Causality-based Sensemaking of Network Traffic for Android Application Security*

Hao Zhang, Danfeng (Daphne) Yao, and Naren Ramakrishnan
Department of Computer Science, Virginia Tech
Blacksburg, VA, USA
{haozhang, danfeng, naren}@cs.vt.edu

ABSTRACT

Malicious Android applications pose serious threats to mobile security. They threaten the data confidentiality and system integrity on Android devices. Monitoring runtime activities serves as an important technique for analyzing dynamic app behaviors. We design a triggering relation model for dynamically analyzing network traffic on Android devices. Our model enables one to infer the dependency of outbound network requests from the device. We describe a new machine learning approach for discovering the dependency of network requests. These request-level dependence relations are used to detect stealthy malware activities. Malicious requests are identified due to the lack of dependency with legitimate triggers. Our prototype is evaluated on 14GB network traffic data and system logs collected from an Android tablet. Experimental results show that our solution achieves a high accuracy (99.1%) in detecting malicious requests sent from new malicious apps.

1. INTRODUCTION

Similar to PC malware, researchers found that malicious mobile applications usually fetch and run code on-the-fly without the user's knowledge [49]. Their purposes are often to stealthily collect and exfiltrate sensitive information.

Static analysis solutions typically inspect the source code, binaries or call sequences for detecting anomalies. For example, Drebin [4] extracts features including APIs, permissions, and app components to characterize and classify apps. SCSDroid [20] identifies malicious apps by extracting subsequences of system calls. However, dynamic code loading, Java reflection-based method invocation, data encryption, and self-verification of signatures are commonly seen in the malware code [11, 38]. These types of code obfuscation make static analysis based detection challenging. Dynamic analysis, as a complementary to the static analysis, detects the runtime behaviors of the malicious apps. For example, TaintDroid [10] is a solution to monitor and verify the sensitive information flows through the apps, but it is infeasible to learn the origin of sensitive data leaking (e.g., triggered by users or malicious code fetching from a remote host). Our solution provides the logic insights between the user's interaction and malicious traffic. Besides, as we treat the apps as blackbox, our solution is lightweight and low-overhead, as opposed to monitoring the apps as whitebox in TaintDroid.

In this paper, we aim at identifying malicious apps by analyzing their dynamic behaviors, specifically the network traffic generated by apps. We profile the traffic pattern of benign apps to detect malicious network requests and enhance the sensemaking process.

Sensemaking is an analysis process including the tasks of investigating the complex data, exploring their connections, and gaining insights [13]. In this paper, we propose a learning-based approach to discover *dependencies* of network requests and thus detect malicious ones. The problem of dependence discovery on network events has been introduced in [44]. The dependence of network requests is defined as the *triggering relationship* (TR). The triggering relation between two network requests r_i and r_j exists if r_j cannot be issued unless r_i is sent out first (denoted by $r_i \rightarrow r_j$). The triggering relations on app-generated requests capture how one network event is related to other events. They model the underlying logical and semantic relations among network events, which is useful for human experts' cognition, reasoning, and decision making in cyber security [12, 43].

The temporal and triggering relationship can be represented in a directed graph that is referred to as *triggering relation graph* (TRG). We refer the first request that triggers others as the *root-trigger* request. It may be caused by legitimate user activities or generated by benign apps. The TRG is composed of a list of trees, which are rooted by their *root-triggers* and can be sorted chronologically. Each tree depicts one scenario that how user requests the network resources (e.g., web-browsing, downloading a song). The legitimacy of the requests relies on the legitimacy of their root-triggers. Therefore, dependence structure of network requests offers important insights for identifying network anomalies, and thus supporting the sensemaking process.

Our Android traffic causal analysis targets at the stealthy network activities via HTTP. HTTP is the protocol chosen by most app developers to implement communication with remote servers [37] and is hardly blocked by anti-virus tools. Repackaged apps and drive-by download attacks are the common initial infection vectors [49]. After the malicious apps are installed, the requests sent to remote hosts

*This work has been supported in part by NSF grant CAREER CNS-0953638 and ARO YIP W911NF-14-1-0535.

AISec'16, October 28 2016, Vienna, Austria

DOI: http://dx.doi.org/10.1145/2996758.2996760

could leak user's information or conduct bot activities for profits. Our proposed solution can detect these types of activities without knowing any signatures of the malware, and thus applicable to detect the new (zero-day) malicious apps.

An important technical enabler of our solution is the ability to discover the triggering relation of pairs of requests. We refer to it as the *pairwise* triggering relation discovery. Pairwise relations are used to construct the *full* triggering relation graph (TRG). The graph depicts the causality of requests and allows one to quickly identify the root triggers of observed events. Our solution classifies the *root triggers* to identify malicious requests based on their dependency features. The new capability of our solution is to distinguish malicious root triggers from legitimate ones in Android network traffic. It allows us to detect the stealthy malware activities that may not be detected by existing traffic analysis methods (e.g., [9, 37]).

Dependence analysis was proposed for detecting anomalies on hosts in [44]. Our work systematically differs from theirs in the problem scope and implementations. First, we extend the triggering relation discovery problem for all types of apps on mobiles. The definition of triggering relationship is studied using the delay injection approach. In comparison, theirs only handles the browser-generated requests by using the referrer-based heuristic. Second, their solution requires a predefined whitelist to filter out the non-user triggered benign requests, while we leverage the two-stage learning-based approach to classify the benign requests.

Our contributions are summarized as follows.

- We utilize the triggering relation model to formalize the dependency of mobile network requests and traffic-generating user inputs. The discovery of triggering relations on pairwise network events enables us to construct a triggering relation graph. We present a two-stage learning-based solution and use it to classify the abnormal requests.
- We conduct our experiments on 14GB network and system data. Our results show that the triggering relations of network traffic on Android can be inferred at the accuracy of 98.2%. Without knowing any *priori* knowledge, our solution enables the detection of malicious requests sent from the newly released apps. Results confirm that we can detect 99.1% malicious requests from all malicious apps.

The significance of our work is to provide insights of the traffic dependency for Android security and demonstrate the use of structural and semantic information in reasoning about network behaviors and detecting stealthy anomalies.

Dependency analysis is a promising security approach that is capable of describing how events are related to each other and further identify the anomalies isolating from others. The pairwise relationship is designed to compare the features of two events, which are also applicable to system calls, services in a network, and events in Internet of Things (IoT). The key challenges of obtaining pairwise relationship are *i)* the features need to sufficiently characterize the logic relation between events, and *ii)* the classification needs to be scalable for voluminous data. In this paper, we present our prototype to address the problem in the context of Android network traffic. We show its feasibility in detecting zero-day network anomalies based on the dependencies of network requests.

2. MOTIVATION AND OVERVIEW

We discuss the challenges of extracting network traffic dependency in Android and the security applications of our analysis model. We also give an overview of our solution.

2.1 New Challenges of Triggering Relation Discovery on Mobile Devices

The problem of detecting malicious requests on mobile devices is challenging. We summarize the technical difficulties on the Android platform as follows.

- **Lack of referrer.** In our study, 83% network traffic sent from Android device is generated by non-browser apps. The app developers create diverse apps that communicate with the remote servers via HTTP protocol. However, the HTTP requests sent from non-browser apps rarely contain correct and meaningful referrers. Some apps attach unified referrer information for all outbound requests. Therefore, existing solutions including the referrer-based inference [36, 45] or the instrumented browser [25, 26] do not work for the general apps.
- **Diverse network traffic from apps.** Compared with browser-generated traffic, requests sent from apps have varying patterns. How to use a unified framework to infer the triggering relationship has not been studied yet. Processing massive data also demands scalable solutions.
- **Automatic notifications and updates.** Mobile devices constantly alert users with notifications, resulting in plentiful network requests in the background. Properly handling the notifications is crucial in achieving a high accuracy and low false positives. The tradition whitelisting created by human experts is no longer adequate. Hence, to automatically generate whitelisting using learning-based methods is desirable.

We infer *triggering relations* by analyzing how the delays are propagated among the network requests of an app. As a result, we discover the dependency without requiring the referrer information. We also design a learning-based approach to classify the triggering relationship of requests, which is scalable and suitable for all types of Android apps. We build the TRG based on classification results. The graph offers rich structural and context-aware features which are further used to identify malicious requests and aid the sensemaking process. The whitelisting generated from machine learning outcomes saves human efforts and improves the detection accuracy.

