# TraceDroid: Detecting Android Malware by Trace of Privacy Leakage

Yueqing Wu[1], Hao Fu[2], Guoming Zhang[1], Bin Zhao[1], Minghui Xu[1], Yifei Zou[1], Xiaotao Feng[3], and Pengfei Hu[1(✉)]

[1] Shandong University, Qingdao 266237, CN, China
yueqingwu@sdu.edu.cn, guomingzhang@sdu.edu.cn, zhaobinsdu@sdu.edu.cn, mhxu@sdu.edu.cn, yfzou@sdu.edu.cn, phu@sdu.edu.cn
[2] UC Davis, Davis, CA 95616, US
haofu@ucdavis.edu
[3] JD.com American Technologies Corporation, Mountain View, CA 94043, US
xiaotao.feng@jd.com

**Abstract.** Along with the popularity of the Android operating system, 98% of mobile malware targets Android devices [1], which has become one of the primary source for privacy leakage. Detecting malicious network transmissions in these apps is challenging because the malware hides its behavior and masquerades as benign software to evade detection. In this work, we propose TraceDroid, a framework that can automatically trace abnormally sensitive network transmissions to detect the malware. By leveraging the static and dynamic analysis, the sensitive informations can be firstly inferred from the call graph, and then, the sensitive transmissions can be detected by analyzing the network traffic per transfer and sensitive information with a machine learning classifier. We validate TraceDroid on 1444 malware and 700 benign applications. And our experiments show that TraceDroid can detect 3433 sensitive connections across 2144 apps with an accuracy of 94%.

**Keywords:** Android malware detection · Static analysis · Dynamic analysis

## 1 Introduction

With the widespread use of mobile devices running Android, it has become the dominant operating system. Users can install applications from third parties without performing any malware checks [31]. As a result, the number of malware samples targeted the Android ecosystem has skyrocketed in recent years, which poses significant threat to user's privacy. Bad guys can infer information such as home address from user privacy, causing great harm to users [3,5,6,29].

This work is supported by the National Key Research and Development Program of China (No. 2021YFB3100400), the Shandong Science Fund for Excellent Young Scholars (No. 2022HWYQ-038).


L. Wang et al. (Eds.): WASA 2022, LNCS 13471, pp. 466–478, 2022.
https://doi.org/10.1007/978-3-031-19208-1_38

However, detecting malware on mobile devices is a big challenge. Due to the limitations of Android's permission mechanism [23], malware can disguise itself as a benign application by using normal APIs to fool the security audit module [28]. For instance, when a user clicks button of an application which shows sending SMS messages, it might transmit the user's sensitive contact information along with that SMS at the background without notifying the user. Furthermore, the sensitive network transmissions are not necessarily regarded as malicious traffic, which is probably sent to a benign server for normal operation. Hence, the key of differentiating normal and abnormal sensitive transmissions lies in properly understanding the intent. To detect the malware, previous works [18, 19] leverage natural language processing techniques to understand whether an application's description is consistent with its permission setting. However, they are unable to infer the privacy leakage. Some of the detection methods are limited by the complexity of Android APIs and runtimes, which include millions of lines of code [8]. Most importantly, they only focus on detecting sensitive traffic and fail to distinguish between normal and abnormal sensitive traffic. Some people use the network traffic of application for detection, but they are not always accurate [17].

**TraceDroid.** To address the aforementioned issues, we propose TraceDroid, which combines static analysis, dynamic analysis, and machine learning methods to detect abnormal sensitive network traffic induced by malware. First, TraceDroid employs a static analysis approach to derive possible execution traces of application and to identify sensitive information. However, some applications send information after encrypting the name of an external server in order to avoid security checks, and the destination server address is only visible at runtime. Therefore, TraceDroid also utilizes a dynamic analysis method to collect runtime information to uncover disguised malware. This hybrid approach achieves better performance than purely static or dynamic analysis methods. Moreover, we are able to obtain more network traffic data and provide a better characterization of network behavior by leveraging a hybrid analysis approach than the widely used static analysis approach [8,14]. At last, the transmission data will be used for model training. We can apply the well tuned model to identify malware even when the source code is unavailable, which could be directly integrated into network-based intrusion detection system.

Our contributions can be summarized as follows.

