditions for whether users will be prompted after clicking a deep link on Android. *App Links always have at least one matched app, the mobile browser.

parameters such as PII and session IDs) [53]. In this threat model, mobile OS and browser (or webview) are not the target of the attack, and I assume they are not malicious.

The Role of Users. Users also play a role in this threat model. After clicking a deep link, a user may be prompted by a dialog box to confirm the destination app. As shown in Table 5.1, prompting can be skipped in many cases. For *scheme URLs*, a malicious app can skip prompting if the victim app is not installed, or by tricking users to set the malicious app as the "preference". *App link* can skip prompting if the link has been verified. Otherwise, users will be prompted to choose between the browser and the app. *Intent URLs* will not prompt users at all since the target app is explicitly specified.

My Goals. My study seeks to answer key questions regarding how mobile deep links are implemented in the wild and their security impact. I ask three sets of questions. *First*, how prevalently are different deep links adopted among apps over time? Are App links and Intent URLs implemented properly as designed? *Second*, how many apps are still vulnerable to hijacking attacks? How many vulnerable apps are exploited by other real-world apps? *Third*, how widely are hijacked links distributed among websites? How much do App links and Intent URLs contribute to mitigating such links?

To answer these questions, I first describe data collection (§5.2), and measure the adoption of App links and scheme URLs among apps (§5.3). I perform extensive security analysis to understand how effective App links can prevent hijacking (§5.4), and then describe the method to detect hijacking attacks among apps (§5.5). Finally, I move to the web to measure the usage of Intent URLs, and the prevalence of hijacked links (§5.6). In §5.7, I summarize key implications and discuss possible solutions

Dataset	Total Apps	Apps accept Scheme URLs	Apps accept App Links	Apps accept either Links	Unique Schemes	Unique Web Hosts
App2014	164,322	10,565 (6.4%)	4,545 (2.8%)	12,428 (7.6%)	8,845	6,471
App2016	164,963	20,257 (12.3%)	8,878 (5.4%)	23,830 (14.5%)	18,839	18,561

Table 5.2: Two snapshots of Android apps collected in 2014 and 2016. 115,399 apps appear in the both datasets; 48,923 apps in App2014 are no longer listed on the market in 2016; App2016 has 49,564 new apps.

5.2 Datasets

I collect data from both mobile apps and websites, including two snapshots of 160,000+ most popular Android apps in 2014 and 2016, and web pages from Alexa top 1 million domains.

Mobile Apps. To examine deep link registration, I crawled two snapshots of mobile apps from Google Play. The first snapshot *App2014* contains 164,322 most popular free apps from 25 categories in December 2014 (crawled with an Android 4.0.1 client). In August 2016, I crawled a second snapshot of top 160,000 free apps using an Android 6.0.1 client. I find that 48,923 apps in App2014 are no longer listed on the market in 2016. 4,963 apps in 2014 snapshot already fell out of the top 160K list in 2016. To match the two datasets, I also crawled these 4,963 apps in 2016, forming an *App2016* dataset of 164,963 apps. The two snapshots have 115,399 overlapping apps. For each app in *App2016*, I also obtained the developer information, downloading count, review count and rating.

My app dataset is biased towards popular apps among the 2.2 million apps in Google Play [128]. Since these popular apps have more downloads, potential vulnerabilities could affect more users. The result can serve as a lower bound of empirical risks.

Alexa Top 1 Million Websites. To understand deep link usage on the web, I crawled Alexa top 1 million domains [1] in October 2016. I simulate using an Android browser (Android 6.0.1, Chrome/41/0/2272.96) to visit these web domains and load both static HTML page (index page) and the dynamic content from JavaScript. This is done using modified OpenWPM [63], a headless browser-based crawler. For each visit, the crawler loads the web page and waits for 300 seconds allowing the page to load the dynamic content, or perform the redirection. I store the final URL and HTML content. This crawling is also biased towards popular websites, assuming that deep links on these sites are more likely to be encountered by users. I refer this dataset as *Alexa1M*.

5.3 Deep Link Registration by Apps

In this section, I start by analyzing mobile apps to understand deep link registration and adoption. In order to receive deep link requests, an app needs to register its URIs to mobile OS during installation. My analysis in this section focuses on Scheme URLs and App links. For Intent URLs, as described in §5.1, developers do not need special registrations in the app. Instead, it is up to the *websites* to decide whether to use Intent URLs or scheme URLs to launch the app. I will examine the adoption Intent URLs later by analyzing web pages (§5.6).

I provide an overview of deep link adoption by analyzing 1) how widely the scheme URLs are adopted among apps, and 2) whether App links are in the process of replacing scheme URLs for a better security.

5.3.1 Extracting URI Registration Entries

Android apps register their URIs in the manifest file (`AndroidManifest.xml`). Both Scheme URLs and App Links are declared in Intent filters as a set of matching rules, which can either be actual links (`fb://login/`) or a wild card (`fb://profile/*`). Since there is no way to exhaustively obtain all links behind a wild card, I treat each matching rule as a registration entry. Given a manifest file, I extract deep link entries in three steps:

- **Step1: Detecting Open Interfaces.** I capture all the Activity intent filters whose "category" field contains both *BROWSABLE* and *DEFAULT*. This returns all the components that are reachable from the web.
- **Step2: Extracting App Link.** Among intent filters in Step 1, I capture those whose "action" contains *VIEW*. This returns intent filters with either App Links or Scheme URLs in their "data" fields[1]. I extract App Link URIs as those with http/https scheme. Note that App Link intent filters have a special field called *autoVerify*. If its value is TRUE, then mobile OS will perform verification on the App link.
- **Step3: Extracting Scheme URL.** All the non-http/https URIs from Step2 are Scheme URLs.

I apply the above method to the dataset and the result is summarized in Table 5.2. Among the 160K apps in *App2016*, I find that 20.3K apps adopted scheme URLs and 8.9K apps adopted App links. Note that for the apps in *App2014* (Android 4.0 or lower), App Link had not been introduced to Android yet. I find that 4,545 apps in *App2014* registered http/https URIs, which are essentially scheme URLs with "`http`" or "`https`" as the scheme. For consistency, I still call these http/https links as App links, but link verification is not supported for these apps.

[1]The rest intent filters whose "action" is not *VIEW* can still be triggered by Intent URLs.

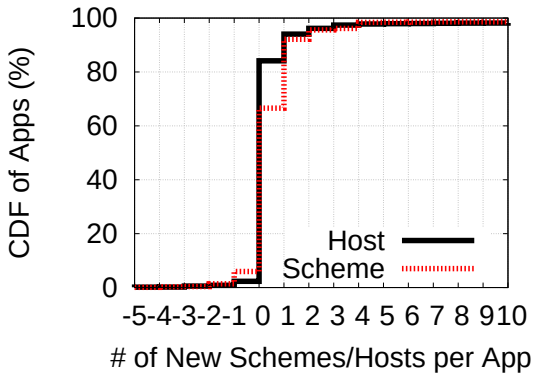


Figure 5.4: # of new schemes and app link hosts per app between 2014 and 2016.

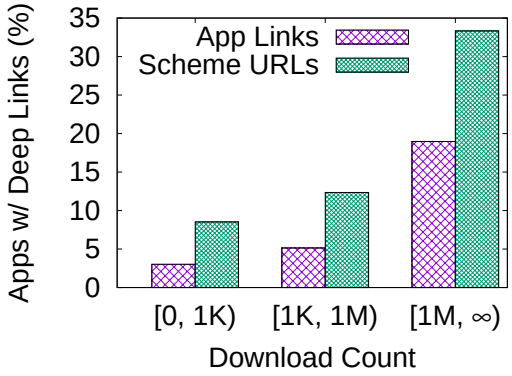


Figure 5.5: % of apps w/deep links; apps are divided by download count.

5.3.2 Scheme URL vs. App Link

Next, I compare the adoption for Scheme URLs and App links across time, app categories and app popularity. I seek to understand if the new App links are on the way of replacing Scheme URLs.

Adoption over Time. As shown in Table 5.2, there are significantly more apps that started to adopt deep links from 2014 to 2016 (about 100% growth). However, the growth rate is almost the same for App links and Scheme URLs. There are still 2-3 times more apps using scheme URLs than those with App links. Apps links are far from replacing scheme URLs.

Figure 5.4 specifically looks at apps in both snapshots. I select those that adopted either type of deep links in either snapshot (13,538 apps), and compute the differences in their number of schemes/hosts between 2014 and 2016. I find that the majority of apps (over 96.2%) either added more deep links or remained the same. Almost no apps remove or replace scheme URLs with App links. The conclusion is the same when I compare the number of URI rules (omitted for brevity). This suggests that scheme URLs are still heavily used, exposing users to potential hijacking threat.

App Popularity. I find that deep links are more commonly used by popular apps (based on download count). In Figure 5.5, I divide apps in 2016 into three buckets based on their download count: $[0, 1K)$, $[1K, 1M)$, $[1M, \infty)$. Each has 20,654, 127,323 and 5,223 apps respectively. Then I calculate the percentage of apps that adopted deep links in each bucket. I observe that 33% of the 5,223 most popular apps adopted scheme URL, and the adoption rate goes down to 8% for apps with <1K downloads. The trend is similar for App links. In addition, I find that apps *with* deep links have averagely 4 million downloads per app, which is orders of magnitude higher than apps *without* deep links (125K downloads per app). As deep links are associated with popular apps, potential vulnerabilities can affect many users.