2.2 Security Applications of Our Detection

Our model detects the malicious network behaviors by discovering the dependency of network packets on Android devices. The behaviors of malware include the unauthorized network activities, e.g., stealthy outbound requests without user's awareness, piggyback requests containing malicious code, and other types of out-of-order requests. These behaviors existing in a wide range of malware families cause sensitive information leaking and system abusing. Malicious apps may not immediately trigger its malicious behaviors, because the timing or logic bombs (e.g., debugger-checks, rooting-checks) could delay the triggering. We list some examples of our security applications as follows.

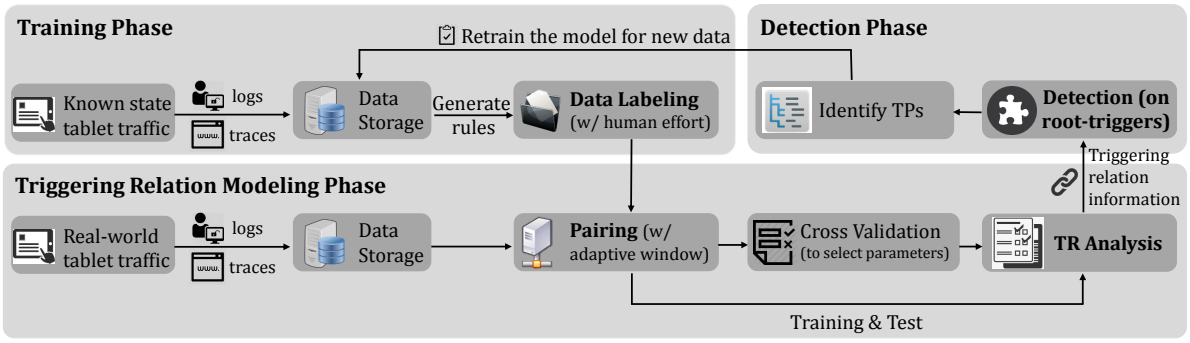


Figure 1: The workflow of our triggering relation discovery based malware detection.

- **Repackaged apps** refer to the malicious apps created by repackaging the existing benign ones [49]. They are known to contain malign payloads that may cause malicious requests, which are usually generated without user's consent, e.g., `AnserverBot` malware family.
- **Drive-by download apps** fetch malicious payloads at runtime. In our context, drive-by downloading refers to the unintentional malicious download. For example, it may lure users and stealthily install a new malicious app or create a shortcut icon to some malicious or advertisement sites, e.g., `com.Punda.Free`.
- **Android bots**, being controlled by bot masters through the network, can be used to conducted remote attacks. Android bots are not exclusive to the other two categories. Their behaviors often include the stealthy network communication to remote command and control (C&C) servers.

The benign apps sometimes send out outbound requests without user's awareness. As an open problem, these "gray" requests are difficult to be classified, because they may be sent with user's consents. Our model labels these requests as *suspicious*.

We target at the malicious apps running on the user-space, therefore malware with root access is not in our threat model. We trust the system logs and keylogger apps to infer the triggering relationship and root-triggers.

2.3 Overview of Our Approach

A triggering relation graph is composed of network events and the edges that link them. The problem of *triggering relation discovery* on a set of network events can be solved by inferring the triggering relations of pairs of events, which is defined as the *pairwise triggering relation*. Given a sequence of network requests $\mathbb{R} = \{r_1, r_2, \ldots, r_n\}$, the purpose of *pairwise comparison* is to generate a set of pairs $\mathbb{P} = \{P(r_i, r_j)\}$, where $1 \leq i < j \leq n$, r_i and r_j have a *high* likelihood to have a triggering relation. The pairwise comparison method has been proposed to solve the general relation discovery problem [16, 17, 44]. In this work, we further advance it in the Android context by improving the pairing efficiency and using the multinomial classification.

Utilizing predicted results from the classification, we build the TRG $\mathbb{G} = \{T_1, T_2, \ldots, T_m\}$, each T_i is a tree rooted at request r_{t_i}. We classify the root-triggers $\mathbb{R}_\mathbb{T} = \{r_{t_i}\}$ $(1 \leq i \leq m)$, based on the extracted features from TRG $\mathbb{G}$. The *root-trigger* request r_{t_i} determines the legitimacy of each tree, i.e., if it is generated from a benign app, the entire tree T_i is legitimate. The anomalies are the requests of trees that are not triggered by users, nor belong to the auto-generated notifications/updates from benign apps.

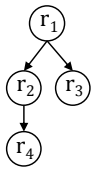
We show an overview of the workflow in Figure 1. Our approach requires the data labeling (in training phase) based on the requests $\mathbb{R}$, details of which are given in §5.

The main operations in our analysis are pairing, TR analysis, and detection. The first two operations are designed to model the triggering relations and build a TRG. Both TR analysis and detection include training and testing, as the standard operations for the machine learning approach. Specifically, we adopt a multinomial classification in TR analysis and use a binary classifier in detection.

A) Pairing takes the $\mathbb{R}$ as an input and outputs a set of paired requests $\mathbb{P} = \{P(r_i, r_j)\}$. (§3.1)
B) TR analysis takes the predicted pairs P as inputs and outputs a constructed TRG $\mathbb{G}$. (§3.2)
C) Detection takes a TRG $\mathbb{G}$ and its root-trigger set $\mathbb{R}_\mathbb{T}$ as inputs. It extracts features of $\mathbb{R}_\mathbb{T}$ and builds classifications to predict the legitimacy of root-triggers. (§4)

3. TRIGGERING RELATION MODELING

We describe our triggering relation modeling in this section. The goal of this modeling is to build a complete and accurate TRG for the later detection phase.



(a)

#	Paired items	Relation labels
(1)	$P(r_1, r_2)$	`Parent-child`
(2)	$P(r_1, r_3)$	`Parent-child`
(3)	$P(r_1, r_4)$	`Same-tree`
(4)	$P(r_2, r_3)$	`Sibling`
(5)	$P(r_2, r_4)$	`Parent-child`
(6)	$P(r_3, r_4)$	`Same-tree`

(b)

Figure 2: A sample tree (a) and its pairing results (b). In (a), nodes and edges denote the network requests and their triggering relations respectively.

We introduce the multi-class to describe the structural relations of two requests r_i and r_j: (1) `parent-child`: $r_i \rightarrow r_j$, (2) `sibling`: r_i and r_j have the same parent (i.e., $r_k \rightarrow r_i$ and $r_k \rightarrow r_j$), (3) `same-tree`: r_i and r_j are situated on the same tree ($r_i \in T_i$ and $r_j \in T_i$) without `parent-child` and `sibling` relations, and (4) `no-relation`: r_i and r_j on

two different trees. The quad-class offers rich information to identify the relations for requests in a TRG. Figure 2 illustrates how we generate pairs from a sample tree based on quad-class labeling and how these relations determine the pattern of a tree in a TRG.

3.1 Pairwise Feature Extraction

The pairing operation is performed on any two requests r_i and r_j in $\mathbb{R}$ for whose time difference is less than a threshold τ. Let the request r_i have p attributes and each is denoted as $r_i.attr$, $attr \in \{\texttt{time}, \texttt{IP}, \texttt{port}, \texttt{host}, ..., \texttt{URL}\}$. Then, the pairing operation is performed on every attribute of the requests. The pairwise comparison can be written as $P(r_i, r_j) = \{f_1(a_i.\texttt{time}, a_j.\texttt{time}), \ldots, f_p(a_i.\texttt{URL}, a_j.\texttt{URL})\}$, where f_k is the pairing function for the k-th attribute. Shown in Table 1, different types of attributes require different pairing strategies (f_k).

Attribute	Pairwise Function (f_k)	Example(s)
Numerical	Compute the difference.	Timestamp.
Nominal	(Sub)string/equality test.	Protocol.
String	String similarity comparison.	Host, URL.
Composite	Comparison of sub-attributes.	IP address.

Table 1: Pairing functions used for different attribute types.

Adaptive Pairing. To avoid the $O(n^2)$ complexity of pairing n requests, we only analyze network requests whose timestamps are within a small time window (threshold). Let $d_{(i,j)}$ be the time difference between a pair of requests r_i and r_j that have triggering relation. We obtain the 3-day moving average of $d_{(i,j)}$ as the μ and its corresponding σ^2 in the learning phase, and calculate the threshold using the *Chebyshev's inequality* ($\tau = \mu + 10 * \sigma$). The probability of the difference between $d_{(i,j)}$ and μ exceeds $10*\sigma$ is no greater than 1%. The Chebyshev's inequality offers an upper bound on the probability of $d_{(i,j)}$, regardless of its distribution.

3.2 Quad-class Classification and TRG Construction

In the TR analysis operation, we train classifiers to predict triggering relations on pairs $\mathbb{P} = \{P(i,j)|i < j\}$. The predicted results are in the form of $(r_i, r_j) \rightarrow l_{ij}$, where l_{ij} is one of the relations in the quad-class.