- We propose an Android malware detection method, TraceDroid, which performs ingress analysis of malicious traffic by lightweight static analysis and pinpoints negative network transmissions by detailed dynamic analysis.
- TraceDroid can distinguish abnormal network traffic. Compared with previous work, TraceDroid achieves higher accuracy and better runtime efficiency while being more robust to malware variants and Android API updates.
- We evaluate TraceDroid on a dataset which includes 1444 malware from various malware datasets and 700 market apps downloaded from AppStore. The

results show that TraceDroid can detect 3433 sensitive connections across 2144 apps with an accuracy of 94%.

**Paper Organization.** The remaining paper is organized as follows. We introduce related work in Sect. 2. In Sect. 3, we presents our system TraceDroid, and describes its technical details. Section 4 performs the evaluation of our system. Section 5 summarizes our work.

## 2 Related Work

Emerging research efforts have been made on Android malware detection, which can be broadly classified into Signature-Based( [11,13]), ML-Based([16,26]), and Behavior-based([25,26]) respectively. The signature-based methods have low computing complexity and can provide specific evidence (detected malicious feature codes) to explain. However, such methods can easily be bypassed by malicious code. Machine learning-based methods have been investigated to detect Android malware. However, the performance of pure machine learning methods is limited to selected features and existing training datasets [4]. MaMaDroid [16] utilizes the statistical methods of Markov chains to detect malware, but it is also vulnerable to some attacks using evasion techniques [7].

FlowDroid [2] and DroidSafe [10] both utilize static analysis solutions to precisely detect suspicious information flows, but the results become inaccurate because the visited code paths are not always feasible. Based on dynamic analysis tools, TaintDroid [9] is a real-time privacy monitoring system. By modifying the Dalvik virtual machine, TaintDroid can report information leaks while the application runs on the Android device. It only identifies leaks triggered during execution, so a driver with good code coverage is required. Naturally, some tools use hybrid analysis, such as AppAudit, to avoid the weaknesses of using a single analysis. Still, it leaves out most unknown branches and only follows one chapter identified by static checks. The downside of all the tools mentioned above is that they treat any breach of user data as malicious, leading to numerous false alarms. Compared to the above system, TraceDroid has the flexibility to extend code coverage based on context and explore as many possible paths as possible when discovering unknown branches. Our hybrid program analysis approach further improves detection efficiency and reduces false positives.

## 3 Design Of Tracedroid

### 3.1 Overview

This section introduces the design of TraceDroid, which leverages data flow analysis technology to determine the source of perceptual data and then track the information flow to select the final destination of sensitive data. The system overview of TraceDroid is given in Fig. 1. It includes three main components: Static analysis, dynamic analysis, and classification.

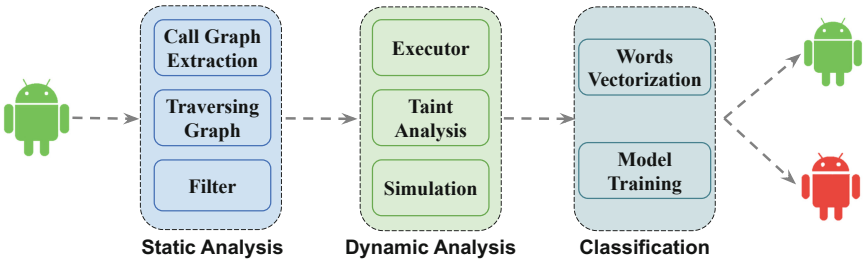


**Fig. 1.** The workflow of TraceDroid.

### 3.2 Static Analysis

The goal of static analysis is to construct the call graph of the target application, which can assist the dynamic analysis and improve its efficiency.

**Call Graph Extractions.** In contrast to traditional Java programs, which have only one entry point (i.e. main), Android apps have multiple entry points. Specifically, Android applications consist of multiple components that each Activity or Service component is a Java class, and each event listener and lifecycle method serve as an entry point for a specific event. Thus, to fully capture the sensitive information traces, all possible transitions in the application's lifecycle must be captured precisely.