App Categories. Among the 25 app categories, I find that the following categories have the highest deep link adoption rate: SHOPPING (25.5%), SOCIAL (23.4%), LIFESTYLE (21.0%), NEWS_AND_MAGAZINES (20.5%) and TRAVEL_AND_LOCAL (20.2%). These apps are content-heavy and often handle user personally identifiable information (*e.g.*, social network app) and financial data (*e.g.*, shopping app). Link hijacking targeting these apps could have practical consequences.

5.4 Security Analysis of App Links

The result shows App links are still not as popular as scheme URLs. Then for apps that adopted App links, are they truly secure against link hijacking? As discussed in §5.1.2, App link is designed to prevent hijacking through a link verification process. If a user clicks on an unverified App link, the mobile OS will prompt users to choose whether she would like to open the link in the browser or using the app. In the following, I empirically analyze the security properties of App links in two aspects. First, I measure how likely app developers make mistakes when deploying App link verification. Second, I discuss a new vulnerability I discovered which allows malicious apps to skip user prompting when unverified App links are clicked. Malicious apps can exploit this to stealthily hijack arbitrary HTTP/HTTPS URLs in the mobile browser without user knowledge.

5.4.1 App Link Verification

I start by examining whether *link verification* truly protects apps from hijacking attacks. Since App link has not been introduced for *App2014*, all the http/https links in 2014 were unverified. In the following, I focus on apps in *App2016*. In total, there are 8,878 apps that registered App links, involving 18,561 unique web domains. I crawled two snapshots of the association files for each domain in January and May of 2017 respectively. I use the January snapshot to discuss the key findings, and then use the May snapshot to check if the identified problems have been fixed.

Failed Verifications. As of Januray 2017, I find a surprisingly low ratio of verified App links. Among 8,878 apps that registered App Links, only 194 apps successfully passed the verification (2%). More specifically, only 415 apps (4.7%) set the "*autoVerify*" field as TRUE, which triggers the verification process during app installation. This means the vast majority of apps (8,463, 95.3%) did not even start the verification process. Interestingly, 434 apps actually have the association file ready on their web servers, but the developers seem to forget to configure the apps to turn on the verification.

Even for apps that turned on the verification, only 194 out of 415 can successfully complete the process as of January 2017. Table 5.3 shows the common mistakes of the failed apps

(one app can have multiple mistakes). More specifically, 26 apps incorrectly set the App link (*e.g.*, with a wildcard in the domain name), which is impossible for mobile OS to connect to. On the server-side, 177 apps turned on the verification, but the destination domain did not host the association file; 11 apps host the file under a HTTP server instead of the required HTTPS server; 10 apps' files are in invalid JSON format; 60 apps' association files do not contain the App link (or the app) to be verified. Note that for these failed apps, I don't distinguish whether they are malicious apps attempting to verify with a domain they don't own, or simply mistakes by legitimate developers.

I confirm all these mistakes lead to failed verifications by installing and testing related apps on a physical phone. I observe many of these mistakes are made by popular apps from big companies. For example, "`com.amazon.mp3`" is Amazon's official music app, which claims to be associated with "`amazon.com`". However, the association file under `amazon.com` did not certify this app. I tested the app on my phone, which indeed failed the verification.

In May 2017, I checked all the apps again and found that most of the identified problems remained unfixed. Moreover, some apps introduced new mistakes: there are 8 more apps with an invalid association files in May compared to that of January. Manual examination shows that new mistakes were introduced when the developers updated the association files.

Date	**Apps w/ App Links**	**Apps Turned Verif. On**	**Apps Verified**	**Apps with Failed Verifications***					
				App Misconfig.	Host w/o Assoc. F.	Host w/ HTTP	Wrong Path	Host Invalid F.	Host Assoc. Other apps
Jan.17	8,878	415	194	26	177	11	0	10	60
May.17	8,878	415	192	26	171	8	0	18	57

Table 5.3: App Link verification statistics and common mistakes (App2016) based on data from January 2017 and May 2017. *One app can make multiple mistakes.

Misconfigurations for iOS and Android. To show that App links verification can be easily misconfigured, I put together 1,012,844 web domains to scan their association files. These 1,012,844 domains is a union of Alexa top 1 million domains and the 18,561 domains extracted from the apps. I scan the association files for both Android and iOS.

As of January 2017, 12,570 domains (out 1 million) has iOS association files and only 1,833 domains have Android association files (Table 5.4). It is unlikely that there are 10x more iOS-exclusive apps. A more plausible explanation is iOS developers are more motivated to perform link verification, since iOS prohibits opening unverified HTTP/HTTPS links in apps. In contrary, Android leaves the decision to users by prompting users to choose between using apps or a browser.

I find iOS apps also have significant misconfigurations. This analysis only covers a subset of possible mistakes compared to Table 5.3, but still returns a large number. As of January 2017, 1817 domains (14%) are hosting the association file under HTTP, and there are additional 108 domains (1%) with invalid JSON files. One example is the Airbnb's iOS app. The app tries to associate with "`airbnb.com.gt`", which only hosts the association file under an

Type	Date	Hosts w/ Assoc. F.	Under HTTP	Wrong Path	Invalid File
iOS	Jan.17	12,570	1,817 (14%)	0 (0%)	108 (1%)
	May.17	13,541	1,820 (13%)	0 (0%)	113 (.8%)
Android	Jan.17	1,833	330 (18%)	4 (.2%)	81 (4%)
	May.17	2,779	474 (17%)	0 (0%)	118 (4%)

Table 5.4: Association files for iOS and Android obtained after scanning 1,012,844 domains.

HTTP server. This means users will not be able to open this link in the Airbnb app.

In May 2017, I scanned these domains again. I observed 7.7% of increase of hosts with association files for iOS and 51.6% increase for Android. However, the number of misconfigured association files also increased.

5.4.2 Over-Permission Vulnerability

In addition to verification failures, I identify a new vulnerability in the setting preferences for App links. Recall that unverified App links still have one last security defense — the end user. Android OS prompts users when unverified App links are clicked, and users can choose between a browser and the matched app. I describe an *over-permission vulnerability* that allows malicious apps to skip prompting for stealthy hijacking.

Over-Permission through Preference Setting. User prompting is there for better security, but prompting users too much can hurt usability. Android's solution is to take a middle ground using "preference" setting. When an App link is clicked, users can set "preference" for always opening the link in the native app without prompting again.

I find that the preference setting gives excessive permissions to this app. Specifically, the preference not only disables the prompting for the current link that the user sees, but all other (unverified) HTTP/HTTPS links that this app registered. For example, if the user sets preference for "`https://bar.com`", all the links with "`https://`" in this app receive the permission. Exploiting this vulnerability allows malicious apps to hijack any HTTP/HTTPS URLS without alerting users.

Proof-of-Concept Attack. Suppose "`bar`" is a malicious app that registered both "`https://bar.com`" and "`https://bank.com/transfer/*`". The user sets preference for using "`bar`" to open the link "`https://bar.com`", which is considered as a normal action. Then the permission also applies to "`https://bank.com/transfer/*`" without user knowledge.

Later, suppose this user visits her bank's website in a mobile browser, and transfers money through an HTTPS request "`https://bank.com/transfer?cookie=1&amount=1000&recipient=tom`". Because of the preference setting, this request will auto-

matically trigger `bar` without prompting the user. The browser wraps up this URL and the parameters in plaintext to create an Intent, and hand it over to the app `bar`. `bar` can then change the recipient and use the cookie to transfer money to the attacker. In this example, the attacker sets the path of the URI as "`/transfer/*`" so that `bar` would only be triggered during money transfer. The app can make this even more stealth by quickly terminating itself after the hijacking, and bouncing the user back to the bank website in the browser.

I validate this vulnerability in both Android 6.0.1 and 7.1.1 (the latest version). I implement the proof-of-concept attack by writing a malicious Android app to hijack the author's own blog website (instead of an actual bank). The attack is successful: the malicious app hijacked the plaintext cookie in the URL parameter, and quickly bounced the user back to original page in the browser. The bouncing is barely noticeable by users.

Discussion. Fundamentally, this vulnerability is caused by the excessive permission to unverified App links. When setting preferences, the permission is not applied to the *link-level*, but to the *scheme-level*. I suspect that the preference system of App links is directly inherent from scheme URLs. For scheme URLs, the preference is also set to the scheme level which makes more sense (*e.g.*, allowing Facebook app to open all "`fb://`"). However, for App links, scheme-level permission means attackers can hijack any HTTPS links. I have filed a bug report to the Android team in 02/2017.

To successfully exploit this vulnerability, a malicious app needs to trick users to set the preference (*e.g.*, using benign functionalities). For example, an attacker may design a recipe app that allows users to open recipe web links in the app for an easy display and sharing. This recipe app can ask users to set the preference for opening recipe links but secretly registers an online bank's App links to receive the same preference. I have filed a bug report through Google's Vulnerability Reward Program (VRP) in February 2017. I am currently working with the VRP team to mitigate the threat.

iOS has a similar preference setting, but not vulnerable to this over-permission attack. In iOS, if the user sets preference for one app to open an HTTPS link. The permission goes to all the HTTPS links that the app *has successfully verified*. The Android vulnerability is caused by the fact that permission goes to unverified links.