We analyze three types of relationship: `parent-child`, `sibling`, and `same-tree` and assign weights w_p, w_s, and w_t to them, respectively.[1] The purpose of the weights is to construct the TRG based on the classified pairwise triggering relations. Given r_i and r_j, we define $w_{(r_i,r_j)}$ in Equation 1 as a score of the pair (r_i, r_j), which is the summation of the weights that indicate their triggering relationship.

$$w_{(r_i,r_j)} = \sum w \begin{cases} w_p, & \text{if } (r_i, r_j) \rightarrow \texttt{parent-child}. \\ w_s, & \text{if } (r_i, r_k) \rightarrow \texttt{parent-child} \\ & \quad \text{and } (r_k, r_j) \rightarrow \texttt{sibling}. \\ w_t, & \text{if } (r_i, r_j) \rightarrow \texttt{same-tree}. \end{cases} \quad (1)$$

We show the procedure to calculate the weight for the classified pair (r_i, r_j) in Algorithm 1. The inputs of Algorithm 1 are the predicted results $\mathbb{P}$ and the weights (w_p, w_s, and w_t). In lines 6-9, there may be multiple triggering candidates (parents) for one request, thus we update the weights for each node of its parents (denoted by Parent(r_i)). The output of Algorithm 1 is a dictionary of keys $\mathbb{D}$, which contains the calculated weights of pairs that (may) have triggering relationship.

Algorithm 1 Triggering Relation Weights Calculation

Input: Requests $\mathbb{R} = \{r_1, r_2, \ldots, r_n\}$, classified pairs $\mathbb{P} = \{(r_i, r_j) \rightarrow p_{ij}\}$ $(1 \leq i < j \leq n)$, and weights w_p, w_s, w_t.
Output: A dictionary of keys $\mathbb{D} = \{(r_i, r_j) = w_{(r_i,r_j)}\}$, where r_i and r_j may have triggering relationship according to the classification, and its weight is $w_{(r_i,r_j)}$.

```
1: define 𝔻 to store the weight w_(r_i,r_j) for the edge r_i → r_j
2: for each pair (r_i, r_j) → p_ij ∈ ℙ do
3:    if p_ij = parent-child relation then
4:       weight w_(r_i,r_j) += w_p, and update w_(r_i,r_j) in 𝔻
5:    else if p_ij = sibling relation then
6:       Parent(r_i) ← r_i's trigger(s) according to ℙ
7:       for r_p in Parent(r_i) do
8:          weight w_(r_p,r_j) += w_s, and update w_(r_i,r_j) in 𝔻
9:       end for
10:   else if p_ij = same-tree relation then
11:      Root(r_i) ← r_i's root request(s) according to ℙ
12:      for each r_r in Root(r_i) do
13:         weight w_(r_r,r_j) += w_t, and update w_(r_i,r_j) in 𝔻
14:      end for
15:   end if
16: end for
17: return 𝔻
```

Building a TRG from $\mathbb{D}$ is equivalent to finding the *maximal spanning tree* in a graph. In our approach, we adopt Prim's algorithm, which suggests starting from one vertex and selects the highest weight from the adjacent neighbors.[2] Prim's algorithm fits our needs of starting from known root-triggers. The prerequisite in Prim's algorithm is the weight calculation. During the process of selecting the *largest* weight in Prim's algorithm, there may be equal weights in $\mathbb{D}$. We design rules to select the best triggering parent for a given request from a group of candidates whose weights are equal, i.e., $w_{(r_i,r_k)} = w_{(r_j,r_k)}$. The following rules are presented in the order of the precedence.

Rule 1) If $r_i \in T_i$, $r_j \in T_j$, and $r_{t_i}.\texttt{time} > r_{t_j}.\texttt{time}$, then $r_i \rightarrow r_k$. (The rule is in favor of the request on the most recent dependency tree.)

Rule 2) If $r_i, r_j \in T_i$, $r_i.\texttt{level} < r_j.\texttt{level}$, then $r_i \rightarrow r_k$. (The rule is in favor of the request on the upper level of the tree.)

Rule 3) If $r_i, r_j \in T_i$, $r_i.\texttt{level} = r_j.\texttt{level}$, $r_i.\texttt{time} > r_j.\texttt{time}$, then $r_i \rightarrow r_k$. (The rule is in favor of the most recent requests on the same level of the tree.)

We make these design choices based on the empirical study and domain knowledge of the Android network traffic. Prim's algorithm outputs the TRG $\mathbb{G}$ by picking the best candidate trigger from the equal-weight candidates. As the classification results may contain redundant or conflicting information, it is indispensable to solve the tie-breaking of weights.

[1] In our design, the weights are calculated using multivariable linear regression.

[2] Prim's algorithm is usually used to find the minimal spanning tree. However, in our scenario, the larger weights mean the higher possibility to have triggering relationship. Therefore, we find the maximal spanning tree for $\mathbb{G}$. The same situation applies when using Kruskal's algorithm.

#	Predicted results	$w_{(r_i,r_j)}$	Tie-break	Built TR
(1)	$(r_1, r_2) \rightarrow$ p-c	$w_p + w_s$	—	$r_1 \rightarrow r_2$
(2)	$(r_1, r_3) \rightarrow$ p-c	$w_p + w_s$	No tie.	$r_1 \rightarrow r_3$
(3)	$(r_2, r_3) \rightarrow$ sib	0		
(4)	$(r_3, r_4) \rightarrow$ p-c	w_p	—	$r_3 \rightarrow r_4$
(5)	$(r_3, r_5) \rightarrow$ p-c	w_p	r_3 is on upper level than r_4.	$r_3 \rightarrow r_5$
(6)	$(r_4, r_5) \rightarrow$ p-c	w_p		

Table 2: The triggering relations are built from the predicted results. In predicted results, p-c and sib denote the parent-child and sibling relations respectively. $w_(r_i, r_j)$ shows the calculated weights using Algorithm 1. The tie-break column shows the rule to resolve ties. Note that the weight $w_p + w_s$ in (1) and (2) is due to the predicted result from (3).

We illustrate our TRG construction using an example in Table 2. First, we calculate the weights by parsing the predicted results. According to formulas (1-3), we observe that r_1 has parent-child relation with r_2 and r_3. Furthermore, the knowledge of formula (3) is redundant, only formulas (1-2) are enough to deduce the triggering relations. In the latter formulas (4-6), the parent of request r_5 has two candidates r_3 and r_4 based on the calculated weights. We resolve this conflict by using our tie-breaking rules.

The complexity of Algorithms 1 is $O(|\mathbb{P}|^2)$. Due to the sparse triggering relations in a TRG (i.e., $|node| \approx |edge|$), the complexity of Prim's algorithm is $O(n * log(n))$, where n is the number of requests in $\mathbb{R}$.

Security discussion. The threshold τ in §3.1 determines the efficiency and accuracy in pairing operation. A large τ causes unnecessary pairs between two irrelevant requests, resulting in unbalanced datasets, which negatively impacts the inference accuracy. A small τ leads to insufficient pairs, resulting in a fragmented TRG. For example, synchronous requests (e.g., AJAX), retrieved much later than normal requests, could be regarded as *vagabond* requests, and causes false positives in detection. Therefore, the adaptive window size can improve the pairing accuracy by adjusting the τ according to the recent user's behaviors.

The TRG construction relies on the weights calculation algorithm and tie-breaking rules. Malicious requests tend to hide themselves by building dependencies to benign requests. Wrong triggering relationship results in the wrong root-trigger, which causes the false positives of malware detection. Our algorithm and rules build correct TRs by exhausting all predicted weights and *reinforce* the sensemaking based on the parent-child and slinging relationships.

4. DETECTION ON ROOT-TRIGGERS

In the detection phase, we identify the root triggers $\mathbb{R}_\mathbb{T}$ for each network request based on the constructed TRG $\mathbb{G} = \{T_1, T_2, \ldots, T_m\}$. Most requests are directly or indirectly triggered by the legitimate user's inputs ($\mathbb{U}_\mathbb{T}$). However, there may exist requests that cannot be linked with any user's activities, e.g., notification requests from benign apps ($\mathbb{A}_\mathbb{T}$), malicious requests from the malware ($\mathbb{M}_\mathbb{T}$). Therefore, $\mathbb{R}_\mathbb{T}$ can be formalized as the disjoint sets of the three.

$$\mathbb{R}_\mathbb{T} = \{\mathbb{U}_\mathbb{T} \cup \mathbb{A}_\mathbb{T} \cup \mathbb{M}_\mathbb{T}\} \quad (2)$$

In our design, we use binary classifiers to detect the malicious requests ($\mathbb{M}_\mathbb{T}$) in two steps: (1) In a clean/labeled dataset, we distinguish the notifications or update requests ($\mathbb{A}_\mathbb{T}$) from other benign ones ($\mathbb{U}_\mathbb{T}$). A whitelisting can be generated from the learning outcomes ($\mathbb{A}_\mathbb{T}$). (2) In a real-world dataset, we filter out the requests that belong to the whitelist, and then differentiate the malicious requests ($\mathbb{M}_\mathbb{T}$) from the benign ones ($\mathbb{U}_\mathbb{T}$).