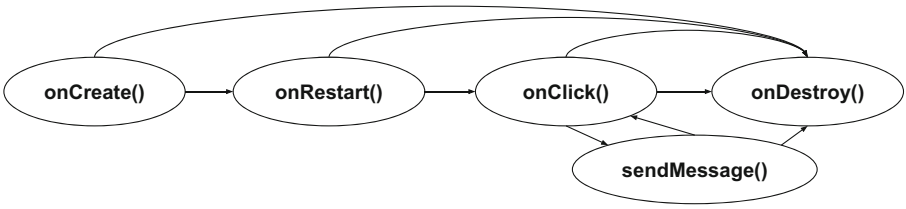


**Fig. 2.** An example of call graph

In order to construct an application's call graph, prior work typically creates one or more virtual main routines that are shared by multiple components [2,27]. However, some components without leaking information will be included with the above methods which will introduce unnecessary interference. Besides, the shared virtual main program may obscure the connection between components. Instead of building a shared virtual main program, each component in TraceDroid has a separate call graph to eliminate clutter and reduce the overhead of dynamic analysis. As shown in Fig. 2, event listeners `onClick()` are embedded in the component and registered after `onCreate()`. `onClick()` is a UI callback function that will be called when the appropriate button is clicked. As the underlying

static analysis framework, we leverage Soot [24] which is a Java optimization and analysis framework to extract call graphs.

**Traversing Graph.** After obtaining the call graph, TraceDoid leverages it to quickly locate the sensitive APIs call. A `source` is an API call which accesses to the sensitive information. Sensitive information includes device identifiers, SMS, contact data, etc. All of these data items are retrieved, sent, or stored through the Android APIs, which is listed in Susi [21]. Typically, `getText()` at line 8 shown in Listing 1.1 is a source. For each `source`, the corresponding entry point of the component is extracted by applying a graph traversal algorithm in the call graph. For instance, the entry point `onRestart()` of the component `PrivataDateLeakag` in Listing 1.1 is located through breadth-first search beginning with `getText()` on the call graph.

**Filter.** Since the static analysis component has a lot of unnecessary entry functions, we thus use filters in the static analysis results to improve the efficiency of our analysis. Some keywords are used in the filter to filter out unnecessary entries such as those containing Android kernel features.

### 3.3 Dynamic Analysis

The dynamic analysis component consists of an execution system with a taint analysis module and a simulation of the Android runtime.

**Executor.** The executor is based on a specially designed Dalvik virtual machine which can unpack Android package files and execute bytecode instructions directly. After static analysis, a set of **traces** can be derived. Then the **traces** are fed into the execution system. Note that each trace is a sequence of specific API calls beginning with a lifecycle callback and ending with an API call related to sensitive information.

For instance, for the entry point `onRestart()` in `PrivateDataLeakage`, TraceDroid builds an execution trace `onRestart()` to `onClick()` that informs the executor to invoke `onClick()` after calling `onRestart()`. When the framework restarts the application, the application reads the password from the text box (line 8). When the user clicks the active button (`onClick()`), the password is sent via SMS (line 20). This constitutes a tainted data flow from the password field (**source**) to the SMS API (**sink**). In this example, `sendMessage()` is associated with a button in the application UI, which is triggered when the user clicks the button. The execution trace is generated by applying depth-first search to find a path from `onRestart()` to `onClick()` in the call graph (Fig. 2). The default values of global variables are normally initialized at the lifecycle callbacks such as `onCreate()` and `onStart()`. We choose to perform these callbacks to reduce the unknown variables. AS it reduces the number of unknown branches to be explored, it improves the efficiency of dynamic analysis.

**Listing 1.1.** Example Android Application

```
class PrivateDataLeakage extends Activity {
  private User user = null;
  void onCreate() {...//initiate the activity }
  void onRestart() {
      EditText usernameText = (EditText)findViewById(R.id.username);
          EditText passwordText =
              (EditText)findViewById(R.id.password);
          String uname = usernameText.toString();
          String pwd = passwordText.getText().toString(); //source
          user = new User(uname, pwd);
  }
  void sendMessage(View view) {
      if(user != null){
          String password = getPassword();
          String obfuscatedUsername = "";
          for(char c : password.toCharArray())
          obfuscatedUsername += c + "_";
            String message = "User: " + user.getUsername() + " |
                Pwd: " + obfuscatedUsername;
            SmsManager smsmanager = SmsManager.getDefault();
            Log.i("TEST", "sendSMS"); //sink
            smsmanager.sendTextMessage("+49 1234", null, message,
                null, null); //sink, leak
          }
  }
  void onDestroy() {... //finish the activity}
}
```