5.4.3 Summary of Vulnerable Apps.

Thus far, my analysis shows that most apps are still vulnerable to link hijacking. First, scheme URLs are still heavily used among apps. Second, for apps that adopt App links, only 2% can pass the link verification. The over-permission vulnerability described above makes the situation even worse. In 2016, out of all 23,830 apps that adopted deep links, 23,636 apps either use scheme URLs or unverified App links. These are candidates of potential hijacking attacks.

5.5 Link Hijacking

While many apps are vulnerable in theory, the real question is how many vulnerable apps are exploited in practice? For a given app, how likely would other apps register the same URIs (*a.k.a.*, link collision)? Do link collisions always have a malicious intention? If not, how can I classify malicious hijacking from benign collisions?

To answer these questions, I first measure how likely it is for different apps to register the same URIs. My analysis reveals key categories of link collisions, and I develop a systematic procedure to label all of them. This allows us to focus on the highly suspicious groups that are involved in malicious hijacking. Finally, I present more in-depth case studies to understand the risk of typical attacks.

5.5.1 Characterizing Link Collision

Links collision happens when two or more apps register the same deep link URIs. When the link is clicked, it is possible for mobile OS to direct users to the wrong app. Note that simply matching "scheme" or app link "host" is not sufficient. For example, "`myapp://a/1`" and "`myapp://a/2`" do not conflict with each other since they use different "paths" in the URI. To this end, I define two apps have link collision only if there is at least one link that is opened by both apps.

Prevalence of Link Collisions. To identify link collision, I first group apps based on the scheme (scheme URL) or web host (App links). Figure 5.6 and Figure 5.7 shows the number of apps that each scheme/host is associated with. About 95% of schemes are exclusively registered by one single app. The percentage is slightly lower for App links (76%–82%). Then for each group, I filter out apps that have no conflicting URIs with any other apps in the group, and produce apps with link collisions. Within *App2014*, I identify 394 schemes, 1,547 web hosts from 5,615 apps involved in link collisions. The corresponding numbers for 2016 are higher: 697 schemes and 3,272 web hosts from 8,961 apps.

The result is a lower bound of actual collisions, biased towards popular apps. Schemes/hosts that are currently mapped to a single app might still have collisions with apps outside of the dataset. For the rest of my analysis, I focus on the more recent 2016 dataset.

Categorizing Link Collisions. I find that not all collisions have malicious intention. After manually analyzing these schemes and hosts, I categorize collisions into 3 types. Table 5.5 shows the top 10 mostly registered schemes/hosts and their labels.

- **Functional scheme (F)** is reserved for a common functionality, instead of a particular app. "`file`" is registered by 1,278 apps that can open files. "`geo`" is registered by 238 apps that can handle GPS coordinates. These schemes are expected to be registered by

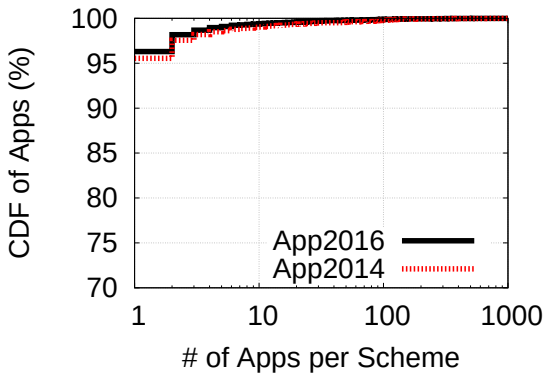


Figure 5.6: # of Collision apps per scheme.

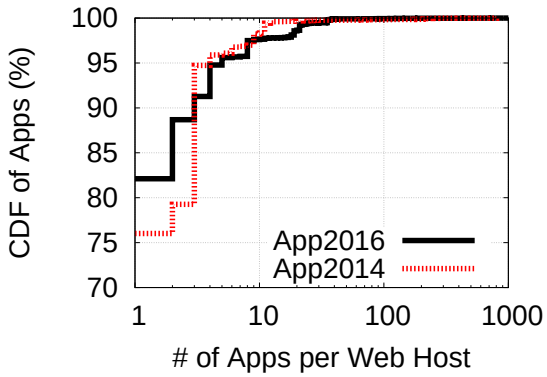


Figure 5.7: # of Collision apps per web host.

multiple apps. IANA [21] maintains a list of URI schemes, most of which are functional ones. This collision type does not apply to App links.

- **Per-app scheme/host (P)** is designated to an individual app. "`maps.google.com`" is to open Google Maps (but registered by 186 other apps) and "`fb`" is supposed to open Facebook app (but registered by 4 other apps). Collisions on per-app schemes/hosts are often malicious, with the exception if all apps are from the same developer.
- **Third-party scheme/host (T)** is used by third-party libraries, which often leads to (unintentional) link collision. "`x-oauthflow-twitter`" is a callback URL for Twitter OAuth. Twitter suggests developers defining their own callback URL, but many developers copy-paste this scheme from an online tutorial (unintentional collision). Another third-party scheme is"`feedproxy.google.com`", which is from a third-party RSS aggregator. Apps use this service to redirect user RSS requests to their apps (benign collision).

Because of the "shared" nature, functional schemes or third-party schemes/hosts are expected to be used by multiple apps. Related link collisions are benign or unintentional. In contrary, per-app scheme/hosts are (expected to be) designated to each app, and thus link collision can indicate malicious hijacking attempts.

5.5.2 Detecting Malicious Hijacking

Next, I detect malicious hijacking by labeling *per-app* schemes/hosts. This is challenging since schemes and hosts are registered without much restriction—it is difficult to tell based on the name of the scheme/host. The high-level intuition is: 1) third-party schemes/hosts often have official documentations to teach developers how to use the library, which are searchable online; 2) functional schemes are well-documented in public URI standard. To these ends, I develop a filtering procedure to label per-app schemes/hosts. For any manual

Scheme	Apps	Web Host	Apps
file Ⓕ	1278	google.com Ⓟ	480
content Ⓕ	727	google.co.uk Ⓟ	441
oauth Ⓣ	520	zxing.appspot.com Ⓣ	410
x-oauthflow-twitter Ⓣ	369	maps.google.com Ⓟ	187
x-oauthflow-espn-twitter Ⓣ	359	beautygirlsinc.com Ⓟ	148
zxing Ⓣ	321	triposo.com Ⓟ	131
testshop Ⓣ	278	feeds.feedburner.com Ⓣ	126
shopgate-10006 Ⓣ	278	feeds2.feedburner.com Ⓣ	123
geo Ⓕ	238	feedproxy.google.com Ⓣ	112
tapatalk-byo Ⓣ	180	feedsproxy.google.com Ⓣ	110

Table 5.5: Top 10 schemes and app link hosts with link collisions in App2016. I manually label them into three types: Ⓕ= Functional, Ⓟ= Per-App, Ⓣ= Third-party

	Deep Links In Total	Link Collisions	After Pre-Processing	Functional	Third-party	Per-app
#Schemes (#Apps)	18,839 (20,257)	697 (7,432)	376 (6,350)	30 (2,135)	197 (3,972)	149 (893)
#Hosts (#Apps)	18,561 (8,878)	3,272 (2,868)	2,451 (2,083)	N/A	137 (999)	2,314 (1,593)

Table 5.6: Filtering and classification results for schemes and App link hosts (App2016).

labeling tasks, I have two authors perform the task independently, and a third person to resolve any disagreements.

Pre-Processing. I start with the 697 schemes and 3,272 hosts (8,961 apps) that have link collisions in *App2016*. I exclude schemes/hosts where all the collision apps are from the same developer. This leaves 376 schemes and 2,451 web hosts for further labeling.

Classifying Schemes. I label schemes in two steps. The results are shown in Table 5.6. First, I filter out functional schemes. IANA [21] lists 256 common URI schemes, among which there are a few per-apps scheme under "provisional" status (*e.g.*, "`spotify`"). I manually filtered them out and got 175 standard functional schemes. Matching this list with the dataset returns 30 functional schemes with link collisions. Then, to label third-party schemes, I manually search for their documentations or tutorials online. For certain third-party schemes, I also check the app code to be sure. In total, I identify 197 third-party schemes, and the rest 149 schemes are per-app schemes (also manually checked).

Figure 5.8 shows the number of collision apps for different schemes. Not surprisingly, per-app scheme has fewer collision apps than functional and third-party schemes.

Classifying App Link Hosts. This only requires labeling third-party hosts from per-app hosts. In total, there are 2,451 hosts after pre-processing. I observe that 1633 hosts are jointly registered by 5 apps, and 347 subdomains of "`google.com`" are registered by 2 apps.

All these hosts are not third-party hosts, which helps to trim down to 471 hosts for manual labeling. I follow the same intuition to label third-party web hosts by manually searching their official documentations. In total, I label 137 third-party hosts, and 2,314 per-app hosts. Figure 5.9 compare per-app hosts and third-party hosts on their number of collision apps, which are very similar.