We describe the features extraction for root-triggers in §4.1 and the root-trigger identification/labeling in §4.2.

4.1 Feature Extraction for Detection

For a given root-trigger $r_t \in \mathbb{R}_\mathbb{T}$, we extract features as described in Table 3. We train and classify the data using common supervised machine learning classifiers (e.g., random forest, logistic regression). The output of the classification in detection is the partition of $\mathbb{R}_\mathbb{T}$.

Type	Feature Definition / Description
$\mathcal{F}_1$	$\mathcal{F}_1$ describes the number of similar requests in the previous user-triggered trees. $f(r_t) = \|r\|$, where $r \in \mathbb{U}_\mathbb{T}$, r_t.time $- r$.time $\leq \tau^*$ †, $f_u(a_r$.attr$, a_{r_t}$.attr$) \leq \tau_u$ ‡
$\mathcal{F}_2$	$\mathcal{F}_2$ describes the number of similar requests in the previous non-user-triggered trees. $f(r_t) = \|r\|$, where $r \in \{\mathbb{A}_\mathbb{T} \cup \mathbb{M}_\mathbb{T}\}$, r_t.time $- r$.time $\leq \tau^*$ † $f_u(a_r$.attr$, a_{r_t}$.attr$) \leq \tau_u$ ‡
$\mathcal{F}_3$	$\mathcal{F}_3$ describes the *statistics* from the previous user-triggered trees in $\mathbb{G}$. $f(r_t) = \|r\|$, where $r \in \mathbb{U}_\mathbb{T}$, r_t.time $- r$.time $\leq \tau^*$ †
$\mathcal{F}_4$	$\mathcal{F}_4$ describes the *temporal* relation of r_t and previous events. E.g., u_p is the most recent user event prior to r_t, then $f(r_t) = \|r_t$.time $- u_p$.time$\|$.

† τ^* is a time threshold used in feature extraction (not the τ for pairing). τ^* could be 1 second, 1 minute, 1 hour, 12 hours and etc.
‡ f_u is the function to compute the similarity between attributes. a_r.attr denotes a request r's (semantic) features, e.g., Host, Referrer, request URL and destination IP.

Table 3: Features extracted for the detection operation.

In this operation, the dependency-based feature set $\mathcal{F}_\mathcal{D} = \{\mathcal{F}_1, \mathcal{F}_2, \mathcal{F}_3, \mathcal{F}_4\}$ contain the rich semantic and structural information of $\mathbb{G}$ and aid the sensemaking in detecting anomalies. The design of $\mathcal{F}_\mathcal{D}$ features is based on our empirical observations on malicious and benign requests. $\mathcal{F}_1$ features are extracted to characterize the frequently visit websites and some benign requests. Because these websites can be cached by apps to save the bandwidth, therefore, the newly generated requests may not contain referrers. $\mathcal{F}_2$ features are used to characterize the auto-generated benign requests. Because Android apps may pull notifications from the remote servers periodically, the IP, host and referrer information of the notifications have usually seen before. $\mathcal{F}_3$ and $\mathcal{F}_4$ features are designed to quantify the pattern of $\mathbb{U}_\mathbb{T}$ in $\mathbb{G}$. The heuristics behind these features is that: *1)* requests are usually sent during the highly interactive periods; *2)* user-trigger requests are inclined to be sent immediately after the user's actions, while the idle between two root-trigger requests is a noticeable long interval.

4.2 Root-trigger Identification

To find the root-trigger is an essential task in identifying the normalcy of the requests, as the requests triggered by the root remain the same legitimacy in our model.

In our solution, we take advantage of the Android system debug logs (Logcat) and the KidLogger app [18] to accurately infer the root-triggers. The Logcat serves as a default means

to view the system debug outputs, and thus can be used to identify if the request is triggered by user's requests or by system/software updates. We focus on the logging levels `information` and `debug`, as they provide the caller app name and its status information. The `KidLogger` app is designed to monitor and track on the Android system as a parental control tool. We obtain the following information by processing the `Logcat` and `KidLogger` data.

- We identify the notification requests ($\mathbb{A}_\mathbb{T}$) by parsing the `ActivityManager` logs from `Logcat`. The `ActivityManager` is a major class that interacts with overall activities running on the Android device. It provides the information about when a particular service is awake and invokes network connections.
- When users interact with an app, `debug` logs reveal details on when and how the app responses. Integrated with the information from `Kidlogger`, we determine the foreground apps. Then, we correlate the root-trigger request ($\mathbb{U}_\mathbb{T}$) with the closest user's activity prior to it.

4.3 Security Analysis and Limitations

Our dependence analysis model aims at analyzing the network activities of Android devices, while it has certain limitations and possible evasions. Our discussions are as follows.

- *Non-HTTP protocol.* Our solution targets at the stealthy network activities via HTTP, because HTTP is the protocol of choice for most app developers to implement communication with remote servers [37] and is hardly blocked by anti-virus tools. By combining other security tools and policies, one can set up firewall rules for apps that use other protocols.
- *Data authenticity.* As an anti-analysis technique, attackers may forge the user and system events so that the malicious requests have triggers. Advanced solutions (e.g., [2, 24, 28]) can be used to ensure the authenticity of the user inputs and system events.
- *Rooting device.* Our prototype requires rooted mobile devices and the monitored apps run in the user-space. This requirement may reduce the usability for non-tech savvy users.
- *Encrypted traffic.* For the encrypted network activities via `HTTPS`, one might adopt an authorized proxy to decrypt and analyze the traffic. This method requires the device to install a self-signed CA to encrypt the network packets [29].

5. LABELING TRIGGERING RELATIONS ON MOBILE DATA

A technical challenge in our work is the lack of readily available labeled data, i.e., network events with labeled triggering relations. We spend a substantial amount of effort designing methods to label the triggering relation of general apps. Our approach is based on the timing perturbation. It delays one request and see if others would be affected. The rationale behind this approach is that the delay of an individual request will be propagated to the requests that are triggered by it.

We elaborate the approach to label the triggering relations (data labeling operation in Figure 1). This process is time-consuming and needs human efforts. Therefore, the time perturbation method is suitable for generating rules on small-scale data, which fits the needs of labeling and training purposes. In comparison, our learning-based approach described in §3 can be used for analyzing and testing on large-scale data.

Given two requests r_i and r_j, where r_i triggers r_j, the time difference $d_{(i,j)}$ can be decomposed into several components: (1) DNS query time to obtain the IP address, (2) TCP connection time and one RTT (round-trip time), (3) network delays (e.g. queuing delay), and (4) processing time at both server and client sides. Any of the factors may impact the discovery of causality for non-browser apps. We use the time sequence of outbound packets to estimate the triggering relations by creating temporal perturbations. To distinguish the artificially-generated delays from other factors and random noise, *we take advantage the statistical tests to achieve high levels of confidence.* Our dependence analysis and Rippler [42] operate at different levels of granularity: we detect dependencies at the level of outbound requests, while Rippler does so at the level of network services.

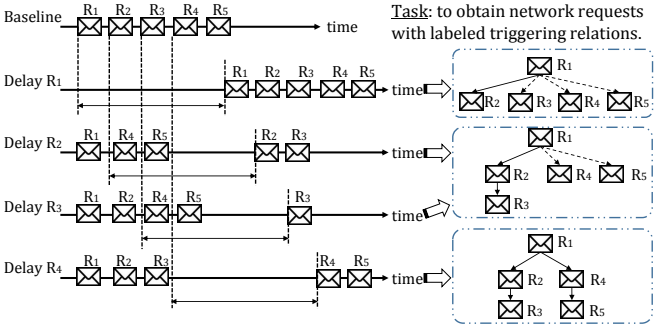


Figure 3: We label training data using the timing perturbation and incrementally build the TRG. Dashed lines represent the intermediate results and solid lines represent the finalized triggering relationship.