**Taint Analysis.** Here, the unknown variables are often closely related to some factors, e.g., user input, device status, surrounding environment, etc. Thus, malicious applications may take advantage of some factors to hide their behavior, creating malicious code that can only be triggered under certain circumstances. To tackle this problem, TraceDroid not only introduces the function of snapshots to handle different cases of unknown quantities, but also proposes a rule base to deal with the problem of path explosion caused by unknown variables. Specifically, if an unknown branch is encountered, TraceDroid creates a snapshot to store the state of the executor and presses the snapshot onto the stack. If the termination condition of a loop or recursion is an unknown quantity, code that contains the loop or recursion may cause an infinite number of paths to be explored. We choose the way that execute the block under the loop only once, and mark all the variables in the block that accept the new value. After exploring the block, the marked variable is symbolically modeled for the rest of the execution. During execution, whenever the source API is invoked, the pollution

analysis module begins to track the propagation of the correspondingly sensitive values. When one or more sensitive values reach a network connection API call (a **sink**, such as the `openConnection()` or `connect()` ), this means that the transport is sensitive, the corresponding runtime information, such as network traffic data, is recorded. We employ the general contamination strategy which has been used in the previous work [9,27] to specify the propagation process.

**Simulation of the Android runtime.** Accurate modeling of the Android runtime state is required to perform taint analysis correctly. Therefore, we manually pad the incomplete Android SDK and emulate the core functionalities. Our inspiration for emulating Android comes from [10]. But the Android device implementation used is only developed for static analysis and does not extend well enough to support our dynamic analysis. We supplemented the android framework to make it support more functions. Meanwhile, the return value of the function is simulated to support our dynamic analysis.

### 3.4 Transmission Classification

The final step is to detect the Android malware by analyzing the traffic generated by the dynamic analysis component. TraceDroid uses a supervised learning approach to train classifiers that we aim to operate in local-based or network-based intrusion detection systems. To generate the representative features from URL sets in the traffic, we finally choose lexical features, as the lexical features contain the purpose of transmissions which can be used to distinguish suspicious and benign traces.

**Words Vectorization.** To extract the lexical features from traffic, a bag-of-words model [15] is employed, which is often used for spam detection. In our framework, URLs can be divided into tags using certain characters as delimiters. Each different tag is then treated as an independent feature. Each collected data stream is converted into a vector of binary values. To reduce the computation cost, we can't use word bags directly because this can result in vast feature Spaces. As described in [22], we limit the size of a feature set by removing tokens that are rarely present in a stream.

**Model Training.** Since TraceDroid is commonly used in traffic classification, we consider Decision Tree as a learning classifier [20,22]. We use labeled transmissions as training and test data, and use ten-fold cross validation [12], which is the standard method for evaluating machine learning solutions. According to the hybrid analysis tool, the traffic generated from different code paths in target app probably goes to the same URL, thus, we merge the same URL connections into one transmission. Later, for the collected transports, we check the target hostname to see if it belongs to an Advertising(AD) server or a malicious server and flag the transports as illegal. Then, we need to check the plain text content

passed through the stream to prevent the server from sending a response that is related to the user data being sent. In order to test the transmission of flow is legal, we will intercept these flows in some special way and repackage these applications. If the function of these applications is affected, then we will mark the transmission of flow as legal and if the application is not affected, so we will have sufficient reason to mark the transmission of flow as illegal transfer.

## 4 Evaluation

### 4.1 Experimental Settings

**Datasets.** We first extract 1223 malicious sensitive transmissions and record the corresponding traffic from the classic malicious software set [30]. In addition, we also obtain 700 malicious samples from `VirusShare`[1], as well as 1147 malicious sensitive transmissions. We crawl 700 applications on the legitimate app store. Since the architecture of android system has changed in recent years, our dataset includes new versions of android applications to make our analysis more convincing.

**Matrics.** In the experiments, we employ the standard F-measure metric, *Accuracy*, $TP, FP, FN$ to evaluate the performance of TraceDroid under different settings. The *Accuracy* refers to the ratio of correctly predicted samples to the total samples. $TP$ denotes the number of correctly classifying normal samples as normal, and $FP$ and $FN$ indicate the number of samples mistakenly identified as malicious and benign respectively.