Testing Automated Classification. Clearly manually labeling cannot scale. Now that I have obtained the labels, I briefly explore the feasibility of automated classification. As a feasibility test, I classify per-app schemes from third-party schemes using 10 features such as unique developers per scheme, and apps per scheme (feature list in Table 5.7). 5-fold cross-validation using SVM and Random Forests classifiers return an accuracy of 59% (SVM) and 62% (RF). If I only focus on schemes that have a higher-level of collisions (*e.g.*, > 4 developers), it returns a higher accuracy: 84% (SVM) and 75% (RF). The accuracy is not high enough for practical usage. Intuitively, there are not many restrictions on how developers register their URIs, and thus it is possible that the patterns of per-app schemes are not that strong.

Since fully automated classification is not yet feasible, I then explore useful heuristics to help app market admins to conduct collision auditing. I rank features based on the information gain, and identify top 3 features: average number of apps from the same developer (apDev), number of unique no-prefix components (npcNum) and number of unique components (ucNum). Regarding apDev, the intuition is that developers are likely to use a different per-app scheme for each of their apps, but would share the same third-party schemes (*e.g.*, `oauth`) for all their apps. A larger apDev of the collision link indicates a higher chance of being a third-party scheme. Moreover, third-party schemes are likely to use the same component name for different apps (*i.e.*, less unique), leading to smaller npcNum and ucNum.

Feature	Description
aNum	Total # of apps
uDev	# of developers
cNum	Total # of components
ucNum	# of unique components
utcNum	# of unique third-party components
npcNum	# of unique components name (no prefix)
tDev	# of developers with third-party components
apDev	Average # of apps per developer
tDevP	% of third-party developers
ucP	% of unique components

Table 5.7: Features used for scheme classification.

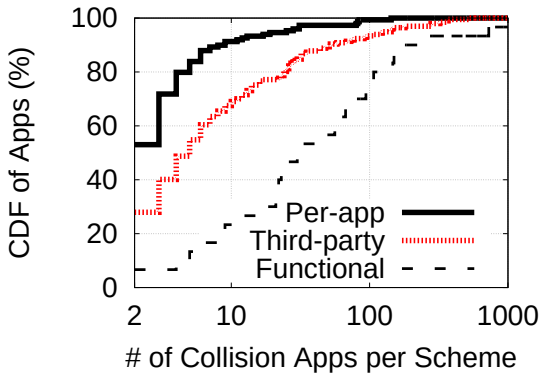


Figure 5.8: # of collision apps per scheme.

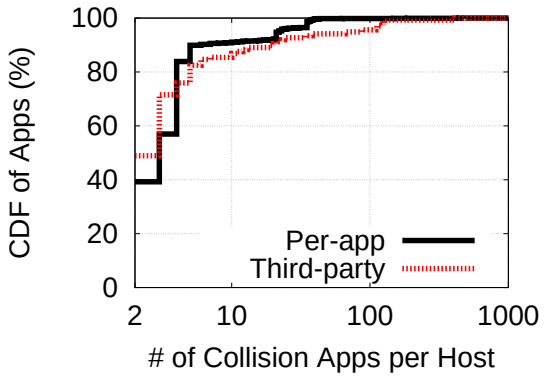


Figure 5.9: # of collision apps per host.

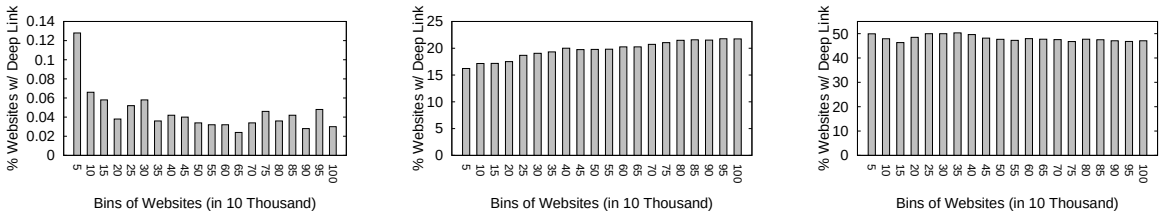
5.5.3 Hijacking Results and Case Studies

In total, I identify 149 per-app schemes and 2,314 per-app hosts that are involved in link collisions. The related apps (893 and 1,593 respectively) are either the attacker or victim in the hijacking attacks. To understand how per-app schemes and hosts are hijacked, I perform in-depth cases studies on a number of representative attacks.

Traffic Hijacking. I find that rogue apps often register popular websites' links (or popular apps' schemes) seeking to redirect user traffic to themselves. For example, "`google.com`" is registered by 480 apps from 305 non-Google developers. The scheme "`google.navigation`" from Google Maps is hijacked by 79 apps from 32 developers. The intuition is that popular sites and apps already have a significant number of links distributed to the web. Hijacking their links are likely to increase the attacker apps' chance of being invoked. I find many popular apps are among the hijacking targets (*e.g.*, Facebook, Airbnb, YouTube, Tumblr). Traffic hijacking is the most common attacks.

URL Redirector MITM. A number of hijackings are conducted by "URL Redirector" apps. When users click on an http/https link in the browser, these Redirector apps redirect users to the corresponding apps. Essentially, Redirector apps play the role of mobile OS in redirecting URLs, but their underlying mechanisms have several security implications. For example, URLLander (`com.chestnutcorp.android.urlander`) and AppRedirect (`com.nevoxo.tapatalk.redirect`) each has registered HTTPS links from 36 and 75 web domains respectively (unverified) and has over 10,000 installs. I suspect that users install Redirector apps because of the convenience, since these apps allow users to open the destination apps (without bouncing to the browser) even if the destination apps have not yet adopted App links. The redirection is hard coded without the consent of the destination apps or the originated websites.

In addition, URL redirector apps can act as man-in-the-middle (MITM). For example, URLLander registered "`https://www.paypal.com`" for redirection. When a user visits



(a) Intent URLs (b) Scheme URLs (c) App Links

Figure 5.10: Deep link distribution among Alexa top 1 million websites. Website domains are sorted and divided into 20 even-sized bins (50K sites per bin). I report the % of websites that contain deep links in each bin.

`paypal.com` using a browser (usually logged-in), the URL contains sensitive parameters including a SESSIONID. Once the user agrees to use URLLander for redirection, the URL and SESSIONID will be handed over to URLLander by the browser in plaintext. This MITM threat applies to all the popular websites that Redirector apps registered such as `facebook.com`, `instagram.com`, and `ebay.com`. Particularly for eBay, I find that the official eBay app explicitly does not register to open the link "`payments.ebay.com`", but this link was registered by Redirector apps. I analyze the code of AppRedirect and find it actually writes every single incoming URL and parameters in a log file. Redirection (and MITM) can be automated without prompting users by exploiting the over-permission vulnerability (see §5.4.2) — if the user once sets a preference for just one of those links.

Hijacking a Competitor's App. Many apps are competitors in the same business, and I find targeted hijacking cases between competing apps. For example, Careem (`com.careem.acma`) and QatarTaxi (`com.qatar.qatartaxi`) are two competing taxi booking apps in Dubai. Careem is more popular (5M+ downloads), which uses scheme "`careem`" for many functionalities such as booking a ride (from hotel websites) and adding credit card information. QatarTaxi (10K downloads) registers to receive all "`careem://*`" deep links. After code analysis, I find all these links redirect users to the QatarTaxi app's home screen, as an attempt to draw customers.

Bad Scheme Names. Hijackings are also caused by developers using easy-to-conflict scheme names. For example, Citi Bank's official app uses "`deeplink`" as its per-app scheme, which conflicts with 6 other apps. These apps are not malicious, but may cause confusions — a user was going to open the Citi Bank app, but a non-related app shows up (and vice versa). I detect 14 poorly named per-app schemes (*e.g.*, "`myapp`", "`app`").

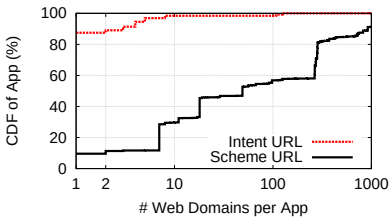


Figure 5.11: Number of websites that host deep links for each app.

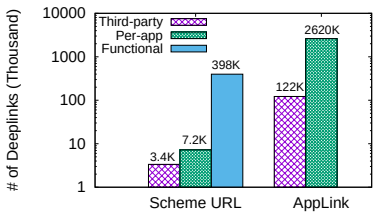


Figure 5.12: Different type of hijacked deep links in Alexa1M.

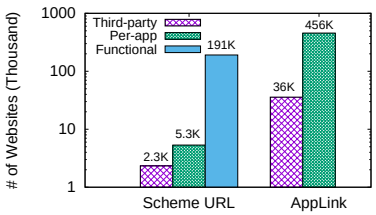


Figure 5.13: Webpages that contain hijacked deep links in Alexa1M.

Dataset	App Link (Webpage)	Scheme URL (Webpage)	Intent URL (Webpage)
Alexa1M	3.2M (480K)	431K (197K)	1,203 (452)

Table 5.8: Number of deep links (and webpages that contain deep links) in Alexa top 1 million web domains.

5.6 Mobile Deep Links on The Web

My analysis shows that hijacking risks still widely exist within apps. Now I move to the web-side to examine how mobile deep links are distributed on the web, and estimate the chance of users encountering hijacked links. In addition, I focus on *Intent URL* to examine its adoption and usage. I seek to estimate the impact of Intent URLs to mitigating hijacking threats.

In the following, I first measure the prevalence Intent URLs on the web, and compare with scheme URLs and App links. Then, I revisit the hijacked links detected in §5.5 and analyze their appearance on the web.