Our dependency inference consists of three steps. In step (i), we record $\mathbb{R} = \{r_1, r_2, \ldots, r_n\}$, a series of network traffic generated from one app. The list $\mathbb{R}$ serves as a baseline, where the elements in $\mathbb{R}$ are ordered by their timestamps. In step (ii), we re-act the scenario and delay t milliseconds for each object in $\mathbb{R}$. This step is based on the fact that r_j is a dependent of r_i, if and only if r_j is not loaded until r_i is fetched. For step (i) and each iteration in step (ii), we perform m times to obtain m observations.

In step (iii), we apply statistical tests to the m observations for identifying the triggering relations. We define $X_{(j,i)}$ as the variable for the outbound timestamp of j-th packet when the i-th packet is delayed ($i = 0$, for the baseline in Figure 3). We denote $\mu_{(j,i)}$ and $\sigma_{(j,i)}$ as the mean and SD of $X_{(j,i)}$. Because the delay injection is independent of each iteration and the samples are from a normal distribution, we adopt the two-sample pooled t-test to figure out whether the j-th packet is affected by the i-th packet [30]. The null hypothesis is $H_0 : \mu_{(j,0)} = \mu_{(j,i)}$. We report the j-th packet is a dependent of the i-th packet, once the statistical test rejects H_0. The dependency graph can be incrementally built based on the finding of dependency from temporal perturbations, as shown in Figure 3.

We assume the requests $\mathbb{R}$ are not changed during our study. However, we do observe some requests from an app

may slightly differ between the iterations. Therefore, we compute the edit distance and regard two URLs identical if the similarity is less than a threshold. Furthermore, we confirm such a discrepancy rarely transforms the TRG to a different one.

6. EVALUATION AND RESULTS

We implement a prototype to analyze the real-world network traffic on an Android device. Based on the collected data, we conducted extensive tests on our proposed dependence analysis solution. The questions we seek to answer are:

- How accurate is the prediction for TRs? (§6.4, §6.6)
- How accurate is the classification for detecting the notifications and malicious requests? (§6.5, §6.7)
- Can it detect *new* malicious Android apps? (§6.7)
- Is our approach scalable for analyzing real-world data? What is the performance of our approach? (§6.8)

6.1 Implementation

Our prototype implements all parts of the dependence discovery and detection system. The data are processed on a Linux machine with Intel i5-3320 and 16GB RAM.

Data collection and preprocessing. We root the device and install `PythonForAndroid` to run Python code. We use `tcpdump` to sniff HTTP requests whose headers contain valid `GET` or `POST` request. To log the user's events, we continuously monitor the files from `/data/input/`. We use `ps` and `netstat` commands to collect the process information and associate the network requests with a process. The collection and preprocessing are implemented with 1,400 lines of code in Python and Bash.

Pairing and classification. We implemented the pairing, TR analysis and feature extraction for detection in Java using the Weka library with 1,200 lines of code. To build the TRG, we wrote 800 lines code in Python.

6.2 Experimental Dataset

The experimental data are obtained from a Nexus 7 tablet. The device, equipped with our data collecting code, is given to a graduate student of a university for regular daily use. The device comes with 28 bundled apps, and then we installed 36 popular apps via the official Android store [3]. We first collected the network and system logs for a continuous 72-day period. These 64 apps are regarded as benign and do not conduct malicious activities. The selected popular apps cover a wide range of categories, including news, social network, shopping, games, entertainment, etc.

Thereafter, we installed 24 malicious apps and collected data for 22 days. The malicious apps include repackaged apps that covertly fetch advertisement requests, drive-by download apps that install other apps when running, and spy apps that keep sending out host and user's sensitive information (e.g., bookkeeping). The total dataset is 14 GB, including 1.4 million user input records, 1.8 million system logs, and 565 thousand network events after consolidation.

Experiment setup. We conduct two experiments.

- **Dataset I**: We apply our approach on clean data only. The purpose is to evaluate its ability to recognize the auto-generated notifications and updates from other benign requests.
- **Dataset II**: With the full dataset, we randomly insert the malicious data (by day) into the benign data and evaluate the effectiveness of our approach to detect malicious requests.

Dependency labeling of training data. We infer the triggering relation of browser-generated HTTP requests, according to a referrer-based method, which achieves nearly 100% accuracy [44]. To label the dependency of network traffic that comes from non-browser apps, we first installed the apps in the Android emulator.[3] We set up a proxy and route the packets to a Linux host as a single point to control the outbound traffic from the emulator. On the proxy, we use `netem` to delay the traversing packets by adding the queuing delays, according to the scheme mentioned in §5. We run the timing perturbation approach and generate heuristic rules to label all the triggering relations. Examples of rules and learned patterns are listed in Table 4, for given two requests r_i and r_j (r_i proceeds r_j), and a threshold τ.

With our hand-tuned rules, we discover the triggering relation for 86.1% of HTTP requests. The rest of the requests are labeled with the timing perturbation method.

6.3 Accuracy and Security Metrics

We apply three classifiers[4], namely random forest (`R-F`), C4.5 algorithm (`C45`), and logistic regression (`LOG`) in both triggering relation modeling and detection phases. The operations are evaluated using the following metrics.

- **Pairing coverage** (PC). This metric is to evaluate if the (potential) triggering relation $r_i \rightarrow r_j$ $(i < j)$ are paired using the adaptive pairwise window (size $= \tau$).

$$PC = \frac{|\forall r_i, r_j \in \mathbb{R}, \text{s.t. } r_i \rightarrow r_j, d_{(i,j)} \leq \tau|}{|\forall r_i, r_j \in \mathbb{R}, \text{s.t. } r_i \rightarrow r_j|} \quad (3)$$

- **TR accuracy** (TRA): For each request $r \in \mathbb{R}$, we calculate the ratio of the number of requests correctly identified its trigger to the total number of requests, based on the ground truth. The metric evaluates the effectiveness of the classifiers.

$$TRA = \frac{|\forall r_j, \exists r_i \in \mathbb{G}, \text{s.t. } r_i \rightarrow r_j|}{|\forall r_j, \exists r_i \in \mathbb{R}, \text{s.t. } r_i \rightarrow r_j|} \quad (4)$$

- **Precision, Recall, F-score**: False negatives (FN) are the malicious requests being detected as benign ones. False positives (FP) are the benign requests being detected as malicious ones. We use the conventional metrics: *i)* false positive rate (FPR) and false negative rate (FNR), *ii)* precision and recall, and *iii)* F-score (the *harmonic mean* of precision and recall).

6.4 TR Analysis on Dataset I

Pairing. We calculate the adaptive pairing windows size for dataset I using the 3-day moving average value and standard deviation (SD). The pairing window size for each sampling day is shown in Figure 4. We observe that the average of the time difference $d_{(i,j)}$ is a stable value, though the SD fluctuates a lot. We confirm that the pairing coverage is nearly 100% and only a few dependent pairs are missed due to their extremely long delays. The high pairing coverage is guaranteed as explained in §3.1.

[3]The emulator is used for labeling the triggering relations. All other experiments are based on the real device.

[4]Classifiers are selected based on the 10-fold cross validation.

No.	Rules (to decide that $r_i \rightarrow r_j$, i.e., r_i and r_j have triggering relationship.)	Related Apps
1	r_i.time - r_j.time $\leq \tau \wedge (r_i.\text{PID} = r_j.\text{PID} \vee r_i.\text{program} = r_j.\text{program}) \wedge r_i.\text{host} = r_j.\text{referrer}$.	Traffic sent from browser apps.
2	r_i.time - r_j.time $\leq \tau \wedge r_i.\text{program} = r_j.\text{program} \wedge r_j.\text{type} =$ `POST` $\wedge\ r_i.\text{host} = r_j.\text{host}$ $\wedge\ r_j.\text{referrer} = \textit{null}$.	Traffic sent from news, social, or education apps.
3	r_i.time - r_j.time $\leq \tau \wedge r_i.\text{program} = r_j.\text{program} \wedge$ `func_sim`$(r_i.\text{host}, r_j.\text{host}) = \text{true}$ $\wedge\ (r_i.\text{referrer} = r_j.\text{referrer} \vee r_j.\text{referrer} = \textit{null})$.	Traffic sent from news, media, or education apps.
4	r_i.time - r_j.time $\leq \tau \wedge r_i.\text{program} = r_j.\text{program} \wedge r_i.\text{destinationIP} = r_j.\text{destinationIP}$ $\wedge$ `func_sim`$(r_i.\text{host}, r_j.\text{host}) = \text{true} \wedge r_i.\text{program} = r_j.\text{XRequestWith}$.	Traffic sent from news or media apps (using `AJAX`).

Table 4: The examples of summarized triggering relation rules using the time injection method. The attributes of request are extracted from packet headers and system events. `func_sim` is used to compare the similarity of two string fields.