### 4.2 Comparison with Benchmark

We first evaluate the performance of TraceDroid on base datasets compared with the other two state-of-the-art Android detection frameworks. Table 1 summarizes the detection results on DroidBench. DroidBench[2] is an open-source benchmarking suite that contains 118 hand-crafted applications. Particularly, it can utilize various features of programming languages to bypass static pollution analysis. We removed 14 apps due to the inter-app communication involved and other reasons.

**Compare with Other Static Detection Methods.** The detection accuracy of FlowDroid is only 76.8%, that is because it can not effectively analyze the runtime data of app and the modeling of the lifecycle of FlowDroid is imprecise.

[1] https://virusshare.com/.
[2] https://github.com/secure-software-engineering/DroidBench.

**Table 1.** Detection results on DroidBench

| Tools | FP | Accuracy | Precision |
|---|---|---|---|
| FlowDroid [2] | 10 | 76.8% | 70.5% |
| AppAudit [27] | 2 | 50.5% | 91.3% |
| TraceDroid | 0 | 98.3% | 100% |

**Compare with Other Hybrid Analysis Detection Methods.** As we can see that TraceDroid achieves higher detection accuracy than AppAudit. The first underlying reason is that AppAudit chooses to terminate the current execution when it encounters a sink, but satisfying reachability does not imply malicious transfers, as discussed in Sect. 3. The second, AppAudit does not take into account the diversity of unknown variables and keeps hanging on to an unknown branch in one direction, which can reduce the detection accuracy of AppAudit. Furthermore, since Android contains various mechanisms, AppAudit does not support any of these Android features.

In summary, TraceDroid provides a more complete dynamic analysis implementation that not only simulates the behavior of the Android runtime to support various mechanisms, but also tracks communication between multiple components.

### 4.3 Real App

**Trasmission Detection.** In this section, we further present the performance comparison with VirusTotal[3] VirusTotal is a popular website that scans submitted URLs with the latest 68 anti-virus engines. In Fig. 3, *Malware* represents the 1,444 malware apps we collected, *AppStore* represents the 700 apps we download from the AppStore, and *All* represents all the pieces of Malware and apps. It can be observed that the performance of VirusTotal is always inferior than TraceDroid no matter on which apps.

**Table 2.** Classification results in different scenarios

| Scene | Class | Precision | F-measure |
|---|---|---|---|
| Local-based | Illegal | 0.982 | 0.962 |
| Local-based | Legal | 0.872 | 0.905 |
| Network-based | Illegal | 0.918 | 0.924 |
| Network-based | Legal | 0.910 | 0.915 |

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

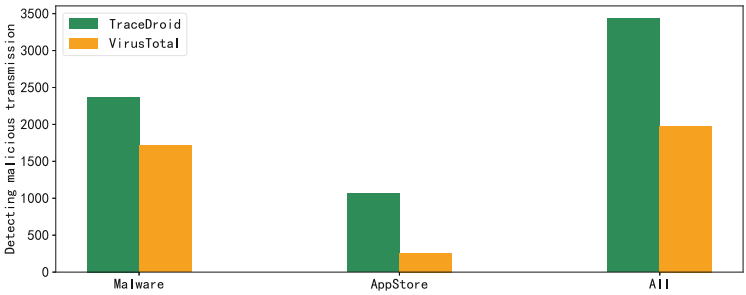


**Fig. 3.** Detecting malicious transmission.

**Comparison with Different Scenarios.** In order to verify the effectiveness of TraceDroid in multiple scenarios, we designed two scenarios: the first is local host system of automatically finds the disclosure points and the second is the scenario involves only the flows of sensitive transmissions. Table 2 shows the classification results in different scenarios. For local-based scene, Table 2 shows that TraceDroid has high precision and F-measure in identifying illegal transmissions. After manually inspecting the misidentified instances, we found that their URLs were very similar to the benign addresses. Also, they put the sensitive data into their body rather than the URL, which makes the URL-based detection more difficult to correctly label them. We plan to consider more features to further reduce the false negatives in the future. For network-based scene, based on the sensitive transmissions we collected, we add the non-sensitive traffic flows to the legitimate class. This reflects the real environment of the network-based detection. Table 2 summarizes our results. As we can see that the prediction accuracy of network-based scenario is slightly lower the local-based detection's.

TraceDroid found 3,433 suspicious behaviors in marketing apps. In order to analyze the reasons why some behaviors are classified as suspicious cases, we compare the behavior characteristics with common behaviors through data visualization and manual comparison, and obtain some results as follows.