5.6.1 Intent URL Usage

Intent URL is a secure way of calling deep links from websites by specifying the target app's package name (unique identifier). In theory, Intent URL can be used to invoke existing app components defined by scheme URLs (and even App links) to prevent hijacking. The key question is how widely are Intent URLs adopted in practice. .

Intent URLs vs. Other Links I start by extracting mobile deep links from web pages in *Alexa1M* collected in §5.2. For App links and scheme URLs, I match all the hyperlinks in the HTML pages with the link registration entries extracted from apps. I admit that this method is conservative as I only include deep links registered by apps in the dataset. But the matching is necessary since not all the HTTP/HTTPS links or schemes on the web can invoke

apps. For Intent URLs, I identify them based on their special format ("`intent://*;end`"). The matching results are shown in Table 5.8.

The key observation is Intent URLs are rarely used. Out of 1 million web domains, only 452 (0.05%) contain Intent URLs in their index page. As a comparison, App links and Scheme URLs appear in 480K (48%) and 197K (19.7%) of these sites. For the total number of links, Intent URL is also orders of magnitude lower than other links (1,203 versus 3.2M and 431K). This extremely low adoption rate indicates that Intent URLs have little impact to mitigating hijacking risks in practice.

Challenges to Intent URL Adoption. Since Android still supports scheme URLs, it is possible that developers are not motivated to use Intent URLs to replace the still-functional scheme URLs. In addition, even if security-aware developers use Intent URLs on their own websites, it is difficult for them to upgrade scheme URLs that have been distributed to other websites.

As shown in Figure 5.10(a), Intent URLs are highly skewed towards to high-ranked websites. In contrary, Scheme URLs are more likely to appear in low-ranked domains (Figure 5.10(b)), and App links' distribution is relatively even (Figure 5.10(c)). A possible explanation is that popular websites are more security-aware.

Then I focus on apps, and examine how many websites that contain an app's deep links (Figure 5.11). I find that most apps have their Intent URLs on a single website (90%). I randomly selects 40+ pairs of the one-to-one mapped apps and websites for manual examination. I find that almost all websites (except 2) are owned by the app developer, which confirms my intuition. Scheme URLs are found in more than 5 websites for 90% of apps (50 websites for more than half of the apps). It is challenging to remove or upgrade scheme URLs across all these sites.

Insecure Usage of Intent URL. Among the 1,203 Intent URLs, I find 25 Intent URLs did not specify the package name of the target app (only the host or scheme). These 25 Intent URLs can be hijacked.

5.6.2 Measuring Hijacking Risk on Web

To estimate the level of hijacking risks on the web, I now revisit the hijacking attacks detected in §5.5 (those on per-app schemes/hosts). I seek to measure the volume of hijacked links among webpages, and estimation App link's contributions over existing risks introduced by scheme URLs.

Hijacked Mobile Deep Links. I extract links from *Alexa1M* that are registered by multiple apps, which returns 408,455 scheme URLs and 2,741,817 App links. Among them, 7,242 scheme URLs and 2,619,565 App links contain per-app schemes/hosts (*i.e.*, hijacked

links).

The key observation is that App links introduce orders of magnitude more hijacked links that App links than scheme URLs, as shown in Figure 5.12 (log scale y-axis). I further examine the number of *websites* that contain hijacked links. As shown in Figure 5.13, App links have a dominating contribution: 456K websites (out of 1 million, 45.6%) contains per-app App links that are subject to link hijacking. The corresponding number for scheme URL is 5.3K websites (0.5%).

App links, designed as the secure version of deep links, actually expose users to a higher level of risks. Intuitively, http/https links have been used on the web for decades. Once apps register App links, a large number of existing http/https links on the web are automatically interpreted as App links. This creates more opportunities for malicious apps to perform link hijacking.

Links Carrying Sensitive Data. To illustrate the practical consequences of link hijacking, I perform a quick analysis on the hijacked links with a focus on their parameters. A quick keyword search returns 74 sensitive parameter names related to authentications (*e.g.*, `cookie`, `authToken`, `sessionid`, `password`, `access_token`, full list in Table 5.9). I find that 1075 hijacked links contain at least one of the sensitive parameters. A successful hijacking will expose these parameters to the attacker app. This is just one example, and by no means exhaustive in terms of possibly sensitive data carried in hijacked links (*e.g.*, PII, location).

access_token, actionToken, api_key, apikey, apiToken, Auth, auth_key, auth_token, authenticity_token, authkey, authToken, autologin, AWSAccessKeyId, cookie, csrf_token, csrfKey, csrfToken, ctoken, fk_session_id, FKSESSID, FOGSESSID, force_sid, formkey, gsessionid, guestaccesstoken, hkey, IKSESSID, imprToken, jsessionid, key, keycode, keys, LinkedinToken, live_configurator_token, LLSESSID, MessageKey, mrsessionid, navKey, newsid, oauth_callback, oauth_token, pasID, pass, pass_key, password, PHPSESSID, piggybackCookie, plkey, redir_token, reward_key, roken2, seasonid, secret_key, secret_perk_token, ses_key, sesid, SESS, sessid, sessid2b4f0b11dea2f7ae4bfff49b6307d50f, SESSION, session_id, session_rikey, sessionGUID, sessionid, sh_auth, sharedKey, SID, tok, token, uepSessionToken, vt_session_id, wmsAuthSign, ytsession

Table 5.9: Sensitive parameters in mobile deep links.

5.7 Discussion

Key Implications. My results shed light on the practical challenges to mitigate vulnerable mobile deep links. First, scheme URL is designed for mixed purposes, including invoking a generic function (functional/third-party schemes) and launching a target app (per-app schemes). The multipurpose design makes it difficult to uniformly enforce security policies (*e.g.*, associating schemes to apps). A more practical solution should prohibit per-app schemes, while not crippling the widely deployed functional/third-party schemes on the web.

Second, App links and Intent URLs are designed with security in mind. However, their practical usage has deviated from the initial design. Particularly for App links, 98% of apps did not implement link verification correctly. In addition to various configuration errors, a more important reason is unverified links still work on Android, and developers are likely not motivated to verify links. As a result, App links not only fail to provide a better security, but worsen the situation significantly by introducing more hijacked links.

Finally, the insecurity of deep links leads to a tough trade-off between security and usability. Mobile deep links are designed for usability, to enable seamless context-aware transitions from web to apps. However, due to the insecure design, mobile platforms have to constantly prompt users to confirm the links they clicked, which in turn hurts usability. The current solution for Android (and iOS) takes a middle ground, by letting users set "preference" for certain apps to disable prompting. I find this leads to new security vulnerabilities (over permission risk in §5.4.2) that allow malicious apps to hijack arbitrary HTTP/HTTPS URLs in the Android browser.

Legacy Issue. Android does not strongly enforce App link verification possibly due to the legacy issues. First, scheme URLs are still widely used on websites as discussed in §5.6. Disabling scheme links altogether would inevitably affect users' web browsing experience (*e.g.*, causing broken links [4]). Second, according to Google's report [13], over 60% of Android devices are still using Android 5.0 or earlier versions, which do not support App link verification. Android allows apps (6.0 or higher) to use verified App links while maintaining backward compatibility by not enforcing the verification.

Countermeasures. I discuss three countermeasures to mitigate link hijacking risks. In the short term, the most effective countermeasures would be disabling scheme URLs in mobile browsers or webviews. Note that this is not to disable the app interfaces defined by schemes, but to encourage (force) websites to use Intent URLs to invoke per-app schemes safely. Android may also whitelist a set of well-defined functional schemes to avoid massively breaking functional links. For custom scheme URLs that are still used on the web, Android needs to handle their failure gracefully without severely degrading user experience. Second, prohibiting apps from opening unverified App links to prevents link hijacking. The drawback is that apps without a web front would face difficulties to use deep links — they will need to

rely on third-party services such as Brach.io [2] or Firebase [3] to host their association files. Third, addressing the over-permission vulnerability (§5.4.2), by adopting more fine-grained preference setting (*e.g.*, at the host level or even the link level). This threat would also go away if Android strictly enforces App link verifications.

Vulnerability Notification & Mitigation. My study identified new vulnerabilities and attacks, and I am taking active steps to notifying the related parties for the risk mitigation. First, regarding the over-permission vulnerability, I have filed a bug report through Google's Vulnerability Reward Program (VRP) in February 2017. As of June 2017, I have established a case and submitted the second round of materials including the proof-of-concept app and a demo of the attack. I am waiting for further responses from Google. Second, I have reported my findings to the Android anti-malware team and the Firebase team regarding the massive unverified App links and the misconfiguration issues. Details regarding their mitigation plan, however, were not disclosed to us. Third, as shown in §5.4.1, most of the misconfigured App links haven't been fixed after 5 months. In the next step, I plan to contact the developers, particularly those of hijacked apps and help them to mitigate the configuration errors.