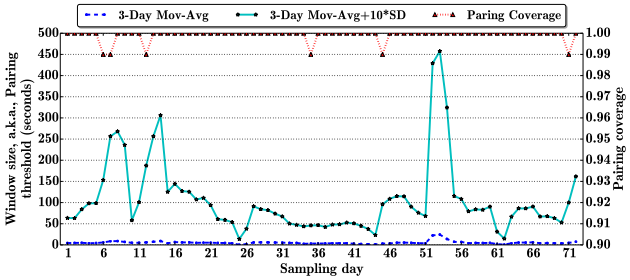


Figure 4: The adaptive pairing window size and its coverage rate for each sampling day on dataset I.

We find that `parent-child`, `sibling`, and `same-tree` relations account for 3%, 19% and 15% of the total number of pairs.[5] The rest of the pairs (62%) are labeled as `no-relation`. We spot that the pairing window size is increased from the 51st to 53rd days. This prolonged window is due to a plural of `AJAX` calls from a browser app (`Chrome`).

Triggering Relation Modeling. We vary the sizes of training and test data using $i \in [1, 5, 10, 15, 20]$, i.e., we train the data for i continuous day(s) and test the data in the following i continuous day(s). Averaged results are reported until the data on the last day of dataset I is tested. We show the TR accuracy rates in Figure 5.

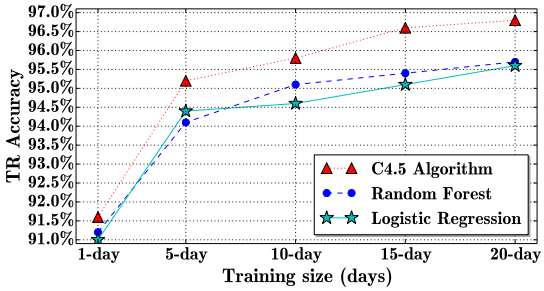


Figure 5: The TR accuracy of dataset I.

All classifiers predict better with the increasing of the training sizes. The C4.5 algorithm outperforms the other two in all settings and it achieves the best TR accuracy at 96.8%.

The wrongly predicted cases mainly come from the browser apps. When users view multiple similar webpages (e.g. price comparison on shopping sites), similar requests are sent to the same host from different tabs. These requests are likely to be predicted to wrong triggers, because users may switch the tabs frequently and shortly. As this only happens on browser apps, a finer-granularity solution on tab-level events can reduce the wrong predictions.

6.5 Recognition of Auto-generated Traffic

The purpose of this experiment is to evaluate our ability to recognize the auto-generated notifications and updates sent from benign apps. This experiment helps us further reduce false positive (false alerts). We evaluate the classification based on the results of running the C4.5 algorithm in §6.4.

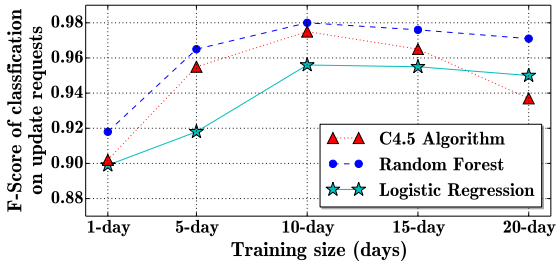


Figure 6: The F-scores of classifying root-trigger requests on dataset I.

In Figure 6, we plot the F-score of the three classifiers on classifying the update requests. A peak is observed in all classifiers, and the F-score reaches its highest value using 10-day training size. This phenomenon is due to the constant changes of app and user behaviors. Therefore, the size of training windows is a trade-off between predict accuracy and coverage. The prolonged training windows are not effective in characterizing the root-triggers from recently active apps.

Show in Figure 6, random forest classifier outperforms the other two classifiers and it maintains a relatively high and steady accuracy.

Whitelist generation. We generate a list of HTTP entries that belong to the auto-generated notifications and benign update requests. The entries are presented as a 4-tuple (host, destination IP, request type and app name[6]). We label 28 distinct entries in the training data. These entries belong to 6 apps. Random forest classifier identifies 41 new entries from the testing data (52-day data) that can be added to the whitelist. In the later detection, the re-

[5]The number of `sibling` relation is greater than that of `same-tree`, which is due to the shallow tree structure formed by the Android network requests.

[6]The request type specifies the `GET` or `POST` requests. The app name is identified using the PID and PPID (parent PID) information obtained from `netstat` and `ps` commands.

quests that match the whitelisted entires can be marked as benign to reduce the false positives.

FP and FN analysis. For all classifiers, the false positive rate ranges from 1.4% to 2.9%. False positives are mostly due to that the benign apps occasionally send out statistical data for advertisement. For example, a `social` app sends `POST` requests to `www.google-analytics.com/collect`.

The update requests sent to a popular third party host may be classified as false negative. For example, a `weather` app sometimes fetches metadata files (`JSON` or `XML`) from `s3.amazonaws.com`, which counts as FNs.

6.6 TR Analysis on Dataset II

We run the TR Analysis on the dataset II. All TR accuracy results demonstrate the same pattern as shown in Figure 5. All TR accuracy results are above 96.0% for training data that are greater than 5 days. The differences of TR accuracy are less than 1.0% for three classifiers when using 15-day and 20-day training data. The results converge to a high accuracy, which shows it is feasible to test real-world (mixed benign and malicious) Android network traffic using a comparable small training dataset.

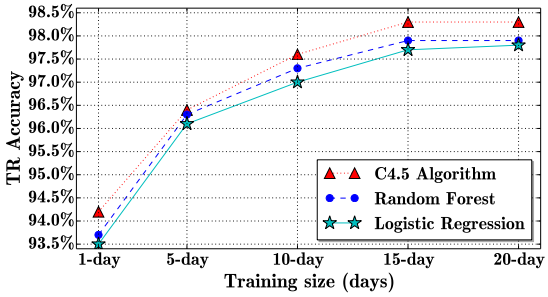


Figure 7: The TR accuracy of dataset II.

6.7 Detection of Malicious Requests

The purpose of this experiment is to evaluate the effectiveness of constructed triggering relation graph in detecting malware network activities. We first remove the notifications and update requests based on the whitelist in §6.5. Thereafter, we run the binary classifiers on the root-triggers of the constructed TRG (training size is 10-day). Shown in Table 5, both tree-based classifiers (`C45` and `R-F`) achieve the better predict accuracy (F-scores) than the log regression.

Classifier	FPR	FNR	Precision	Recall	F-score
C45	5.2%	0.6%	0.991	0.994	0.993
R-F	0.7%	3.6%	0.999	0.964	0.981
LOG	8.1%	27.8%	0.980	0.722	0.832

Table 5: Detection accuracy rates on Dataset II.

We label 12 malicious apps, including adware, Trojan, and drive-by download apps. We successfully detect 12 newly installed malicious apps throughout the detection. A summary of the detected malicious apps is listed in Table 6. Our solution provides a high accuracy in detecting both existing malware families and new ones of all types. For the known malware variants, e.g. `wbfire.facts` and `com.crazyapps` families, our solution detects them as they send requests to fetch advertisements or conduct bot activities. During the study, our organization IDS (FireEye) only reported the malware activities from the `com.crazyapps.angry.birds` and classified it as `Trojan.Android.Plankton`. FireEye fails to detect other malware in our experiment.

ID	Package Name	Communication Server
Adware		
1	`com.allen.txtjjsz`	`gw.youmi.net`
2	`com.chenyx.tiltmazs`	`ade.wooboo.com.cn`
3	`com.zxcalendar.chapp`	`api.is.apmob.cn`
4	`wbfire.facts` family	`media.admob.com`
5	`com.ctinfotech.snake`	`media.admob.com`
Trojan/Backdoor/Bots/Spyware		
6	`com.crazyapps` family	`searchwebmobile.com`
7†	`com.gfgfgg.dsdf`	`send.cxpts.com`
8‡	`com.gucdxjdl.batterysaver`	`send.cxpts.com`
9	`com.GoldDream.TingTing06i`	`lebar.gicp.net`
10	`com.qnuou.game`	`lebar.gicp.net`
11	`com.wing.qingshongxry`	`static2.ad.anzhi.com`
Drive-by download/Update attack		
7†	`com.gfgfgg.dsdf`	`subscription.teebik.com`
8‡	`com.gucdxjdl.batterysaver`	`au.umeng.com`
12	`com.Punda.Free`	`ad.leadboltapps.net`

†, ‡: Both apps exhibit malicious behaviors in two categories.

Table 6: The package name and the communication server of malicious apps in our testing set of dataset II.