**Finding1.** The 700 apps acquired in the AppStore that are generally considered benign, we detect 1,063 sensitive transmissions and 74.5% of these sensitive transmissions are related to advertising, as shown in Fig. 4. For example, a weather-forecasting app takes a user's location and sends it to an AD server.

**Finding2.** Some applications use specific environments to steal privacy. Such as an app from the DroidDream malware family only sends messages at night.

**Finding3.** Most malicious transcribes use resources that have clear semantics and are related to users' private information. To be specific, many malicious transcribes obtain the users' personal information and write them to files or

logs. Then malicious transcribes use APIs under NETWORK and SMS packets to transport sensitive information.

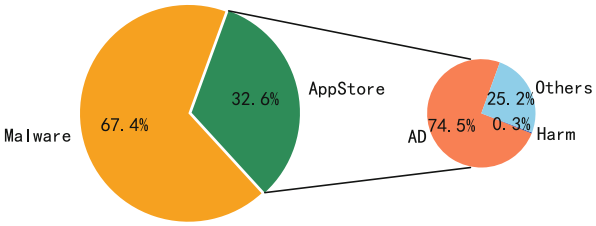


**Fig. 4.** A sensitive transport classification map of a dataset downloaded from AppStore.

## 5 Conclusion

This paper proposes TraceDroid, an Android malware detection system. At first we apply lightweight static analysis to get the entry points, and then perform dynamic analysis to track the traces of privacy leaks. To the best of our knowledge, our framework can identify anomalies in sensitive information instead of treating all sensitive information transmission as unreliable. We have conducted plenty of experiments to evaluate the performance of TraceDroid and results show that TraceDroid can effectively detect unknown malware samples with a 94% accuracy. However, there are still some problems to be explored, such as insufficient sample collection and the accuracy of online detection of unknown malware is low. In the next work, we will collect more samples, optimize our model, and improve the accuracy of unknown software detection.

## References

1. Cyber security statistics the ultimate list of stats, data & trends. purplesec.us/resources/cyber-security-statistics/
2. Arzt, S., et al.: Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In: PLDI (2014)
3. Cai, Z., He, Z.: Trading private range counting over big iot data. In: 39th IEEE International Conference on Distributed Computing Systems, ICDCS 2019, Dallas, TX, USA, July 7–10, 2019. pp. 144–153. IEEE (2019). https://doi.org/10.1109/ICDCS.2019.00023
4. Cai, Z., He, Z., Guan, X., Li, Y.: Collective data-sanitization for preventing sensitive information inference attacks in social networks. IEEE Trans. Dependable Secur. Comput. **15**(4), 577–590 (2018). https://doi.org/10.1109/TDSC.2016.2613521
5. Cai, Z., Zheng, X.: A private and efficient mechanism for data uploading in smart cyber-physical systems. IEEE Trans. Netw. Sci. Eng. **7**(2), 766–775 (2020). https://doi.org/10.1109/TNSE.2018.2830307