Limitations. My study has a few limitations. First, the conclusions are limited to mobile deep links of Android. Although iOS takes a more strict approach to enforcing the link verification, it remains to be seen how well the security guarantee are achieved in practice. The brief measurement in §5.4.1 already shows that iOS universal links also have misconfigurations. More extensive measurements are needed to fully understand the potential security risks of iOS deep links (future work). Second, the measurement scope is still limited comparing to the size of Android app market and the whole web. I argue that data size is sufficient to draw the conclusions. By measuring the most popular apps (160,000+) and web domains (1,000,000), I collect strong evidence on the incompetence of the newly introduced linking mechanisms in providing better security. Third, I only focus on the link hijacking threat, because this is the security issue that App links and Intent URLs are designed to address. There are other threats related to web-to-mobile communications such as exploiting webviews and browsers [51, 98], and cross-site request forgery on apps [69, 120, 132]. my work is complementary to existing work to better understand and secure the web-and-app ecosystem.

Chapter 6

Conclusions and Future Work

In this thesis, I present my attempts to mining security risks in massive datasets: detect data leakage in large datasets while preserving the privacy of sensitive data in outsourced environment, prioritize inter-app communication risks based on analyzing the communication context of large number of Android apps, measure the effectiveness of mobile deep links and their impact on mobile phone and the web.

In Chapter 3, I presented a data leak detection system and the MapReduce collection intersection algorithms for detecting the occurrences of sensitive data patterns in massive-scale content in data storage or network transmission. My system provides privacy enhancement to minimize the exposure of sensitive data during the outsourced detection. I deployed and evaluated my prototype with the Hadoop platform on Amazon EC2 and a local cluster, and achieved 225 Mbps analysis throughput.

In Chapter 4, I presented the design and implementation of MR-Droid, a MapReduce pipeline for large-scale inter-app ICC risk analyses. By constructing ICC graphs with efficient parallelization, my system enables highly scalable inter-app security analysis and accurate risk prioritization. Using MR-Droid, I analyzed 11,996 most popular Android apps (13 million app pairs) and examined their security risk levels against intent hijacking, intent spoofing, and collusion attacks. My analysis reveals new types of app collusion and hijacking risks (e.g., collusion through stealthy implicit intent). I manually inspected a subset of the apps and validated the security risk analysis. The empirical results and manual validation demonstrated the merits of MR-Droid in prioritizing various ICC risks and the effectiveness of the prioritization.

In Chapter 5, I conducted the first large scale measurement study on mobile deep links across popular apps and websites. The results show strong evidences that the newly proposed deep link methods (App links and Intent URLs) in Android fail to address the existing hijacking risks in practice. In addition, I identify new vulnerabilities and empirical misconfigurations in App links, which ultimately expose users to a higher level of risks. Moving forward, I

make a list of suggestions to countermeasure the link hijacking risks in Android.

For future work, I plan to continue to work on mining the security risks in massive datasets. For example, I only measured the hijacking risks of deep links. Deep links can also be used to attack mobile browser or mobile apps. I will work on the detection of vulnerable mobile apps and malicious deep links on web. In addition, I will polish my current approaches making it more practical for various real-world applications. For example, I plan to update the current data leak detection algorithms and platforms so that it can process streaming network traffic efficiently.

Bibliography

[1] Alexa. http://www.alexa.com.

[2] Branch. https://developer.branch.io/, .

[3] Firebase App Indexing. https://firebase.google.com/docs/app-indexing, .

[4] iOS 9.2 Update: The Fall of URI Schemes and the Rise of Universal Links. https://blog.branch.io/ios-9-2-redirection-update-uri-scheme-and-universal-links/, .

[5] Handling App Links. https://developer.android.com/training/app-links/index.html, .

[6] Android Intents with Chrome. https://developer.chrome.com/multidevice/android/intents.

[7] Interacting with Other Apps. https://developer.android.com/training/basics/intents/filters.html, .

[8] App programming guide for ios. https://developer.apple.com/library/content/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/Inter-AppCommunication/Inter-AppCommunication.html, .

[9] Spark. http://spark.apache.org.

[10] Support Universal Links. https://developer.apple.com/library/content/documentation/General/Conceptual/AppSearch/UniversalLinks.html.

[11] Mcafee labs threats report. http://www.mcafee.com/us/resources/reports/rp-quarterly-threats-may-2016.pdf, 2016.

[12] Smartphone apps crushing mobile web times. https://www.emarketer.com/Article/Smartphone-Apps-Crushing-Mobile-Web-Time/1014498, October 2016.

[13] Android platform versions. https://developer.android.com/about/dashboards/index.html, May 2017.

[14] Foto N. Afrati, Anish Das Sarma, David Menestrina, Aditya G. Parameswaran, and Jeffrey D. Ullman. Fuzzy joins using MapReduce. In *IEEE 28th International Conference on Data Engineering (ICDE 2012)*, 2012.

[15] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. *Commun. ACM*, 1975.

[16] Devdatta Akhawe and Adrienne Porter Felt. Alice in warningland: A large-scale field study of browser security warning effectiveness. In *Proc. of USENIX Security*, 2013.

[17] Mishari Almishari, Paolo Gasti, Gene Tsudik, and Ekin Oguz. Privacy-preserving matching of community-contributed content. In *Computer Security ESORICS 2013*. 2013.

[18] Amazon Web Services, Inc. AWS Import/Export Snowball, 2016. https://aws.amazon.com/importexport/, accessed January 2016.

[19] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *PLDI*, 2014.

[20] Muhammad Rizwan Asghar, Mihaela Ion, Giovanni Russello, and Bruno Crispo. Espoonerbac: Enforcing security policies in outsourced environments. *Computers & Security*, 2013.

[21] Internet Assigned Numbers Authority. Uniform resource identifier (URI) schemes. http://www.iana.org/assignments/uri-schemes/uri-schemes.xhtml, February 2017.

[22] H. Bagheri, A. Sadeghi, J. Garcia, and s. Malek. Covert: Compositional analysis of Android inter-app permission leakage. *TSE, IEEE*, 2015.

[23] Ranieri Baraglia, Gianmarco De Francisci Morales, and Claudio Lucchese. Document similarity self-join with MapReduce. In *Data Mining (ICDM), 2010 IEEE 10th International Conference on*, 2010.

[24] Elisa Bertino and Gabriel Ghinita. Towards mechanisms for detection and prevention of data exfiltration by insiders: Keynote talk paper. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, 2011.

[25] Spyros Blanas, Jignesh M. Patel, Vuk Ercegovac, Jun Rao, Eugene J. Shekita, and Yuanyuan Tian. A comparison of join algorithms for log processing in MapReduce. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, 2010.

[26] Marina Blanton, Mikhail J. Atallah, Keith B. Frikken, and Qutaibah M. Malluhi. Secure and efficient outsourcing of sequence comparisons. In *Computer Security - ESORICS 2012 - 17th European Symposium on Research in Computer Security, Pisa, Italy, September 10-12, 2012. Proceedings*, 2012.

[27] Erik-Oliver Blass, Guevara Noubir, and Triet D. Vo-Huu. Epic: Efficient privacy-preserving counting for MapReduce. Cryptology ePrint Archive, Report 2012/452, 2012.

[28] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 1970.

[29] Kevin Borders and Atul Prakash. Quantifying information leaks in outbound web traffic. In *IEEE Symposium on Security and Privacy*, 2009.

[30] Kevin Borders, Eric Vander Weele, Billy Lau, and Atul Prakash. Protecting confidential data on personal computers with storage capsules. In *USENIX Security Symposium*, 2009.

[31] Amiangshu Bosu, Fang Liu, Danfeng (Daphne) Yao, and Gang Wang. Collusive data leak and more: Large-scale threat analysis of inter-app communications. In *Proc. of ASIACCS*, 2017.

[32] BREACH LEVEL INDEX. 2014 year of mega breaches & identity theft, 2014. URL http://breachlevelindex.com/pdf/Breach-Level-Index-Annual-Report-2014.pdf.

[33] Andrei Z. Broder. Some applications of rabin's fingerprinting method. In *Sequences II: Methods in Communications, Security, and Computer Science*, 1993.

[34] Andrei Z. Broder. Identifying and filtering near-duplicate documents. In *Combinatorial Pattern Matching, 11th Annual Symposium*, 2000.

[35] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, and Ahmad-Reza Sadeghi. XManDroid: A new Android evolution to mitigate privilege escalation attacks. Technical report, Technische Universität Darmstadt, 2011.

[36] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. Towards taming privilege-escalation attacks on Android. In *NDSS*, 2012.

[37] Sven Bugiel, Stephan Heuser, and Ahmad-Reza Sadeghi. Flexible and fine-grained mandatory access control on Android for diverse security and privacy policies. In *Proc. of USENIX Security*, 2013.

[38] Min Cai, Kai Hwang, Yu-Kwong Kwok, S. Song, and Yu Chen. Collaborative internet worm containment. *Security Privacy, IEEE*, 2005.

[39] Ning Cao, Cong Wang, Ming Li, Kui Ren, and Wenjing Lou. Privacy-preserving multi-keyword ranked search over encrypted cloud data. *IEEE Trans. Parallel Distrib. Syst.*, 2014.

[40] Bogdan Carbunar and Radu Sion. Joining privately on outsourced data. In *Proceedings of the 7th VLDB Conference on Secure Data Management*, SDM'10, 2010.

[41] Godwin Caruana, Maozhen Li, and Man Qi. A MapReduce based parallel SVM for large scale spam filtering. In *FSKD*, 2011.

[42] Eric Y. Chen, Yutong Pei, Shuo Chen, Yuan Tian, Robert Kotcher, and Patrick Tague. Oauth demystified for mobile application developers. In *Proc. of CCS*, 2014.