The malicious apps we detect cover a wide range of adware, Trojan, bots, spyware and drive-by download apps. For example, a game app (`com.Punda.Free`) generates a shortcut link icon to `ad.leadboltapps.net/show_app_icon` on the Android desktop once it is installed. It issues `GET` and `POST` requests to `ad.leadboltapps.net`. A bot app (`com.gfgfgg.dsdf`), pretending to offer optimization tools for Android, actively sends out `POST` requests to `send.cxpts.com` every 5 minutes in the background. We note that the Android bot is not included in the training set, while we can detect it in the testing. We identify the bot's malicious activities, as their requests are the lack of valid triggers. The detection does not rely on its signatures or C&C servers information.

We regard these malicious apps as *new*, as none of them is in the training set. Their traffic patterns may never be seen before. Results show that our solution successfully identifies all malware by flagging more than 99.1% their requests (out of 33,000+ HTTP requests). Our solution can detect them because the triggering relation tells the causality of requests and the root-triggers reveal the requests' legitimacy. Our method does not have any assumptions on the type of apps or what code obfuscation techniques used in malware.

Comparison with existing solutions. Existing solutions for detecting malicious URLs and domains use temporal, lexical and host-based features from the root-trigger requests [5, 22]. These features are effective in detecting malware that communicates to the hosts containing abnormal and meaningless URLs. However, their solutions do not consider human inputs and event triggers.

In our experiments, we confirm that the same classifiers, solely using lexical and host-based features, fail to identify 3.3%-4.9% malicious requests and cannot detect 4 malicious apps in Dataset II. By checking the network traffic of these malicious apps, we found that their URLs of communication hosts contain the common English vocabulary seen in benign websites.

FP and FN analysis. Based on the detection using C4.5 algorithm classifier, we find that the FPs are mostly due to

the redirected traffic to some well-known domain servers. E.g., when a malicious app fetches a list of apps in its WebView for luring users to download, some requests are sent to `lh3.ggpht.com` for retrieving images. These requests are FPs, because the domain is benign. Other FPs may be due to the uncommon domain names and unconventionally long request strings. FNs are mainly sent by spyware or Trojan.

6.8 Performance

We evaluate the performance in terms of *i)* the pairing efficiency for using our pairing algorithm and a baseline, and *ii)* the running time of testing on pairwise comparisons and detecting on root-triggers. We obtain the performance results by averaging the running time over 5 rounds.

Pairing efficiency. The baseline pairing operation was proposed in [44], which uses a fixed time threshold as pairing window and an efficient pairing algorithm to pre-screen the data before pairing. It takes 185 and 121 seconds to use the baseline algorithm and ours respectively, for pairing all requests in our dataset. Our algorithm improves at least 30% pairing performance.

TR analysis and detection. In Table 7, we report the runtime of each operation in our prototype. The performance of the classifiers is consistent for both datasets in each operation. The training time is a dominant factor of the total running time for all classifiers.

Operation	Dataset	Runtime: training + test (seconds)		
Pairing	I	0.064		
	II	0.087		
		C45	R-F	LOG
TR analysis	I	0.13+1.9e-3	0.16+2.3e-3	1.65+6.8e-3
	II	0.18+2.3e-3	0.19+3.1e-3	1.66+7.2e-3
Detection	I	0.05+8.2e-3	0.07+9.7e-3	0.21+1.2e-2
	II	0.03+1.1e-3	0.03+2.0e-3	0.17+4.8e-3

Table 7: The performance of each operation is shown. The results are calculated in seconds per 1000 records. The running time includes both training and testing in TR analysis and detection operations.

In both TR analysis and detection operation, C4.5 algorithm takes the least running time. The random forest classification is slightly slower than C4.5 algorithm, while logistic regression takes the longest time. Overall, the total runtime of the detection operation is significantly less than that of the TR analysis, as the size of $\mathbb{R}_{\mathbb{T}}$ is much less than that of $\mathbb{P}$, i.e., $|\mathbb{R}_{\mathbb{T}}| \ll |\mathbb{P}|$.

Summary. We highlight our findings below.

- The C4.5 algorithm and random forest classifiers give high precision and recall on finding the triggering relations and identifying the malicious requests. The false positive rate is 0.7% using the random forest classifier. Our approach to detecting malicious requests is scalable and does not require training on the entire data.
- Our evaluation involving 33,000+ Android app requests achieves a high detection accuracy rate. Results show that 99.1% of stealthy network activities with remote hosts can be identified. We confirm the detection capability of our approach by pinpointing the sparse anomalies out of voluminous traffic data.

7. RELATED WORK

Existing work on protecting Android system and detecting malicious apps is studied in the form of static analysis [21, 40], taint analysis [10, 38, 41], and privilege control [14, 32, 47]. Most static analysis solutions for detecting malicious apps leverage the features from API calls [1, 35, 39], system calls [20], function calls [15].

Dynamic analysis solutions (e.g., [6, 10, 41]) focus on analyzing the application behaviors at runtime. Crowdroid [6] classifies the benign and malicious apps based on the commonly seen system calls, which are obtained from real traces via crowdsourcing. TaintDroid [10] monitors the sensitive data flow on the system-wide, but it cannot provide the insights of how the data are triggered from the user's perspective. AppIntent [41] considers the user-intended data transmission on Android and builds the missing leak between the data leaking and user's interactions, while it needs tremendous efforts in static taint analysis as it treats the apps as whitebox. SmartDroid [48] proposes a hybrid analysis method to identify UI-based event trigger conditions using the sensitive APIs. Our solution systematically differs from the aforementioned dynamic analysis, as ours provides a new aspect by connecting the knowledge from user's interaction and network traffic for analysis the app's behavior, while treating the app as blackbox. App Guardian [46] proposes to temporally pause suspicious background process to prevent the potential data leaking. We do not solve the side channel attacks as App Guardian does, but our solution prevents the data exfiltration by identifying the suspicious outbound traffic. Recent work on policy analysis [31] uses semi-supervised learning to model and analyze the policy refinement process. In our work, we applied the learning-based approach on modeling and enforcing the dependency of network requests.

Dependence analysis plays an important role in detection Android malware. In our work, we utilize the dependence analysis on Android-generated network traffic to detect malicious requests and apps. Our approach extends the technologies on dependence analysis of network traffic [19, 42, 44, 45] and outperforms rule-based methods [8, 36]. Webprophet [19] and Rippler [42] present an approach to actively inject delays to infer the packet or service dependency. Their solutions are designed for the purposes of performance optimization and service dependency inference, as opposed to malware detection. WebWitness [27] is proposed to trace back the sequence of events (e.g., visited web pages) preceding malware downloads. It leverages automatically labeled malware download paths to better understand the attack trends, especially how users reach attack pages on the web. A most related work is introduced in [44]. Yet, our solution differs from theirs in three aspects.

1. Their work depends on a referrer-based heuristic, which limits its applications. In comparison, we use the delay injection approach for discovering triggering relations.
2. Their work is browser specific, as it requires a browser extension to log users' inputs. Our solution supports all types of apps on Android and is more general.
3. Their solution requires a straightforward manually generated whitelist to filter out non-user triggered benign requests. We designed a novel two-stage learning approach to automatically recognize benign requests.

Android network traffic classification has been studied in the literature [7,9,33,37]. Authors in [37] proposed to distinguish the mobile traffic by investigating the traffic identify from HTTP headers. NetworkProfiler introduces a technique to generate network profiles for identifying Android apps based on their HTTP traffic [9]. Recent studies [7, 23] show that an attacker can recognize user's actions by analyzing the network traffic, even it is encrypted. In comparison, our solution is focused on inferring the traffic dependence for identifying the malicious requests, which is beyond the traffic classification and profiling problems.

Also related are studies that proposed classification methods for identifying the suspicious/phishing URLs [5, 22, 34]. For example, Ma *et al.* proposed to examine the lexical features of the URLs and features of the domain information to identify malicious URLs [22]. Authors in [34] utilized page content-related features to detect phishing pages. Most of their features can be categorized into: *i)* time-based, *ii)* lexical and *iii)* host-based features. However, they do not include the dependency features as ours. Our study makes an important step towards addressing the significance of dependency knowledge in the detection of malicious requests on Android.

8. CONCLUSION AND FUTURE WORK

We described an Android malware detection technique that analyzes the dependency of mobile network traffic. Our analysis explores the request-level traffic dependence and reasons about the root-triggers for all HTTP requests sent from the device. We successfully demonstrated the use of triggering relation discovery for enhancing the sensemaking for security and identifying suspicious requests. Our evaluation confirms that our solution can identify 12 new malicious Android apps that are not previously seen in the training set. For the future work, we plan to deploy our tool to collect Android network traffic and detect more malicious apps via crowdsourcing. Also, we plan to extend our solution for real-time triggering relation inference and online detection.