6. Cai, Z., Zheng, X., Wang, J., He, Z.: Private data trading towards range counting queries in internet of things. IEEE Trans. Mob. Comput. (2022)
7. Chen, X., et al.: Android HIV: a study of repackaging malware for evading machine-learning detection. IEEE Trans. Inf. Forensics Security **15**, 987–1001 (2019)
8. Chen, X., Zhu, S.: Droidjust: automated functionality-aware privacy leakage analysis for android applications. In: WiSec (2015)
9. Enck, W., et al.: Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In: OSDI (2010)
10. Gordon, M.I., Kim, D., Perkins, J.H., Gilham, L., Nguyen, N., Rinard, M.C.: Information flow analysis of android applications in droidsafe. In: NDSS (2015)
11. Grace, M.C., Zhou, Y., Zhang, Q., Zou, S., Jiang, X.: Riskranker: scalable and accurate zero-day android malware detection. In: Davies, N., Seshan, S., Zhong, L. (eds.) The 10th International Conference on Mobile Systems, Applications, and Services, MobiSys'12, Ambleside, United Kingdom - June 25–29, 2012. pp. 281–294. ACM (2012). https://doi.org/10.1145/2307636.2307663
12. Kohavi, R., et al.: A study of cross-validation and bootstrap for accuracy estimation and model selection. In: Ijcai (1995)
13. Lee, J., Lee, S., Lee, H.: Screening smartphone applications using malware family signatures. Comput. Secur. **52**, 234–249 (2015). https://doi.org/10.1016/j.cose.2015.02.003
14. Lu, K., et al.: Checking more and alerting less: Detecting privacy leakages via enhanced data-flow analysis and peer voting. In: NDSS (2015)
15. Ma, J., Saul, L.K., Savage, S., Voelker, G.M.: Beyond blacklists: Learning to detect malicious web sites from suspicious urls. In: KDD (2009)
16. Mariconti, E., Onwuzurike, L., Andriotis, P., Cristofaro, E.D., Ross, G., Stringhini, G.: Mamadroid: Detecting android malware by building markov chains of behavioral models (2017)
17. Meng, Z., Xiong, Y., Huang, W., Qin, L., Jin, X., Yan, H.: Appscalpel: combining static analysis and outlier detection to identify and prune undesirable usage of sensitive data in android applications. Neurocomputing **341**, 10–25 (2019). https://doi.org/10.1016/j.neucom.2019.01.105
18. Pandita, R., Xiao, X., Yang, W., Enck, W., Xie, T.: WHYPER: towards automating risk assessment of mobile applications. In: King, S.T. (ed.) Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14–16, 2013. pp. 527–542. USENIX Association (2013). www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/pandita
19. Qu, Z., Rastogi, V., Zhang, X., Chen, Y., Zhu, T., Chen, Z.: Autocog: Measuring the description-to-permission fidelity in android applications. In: Ahn, G., Yung, M., Li, N. (eds.) Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3–7, 2014. pp. 1354–1365. ACM (2014). https://doi.org/10.1145/2660267.2660287
20. Raghuramu, A., Zang, H., Chuah, C.N.: Uncovering the footprints of malicious traffic in cellular data networks. In: PAM (2015)
21. Rasthofer, S., Arzt, S., Bodden, E.: A machine-learning approach for classifying and categorizing android sources and sinks. In: NDSS (2014)
22. Ren, J., Rao, A., Lindorfer, M., Legout, A., Choffnes, D.: Recon: Revealing and controlling pii leaks in mobile network traffic. In: MobiSys (2016)
23. Sihan, Q.: Research progress on android security. Ruan Jian Xue Bao J. Softw. **1**, 27 (2016)

24. Vallée-Rai, R. Co, P., Gagnon, E., Hendren, L.J., Lam, P., Sundaresan, V.: Soot - a java bytecode optimization framework. In: MacKay, S.A., Johnson, J.H. (eds.) Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative Research, November 8–11, 1999, Mississauga, Ontario, Canada. p. 13. IBM (1999). dl.acm.org/citation.cfm?id=782008
25. Wang, Z., Li, C., Yuan, Z., Guan, Y., Xue, Y.: Droidchain: a novel android malware detection method based on behavior chains. Pervasive Mob. Comput. **32**, 3–14 (2016). https://doi.org/10.1016/j.pmcj.2016.06.018
26. Wüchner, T., Cislak, A., Ochoa, M., Pretschner, A.: Leveraging compression-based graph mining for behavior-based malware detection. IEEE Trans. Dependable Secur. Comput. **16**(1), 99–112 (2019)
27. Xia, M., Gong, L., Lyu, Y., Qi, Z., Liu, X.: Effective real-time android application auditing. In: S and P (2015)
28. Yang, W., Xiao, X., Andow, B., Li, S., Xie, T., Enck, W.: Appcontext: Differentiating malicious and benign mobile app behaviors using context. In: Bertolino, A., Canfora, G., Elbaum, S.G. (eds.) 37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16–24, 2015, Volume 1. pp. 303–313. IEEE Computer Society (2015). https://doi.org/10.1109/ICSE.2015.50
29. Zheng, X., Cai, Z.: Privacy-preserved data sharing towards multiple parties in industrial Iots. IEEE J. Sel. Areas Commun. **38**(5), 968–979 (2020). https://doi.org/10.1109/JSAC.2020.2980802
30. Zhou, Y., Jiang, X.: Dissecting android malware: Characterization and evolution. In: S and P (2012)
31. Zhou, Y., Wang, Z., Zhou, W., Jiang, X.: Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In: 19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5–8, 2012. The Internet Society (2012), www.ndss-symposium.org/ndss2012/hey-you-get-my-market-detecting-malicious-apps-official-and-alternative-android-markets