[43] Yangyi Chen, Bo Peng, XiaoFeng Wang, and Haixu Tang. Large-scale privacy-preserving mapping of human genomic sequences on hybrid clouds. In *NDSS*, 2012.

[44] Long Cheng, Fang Liu, and Danfeng (Daphne) Yao. Enterprise data breach: causes, challenges, prevention, and future directions. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*.

[45] Yue Cheng, Aayush Gupta, Anna Povzner, and Ali R. Butt. High performance in-memory caching through flexible fine-grained services. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, 2013.

[46] Yue Cheng, Aayush Gupta, and Ali R. Butt. An in-memory object caching framework with adaptive load balancing. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, 2015.

[47] Yue Cheng, M. Safdar Iqbal, Aayush Gupta, and Ali R. Butt. Cast: Tiering storage for data analytics in the cloud. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, 2015.

[48] Yue Cheng, M. Safdar Iqbal, Aayush Gupta, and Ali R. Butt. Pricing games for hybrid object stores in the cloud: Provider vs. tenant. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*, 2015.

[49] Yue Cheng, Fred Douglis, Philip Shilane, Grant Wallace, Peter Desnoyers, and Kai Li. Erasing belady's limitations: In search of flash cache offline optimality. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, 2016.

[50] Yue Cheng, M. Safdar Iqbal, Aayush Gupta, Ali R. Butt, undefined, undefined, undefined, and undefined. Provider versus tenant pricing games for hybrid object stores in the cloud. *IEEE Internet Computing*, 2016.

[51] Erika Chin and David Wagner. Bifocals: Analyzing webview vulnerabilities in Android applications. In *Proc. of WISA*, 2014.

[52] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in Android. In *MobiSys*, 2011.

[53] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in Android. In *Proc. of MobiSys*, 2011.

[54] Jason Croft and Matthew Caesar. Towards practical avoidance of information leakage in enterprise networks. In *6th USENIX Workshop on Hot Topics in Security, HotSec'11*. USENIX Association, 2011.

[55] Akash Das Sarma, Yeye He, and Surajit Chaudhuri. ClusterJoin: A similarity joins framework using Map-Reduce. *Proc. VLDB Endow.*, 2014.

[56] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege escalation attacks on Android. In *ISC*, 2011.

[57] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 2008.

[58] Dong Deng, Guoliang Li, Shuang Hao, Jiannan Wang, and Jianhua Feng. MassJoin: A MapReduce-based method for scalable string similarity joins. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, 2014.

[59] Serge Egelman, Lorrie Faith Cranor, and Jason Hong. You've been warned: An empirical study of the effectiveness of web browser phishing warnings. In *Proc. of CHI*, 2008.

[60] Karim O. Elish, Danfeng (Daphne) Yao, and Barbara G. Ryder. On the need of precise Inter-App ICC classification for detecting Android malware collusions. In *MoST, IEEE S&P*, 2015.

[61] Tamer Elsayed, Jimmy J. Lin, and Douglas W. Oard. Pairwise document similarity in large collections with MapReduce. In *ACL (Short Papers)*, pages 265–268. The Association for Computer Linguistics, 2008.

[62] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol Sheth. TaintDroid: an information flow tracking system for real-time privacy monitoring on smartphones. *Commun. ACM*, 2014.

[63] Steven Englehardt and Arvind Narayanan. Online tracking: A 1-million-site measurement and analysis. In *Proc. of CCS*, 2016.

[64] Michael J. Freedman, Kobbi Nissim, and Benny Pinkas. Efficient private matching and set intersection. In *Advances in Cryptology - EUROCRYPT 2004, International Conference on the Theory and Applications of Cryptographic Techniques*, 2004.

[65] Adam P. Fuchs, Avik Chaudhuri, and Jeffrey S. Foster. SCanDroid: Automated security certication of Android applications. Technical report, Department of Computer Science, University of Maryland, College Park, 2009.

[66] Global Velocity Inc. Global velocity inc., 2015. http://www.globalvelocity.com/, accessed February 2015.

[67] Michael I. Gordon, Deokhwan Kim, Jeff H. Perkins, Limei Gilham, Nguyen Nguyen, and Martin C. Rinard. Information flow analysis of Android applications in DroidSafe. In *NDSS*, 2015.

[68] GTB Technologies Inc. GoCloudDLP, 2015. http://www.goclouddlp.com/, accessed February 2015.

[69] Roee Hay and Yair Amit. Android browser cross-application scripting (cve-2011-2357). Technical report, July 2011.

[70] Roee Hay, Omer Tripp, and Marco Pistoia. Dynamic detection of inter-application communication vulnerabilities in Android. In *Proc. of ISSTA*, 2015.

[71] Qing He, Tianfeng Shang, Fuzhen Zhuang, and Zhongzhi Shi. Parallel extreme learning machine for regression based on MapReduce. *Neurocomput.*, 2013.

[72] 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*, 2014.

[73] 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)*, 2016.

[74] 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)*, 2017.

[75] Kaixi Hou, Hao Wang, and Wu-chun 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, 2015.

[76] Kaixi Hou, Weifeng Liu, Hao Wang, and Wu-chun Feng. Fast Segmented Sort on GPUs. In *Proceedings of the 2017 International Conference on Supercomputing*, ICS '17, 2017.

[77] Kaixi Hou, Hao Wang, and Wu-chun Feng. Gpu-unicache: Automatic code generation of spatial blocking for stencils on gpus. In *Proceedings of the ACM Conference on Computing Frontiers*, CF '17, 2017.

[78] Identifyfinder. Identity finder, 2015. http://www.identityfinder.com/, accessed February 2015.

[79] International Data Corporation (IDC). Smartphone OS Market Share. http://www.idc.com/prodserv/smartphone-os-market-share.jsp, November 2016.

[80] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*, 2012.

[81] S. Jha, L. Kruger, and V. Shmatikov. Towards practical privacy for genomic computation. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, 2008.

[82] Yiming Jing, Gail-Joon Ahn, Adam Doupé, and Jeong Hyun Yi. Checking intent-based communication in Android with intent space analysis. In *Proc. of ASIACCS*, 2016.

[83] David Kantola, Erika Chin, Warren He, and David Wagner. Reducing attack surfaces for intra-application communication in Android. In *Proc. of SPSM*, 2012.

[84] Kaspersky Lab. IT security risks survey 2014: a business approach to managing data security threats, 2014. URL http://media.kaspersky.com/en/IT_Security_Risks_Survey_2014_Global_report.pdf.

[85] William Klieber, Lori Flynn, Amar Bhosale, Limin Jia, and Lujo Bauer. Android taint flow analysis for app sets. In *SOAP*, 2014.

[86] Wang Lam, Lu Liu, STS Prasad, Anand Rajaraman, Zoheb Vacheri, and AnHai Doan. Muppet: MapReduce-style processing of fast data. *Proc. VLDB Endow.*, 2012.

[87] Feng Li, Beng Chin Ooi, M. Tamer Özsu, and Sai Wu. Distributed data management using MapReduce. *ACM Comput. Surv.*, 2014.

[88] Jin Li, Qian Wang, Cong Wang, Ning Cao, Kui Ren, and Wenjing Lou. Fuzzy keyword search over encrypted data in cloud computing. In *INFOCOM*, 2010.

[89] Kang Li, Zhenyu Zhong, and Lakshmish Ramaswamy. Privacy-aware collaborative spam filtering. *IEEE Trans. Parallel Distrib. Syst.*, 2009.

[90] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick Mcdaniel. IccTA: Detecting inter-component privacy leaks in Android apps. In *ICSE*, 2015.

[91] Li Li, Alexandre Bartel, TegawendéF. Bissyandé, Jacques Klein, and YvesLe Traon. ApkCombiner: Combining multiple Android apps to support inter-app analysis. In *IFIP SEC*. 2015.

[92] Po-Ching Lin, Ying-Dar Lin, Yuan-Cheng Lai, and Tsern-Huei Lee. Using string matching for deep packet inspection. *Computer*, 2008.

[93] Fang Liu, Xiaokui Shu, Danfeng Yao, and Ali R. Butt. Privacy-preserving scanning of big content for sensitive data exposure with mapreduce. In *CODASPY*, 2015.

[94] Fang Liu, Haipeng Cai, Gang Wang, Danfeng (Daphne) Yao, Karim O. Elish, and Barbara G. Ryder. MR-Droid: A scalable and prioritized analysis of inter-app communication risks. In *Proc. of MoST*, 2017.

[95] Fang Liu, Chun Wang, Andres Pico, Danfeng (Daphne) Yao, and Gang Wang. Measuring the insecurity of mobile deep links in android. In *Proc. Proc. of USENIX Security*, 2017.

[96] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. CHEX: Statically vetting Android apps for component hijacking vulnerabilities. In *CCS*, 2012.

[97] Yanbin Lu. Privacy-preserving logarithmic-time search on encrypted data in cloud. In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*, 2012.

[98] Tongbo Luo, Hao Hao, Wenliang Du, Yifei Wang, and Heng Yin. Attacks on webview in the Android system. In *Proc. of ACSAC*, 2011.

[99] Claudio Marforio, Aurélien Francillon, and Srdjan Capkun. Application collusion attack on the permission-based security model and its implications for modern smartphone systems. Technical report, ETH Zurich, 2011.