9. REFERENCES

[1] Y. Aafer, W. Du, and H. Yin. DroidAPIMiner: Mining API-level features for robust malware detection in Android. In *SecureComm'13*. Springer.

[2] H. M. Almohri, D. Yao, and D. Kafura. Process authentication for high system assurance. *IEEE TDSC*, 11(2):168–180, 2014.

[3] Official Android store. https://play.google.com/store/apps.

[4] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck. DREBIN: effective and explainable detection of Android malware in your pocket. In *NDSS'14*, 2014.

[5] L. Bilge, E. Kirda, C. Kruegel, and M. Balduzzi. EXPOSURE: Finding malicious domains using passive DNS analysis. In *NDSS'11*.

[6] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. Crowdroid: behavior-based malware detection system for Android. In *SPSM'11*, pages 15–26.

[7] M. Conti, L. V. Mancini, R. Spolaor, and N. V. Verde. Can't you hear me knocking: Identification of user actions on Android apps via traffic analysis. In *CODASPY'15*, pages 297–304, 2015.

[8] W. Cui, R. H. Katz, and W.-t. Tan. BINDER: an extrusion-based break-in detector for personal computers. In *ACSAC'05*, pages 361–370.

[9] S. Dai, A. Tongaonkar, X. Wang, A. Nucci, and D. Song. NetworkProfiler: Towards automatic fingerprinting of Android apps. In *INFOCOM'13*, pages 809–817, 2013.

[10] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM TOCS*, 32(2):5, 2014.

[11] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of Android application security. In *USENIX'11*, 2011.

[12] A. Endert, P. Fiaux, and C. North. Semantic interaction for sensemaking: inferring analytical reasoning for model steering. *IEEE VCG*, 18(12):2879–2888, 2012.

[13] A. Endert, P. Fiaux, and C. North. Semantic interaction for visual text analytics. In *CHI'12*, pages 473–482, 2012.

[14] Y. Fratantonio, A. Bianchi, W. K. Robertson, M. Egele, C. Kruegel, E. Kirda, and G. Vigna. On the security and engineering implications of finer-grained access controls for android developers and users. In *DIMVA'15*, pages 282–303, 2015.

[15] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck. Structural detection of Android malware using embedded call graphs. In *AISec'13*, pages 45–54.

[16] L. Getoor and C. P. Diehl. Link mining: a survey. *SIGKDD Explor. Newsl.*, 7(2):3–12, December 2005.

[17] I. Kahanda and J. Neville. Using transactional information to predict link strength in online social networks. In *ICWSM'09*.

[18] Kidlogger. http://kidlogger.net/about.html.

[19] Z. Li, M. Zhang, Z. Zhu, Y. Chen, A. G. Greenberg, and Y.-M. Wang. WebProphet: Automating performance prediction for web services. In *NSDI'10*.

[20] Y.-D. Lin, Y.-C. Lai, C.-H. Chen, and H.-C. Tsai. Identifying Android malicious repackaged applications by thread-grained system call sequences. *computers & security*, 39:340–350, 2013.

[21] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. CHEX: statically vetting Android apps for component hijacking vulnerabilities. In *CCS'12*, pages 229–240, 2012.

[22] J. Ma, L. K. Saul, S. Savage, and G. M. Voelker. Beyond blacklists: learning to detect malicious web sites from suspicious URLs. In *KDD'09*, pages 1245–1254, 2009.

[23] B. Miller, L. Huang, A. D. Joseph, and J. D. Tygar. I know why you went to the clinic: risks and realization of https traffic analysis. In *PETS'14*, pages 143–163. Springer, 2014.

[24] D. Muthukumaran, A. Sawani, J. Schiffman, B. M. Jung, and T. Jaeger. Measuring integrity on mobile phone systems. In *SACMAT'08*, pages 155–164. ACM, 2008.

[25] C. Neasbitt, B. Li, R. Perdisci, L. Lu, K. Singh, and K. Li. WebCapsule: Towards a lightweight forensic engine for web browsers. In *CCS '15*, pages 133–145, 2015.

[26] C. Neasbitt, R. Perdisci, K. Li, and T. Nelms. ClickMiner: Towards forensic reconstruction of user-browser interactions from network traces. In *CCS'14*, pages 1244–1255.

[27] T. Nelms, R. Perdisci, M. Antonakakis, and M. Ahamad. WebWitness: investigating, categorizing, and mitigating malware download paths. In *USENIX Security'15*, pages 1025–1040, 2015.

[28] F. Roesner, T. Kohno, A. Moshchuk, B. Parno, H. J. Wang, and C. Cowan. User-driven access control: Rethinking permission granting in modern operating systems. In *SP'12*, pages 224–238. IEEE, 2012.

[29] D. Shackleford. Blind as a bat? supporting packet decryption for security scanning (white paper). November 2012.

[30] G. Snedecor. *Statistical methods*. Iowa State University Press, 1989.

[31] R. Wang, W. Enck, D. Reeves, X. Zhang, P. Ning, D. Xu, W. Zhou, and A. M. Azab. EASEAndroid: Automatic policy analysis and refinement for security enhanced Android via large-scale semi-supervised learning. In *USENIX Security'15*, pages 351–366, 2015.

[32] Y. Wang, S. Hariharan, C. Zhao, J. Liu, and W. Du. Compac: Enforce component-level access control in Android. In *CODASPY'14*, pages 25–36.

[33] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos. ProfileDroid: multi-layer profiling of Android applications. In *Mobicom'12*, pages 137–148, 2012.

[34] C. Whittaker, B. Ryner, and M. Nazif. Large-scale automatic classification of phishing pages. In *NDSS'10*.

[35] D.-J. Wu, C.-H. Mao, T.-E. Wei, H.-M. Lee, and K.-P. Wu. DroidMat: Android malware detection through manifest and API calls tracing. In *Asia JCIS'12*, pages 62–69.

[36] G. Xie, M. Iliofotou, T. Karagiannis, M. Faloutsos, and Y. Jin. ReSurf: Reconstructing web-surfing activity from network traffic. In *IFIP Networking Conference, 2013*, pages 1–9.

[37] Q. Xu, Y. Liao, S. Miskovic, M. Baldi, Z. M. Mao, A. Nucci, and T. Andrews. Automatic generation of mobile app signatures from traffic observations. In *INFOCOM'15*.

[38] L. Yan and H. Yin. DroidScope: Seamlessly reconstructing the OS and dalvik semantic views for dynamic Android malware analysis. In *USENIX'12*, pages 569–584, 2012.

[39] C. Yang, Z. Xu, G. Gu, V. Yegneswaran, and P. Porras. DroidMiner: Automated mining and characterization of fine-grained malicious behaviors in Android applications. In *ESORICS'14*, pages 163–182. Springer, 2014.

[40] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck. AppContext: Differentiating malicious and benign mobile app behaviors using context. In *ICSE'15*, 2015.

[41] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang. AppIntent: Analyzing sensitive data transmission in Android for privacy leakage detection. In *CCS'13*, pages 1043–1054.

[42] A. Zand, G. Vigna, R. Kemmerer, and C. Kruegel. Rippler: Delay injection for service dependency detection. In *INFOCOM'14*.

[43] H. Zhang, M. Sun, D. Yao, and C. North. Visualizing traffic causality for analyzing network anomalies. In *IWSPA'15*, pages 37–42, 2015.

[44] H. Zhang, D. Yao, and N. Ramakrishnan. Detection of stealthy malware activities with traffic causality and scalable triggering relation discovery. In *ASIACCS'14*, pages 39–50.

[45] H. Zhang, D. Yao, N. Ramakrishnan, and Z. Zhang. Causality reasoning about network events for detecting stealthy malware activities. *Computers & Security*, 58:180–198, 2016.

[46] N. Zhang, K. Yuan, M. Naveed, X. Zhou, and X. Wang. Leave me alone: App-level protection against runtime information gathering on Android. In *SP'15*, pages 915–930, 2015.

[47] X. Zhang, A. Ahlawat, and W. Du. AFrame: Isolating advertisements from mobile applications in Android. In *ACSAC'13*, pages 9–18.

[48] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou. SmartDroid: an automatic system for revealing UI-based trigger conditions in Android applications. In *SPSM'12*, pages 93–104. ACM, 2012.

[49] Y. Zhou and X. Jiang. Dissecting Android malware: Characterization and evolution. In *SP'12*, pages 95–109. IEEE, 2012.