[100] Claudio Marforio, Hubert Ritzdorf, Aurélien Francillon, and Srdjan Capkun. Analysis of the communication between colluding applications on modern smartphones. In *ACSAC*, 2012.

[101] Collin Mulliner, Jon Oberheide, William Robertson, and Engin Kirda. PatchDroid: Scalable third-party security patches for Android devices. In *Proc. of ACSAC*, 2013.

[102] Patrick Mutchler, Adam Doupe, John Mitchell, Christopher Kruegel, and Giovanni Vigna. A large-scale study of mobile web app security. In *MOST*, 2015.

[103] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. Effective inter-component communication mapping in Android with Epicc: An essential step towards holistic security analysis. In *USENIX Security*, 2013.

[104] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick McDaniel. Composite constant propagation: Application to Android inter-component communication analysis. In *ICSE*, 2015.

[105] Damien Octeau, Somesh Jha, Matthew Dering, Patrick McDaniel, Alexandre Bartel, Li Li, Jacques Klein, and Yves Le Traon. Combining static analysis with probabilistic models to enable market-scale Android inter-component analysis. In *POPL*, 2016.

[106] Panagiotis Papadimitriou and Hector Garcia-Molina. Data leakage detection. *IEEE Trans. Knowl. Data Eng.*, 2011.

[107] Janak J. Parekh, Ke Wang, and Salvatore J. Stolfo. Privacy-preserving payload-based correlation for accurate malicious traffic detection. In *Proceedings of the 2006 SIGCOMM Workshop on Large-scale Attack Defense*, LSAD '06, 2006.

[108] Daniel Peng and Frank Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, 2010.

[109] Niels Provos, Dean McNamee, Panayiotis Mavrommatis, Ke Wang, and Nagendra Modadugu. The ghost in the browser analysis of web-based malware. In *HotBots*, 2007.

[110] Michael O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Harvard Aliken Computation Laboratory, 1981.

[111] S. Rasthofer, S. Arzt, E. Lovat, and E. Bodden. Droidforce: Enforcing complex, data-centric, system-wide policies in Android. In *ARES*, 2014.

[112] Tristan Ravitch, E. Rogan Creswick, Aaron Tomb, Adam Foltzer, Trevor Elliott, and Ledah Casburn. Multi-App security analysis with FUSE: Statically detecting Android app collusion. In *Proc. of PPREW*, 2014.

[113] Chuangang Ren, Yulong Zhang, Hui Xue, Tao Wei, and Peng Liu. Towards discovering and understanding task hijacking in Android. In *USENIX Security*, 2015.

[114] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, 2009.

[115] Dan Rowinski. Digital strategy: Why native apps versus mobile web is a false choice. https://arc.applause.com/2016/09/13/native-apps-versus-mobile-web-decision/, September 2016.

[116] Indrajit Roy, Srinath T. V. Setty, Ann Kilzer, Vitaly Shmatikov, and Emmett Witchel. Airavat: Security and privacy for MapReduce. In *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation*, 2010.

[117] Carl Sabottke, Octavian Suciu, and Tudor Dumitras. Vulnerability disclosure in the age of social media: Exploiting Twitter for predicting real-world exploits. In *USENIX Security*, 2015.

[118] D. Sbîrlea, M. G. Burke, S. Guarnieri, M. Pistoia, and V. Sarkar. Automatic detection of inter-application permission leaks in Android applications. *IBM J. Res. Dev.*, 2013.

[119] Roman Schlegel, Kehuan Zhang, Xiao-yong Zhou, Mehool Intwala, Apu Kapadia, and XiaoFeng Wang. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *NDSS*, 2011.

[120] Roman Schlegel, Kehuan Zhang, Xiao-yong Zhou, Mehool Intwala, Apu Kapadia, and XiaoFeng Wang. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *Proc. of NDSS*, 2011.

[121] Rainer Schnell, Tobias Bachteler, and Jörg Reiher. Privacy-preserving record linkage using bloom filters. *BMC medical informatics and decision making*, 9(1):41, 2009.

[122] Xiaokui Shu and Danfeng (Daphne) Yao. Data leak detection as a service. In *SecureComm*, 2012.

[123] Xiaokui Shu, Danfeng Yao, and Elisa Bertino. Privacy-preserving detection of sensitive data exposure with applications to data-leak detection as a service. *Information Forensics & Security (TIFS), IEEE Transactions on*, 2015.

[124] Xiaokui Shu, Jing Zhang, Danfeng Yao, and Wu-Chun Feng. Rapid and parallel content screening for detecting transformed data exposure. In *In Proceedings of the International Workshop on Security and Privacy in Big Data (BigSecurity), co-located with IEEE INFOCOM*, BigSecurity 2015. IEEE, 2015.

[125] Xiaokui Shu, Fang Liu, and Danfeng (Daphne) Yao. *Rapid Screening of Big Data Against Inadvertent Leaks*. 2016.

[126] Dawn Xiaodong Song, David Wagner, and Adrian Perrig. Practical techniques for searches on encrypted data. In *IEEE Symposium on Security and Privacy*, 2000.

[127] Anna Cinzia Squicciarini, Smitha Sundareswaran, and Dan Lin. Preventing information leakage from indexing in the cloud. In *IEEE International Conference on Cloud Computing, CLOUD 2010*, 2010.

[128] Statista: The statistics protal. Number of available applications in the Google Play store from December 2009 to February 2016. http://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/, 2016.

[129] J. A. Stuart and J. D. Owens. Multi-gpu mapreduce on gpu clusters. In *IPDPS*, 2011.

[130] Symantec. Symantec data loss prevention, 2015. http://www.symantec.com/data-loss-prevention, accessed February 2015.

[131] Suman Nath Tanzirul Azim, Oriana Riva. uLink: Enabling user-defined deep linking to app content. In *Proc. of Mobisys*, 2016.

[132] Takeshi Terada. Attacking Android browsers via intent scheme urls. Technical report, March 2014.

[133] Guliz Seray Tuncay, Soteris Demetriou, and Carl A. Gunter. Draco: A system for uniform and fine-grained access control for web code on Android. In *Proc. of CCS*, 2016.

[134] Rares Vernica, Michael J. Carey, and Chen Li. Efficient parallel set-similarity joins using MapReduce. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, 2010.

[135] Rui Wang, Luyi Xing, XiaoFeng Wang, and Shuo Chen. Unauthorized origin crossing on mobile platforms: Threats and mitigation. In *Proc. of CCS*, 2013.

[136] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps. In *CCS*, 2014.

[137] Min Wu, Robert C. Miller, and Simson L. Garfinkel. Do security toolbars actually prevent phishing attacks? In *Proc. of CHI*, 2006.

[138] Luyi Xing, Xiaolong Bai, Tongxin Li, XiaoFeng Wang, Kai Chen, Xiaojing Liao, Shi-Min Hu, and Xinhui Han. Cracking app isolation on apple: Unauthorized cross-app resource access on MAC OS X and iOS. In *Proc. of CCS*, 2015.

[139] Lei Xu, Han-Yee Kim, Xi Wang, Weidong Shi, and Taeweon Suh. Privacy preserving large scale DNA read-mapping in MapReduce framework using FPGAs. In *24th International Conference on Field Programmable Logic and Applications, FPL 2014*, 2014.

[140] Kun Yang, Jianwei Zhuge, Yongke Wang, Lujue Zhou, and Haixin Duan. IntentFuzzer: Detecting capability leaks of Android applications. In *Proc. of ASIACCS*, 2014.

[141] Zhemin Yang, Min Yang, Yuan Zhang, Guofei Gu, Peng Ning, and X. Sean Wang. AppIntent: analyzing sensitive data transmission in Android for privacy leakage detection. In *CCS*, 2013.

[142] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *Foundations of Computer Science, 1986., 27th Annual Symposium on*, 1986.

[143] Xun Yi, Russell Paulet, and Elisa Bertino. *Private Information Retrieval*. Synthesis Lectures on Information Security, Privacy, and Trust. 2013.

[144] Eunjung Yoon and A. Squicciarini. Toward detecting compromised MapReduce workers through log analysis. In *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, 2014.

[145] Chunwang Zhang, Ee-Chien Chang, and R.H.C. Yap. Tagged-MapReduce: A general framework for secure computing with mixed-sensitivity data on hybrid clouds. In *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, 2014.

[146] Hao Zhang, Danfeng Daphne Yao, and Naren 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*, 2014.

[147] Hao Zhang, Danfeng (Daphne) Yao, and Naren 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)*, 2016.

[148] Hao Zhang, Danfeng (Daphne) Yao, Naren Ramakrishnan, and Zhibin Zhang. Causality reasoning about network events for detecting stealthy malware activities. *Computers & Security*, 2016.

[149] Mu Zhang and Heng Yin. AppSealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in Android applications. In *Proc. of NDSS*, 2014.

[150] Xuyun Zhang, L.T. Yang, Chang Liu, and Jinjun Chen. A scalable two-phase top-down specialization approach for data anonymization using MapReduce on cloud. *TPDS, IEEE*, 2014.

[151] Yuan Zhang, Min Yang, Guofei Gu, and Hao Chen. FineDroid: Enforcing permissions with system-wide application execution context. In *Proc. of SecureComm*, 2015.