2021 IEEE 7th International Conference on Big Data Intelligence and Computing (DataCom) | 978-1-6654-2071-6/21/$31.00 ©2021 IEEE | DOI: 10.1109/DataCom53700.2021.00014

Mitigating App Collusion using Machine Learning

Xuefei Duan, Hua Lu
GuangDong Communications & Networks Institute,
Huangpu District, Guangzhou City,
Guangdong Province, China
duanxuefei@gdcni.cn, luhua@gdcni.cn

Jinliang Yuan, Qiyang Zhang, Dongqi Cai
Beijing University of Posts and Telecommunications
Haidian District, Beijing City, China
yuanjinliang@bupt.edu.cn
qyzhang@bupt.edu.cn, cdq@bupt.edu.cn

***Abstract*—It is well-known that many apps can drain smartphones' compute resources at background and thus affect user experience. Recently, researchers are studying hidden compositional launch, a malicious behavior that apps wake up other apps in background without users' awareness. Prior study has shown that covert, cooperative behaviors in background app launch are surprisingly pervasive especially on third-party appstores, and such collusion has serious impact on performance, efficiency, and security. In this work, we develop a runtime scheduler namely AppConan that can learn the user interaction behaviors over apps, monitor the hidden compositional launch online, and automatically suppresses those colluded apps affecting user-interaction experiences.**

I. Introduction

Modern mobile applications (a.k.a., apps) are indispensable in our daily life. It is well-known that the apps often run background activities periodically to poll sensor data, maintain cloud activity, or update their local state [1]. While such background activities could be essential to the "always-on" interaction experience desired by users, they often have incurred substantial overhead which may be undesirable. For example, a recent study [2] examined 800 apps running over 1,520 devices, and the results surprisingly indicated that background activities can account for more than 50% of the total energy for 22.7% of the apps, with an average of 27.1% across all 800 apps.

Essentially, the intention of background execution in Android or iOS is to enable an app – directly launched by the user – to remain alive and get executed from time to time. However, the mechanism also allows one app to *programmatically* launch other apps in the background, even though the users never interact with or intend to use such apps. More seriously, the users may be never aware of the launched apps.

Following prior work [3], we use the term, *app collusion*, to denote such **user-unaware cross-app compositional launch**. Compared to the background activities that are triggered by explicit user input or configuration, hidden compositional launch is covert and often more sophisticated. As a result, the incurred system overhead and security risk are not only substantial but also often hidden. What is even worse, such overhead and risk can grow as one user installs more apps on her mobile device.

The seriousness and overhead of hidden compositional launch have been quantified in prior work [3]. A controlled experiment with 40 popular apps indicates that for every single foreground app, the background collusion can lead to additional 202 MB memory use, extra 8.9% CPU usage, and 16 additional high-risk permissions on average. With hidden compositional launch present, the user perceives 2× latency in launching foreground apps and the off-screen battery life is reduced by 57%. A field study over the two-month app-usage logs collected from 78 volunteer students also evidences the overhead caused by app collusion, which can reach 9.2% CPU and 288 MB memory daily on average.

To mitigate such overhead, we design the AppConan tool along with a machine learning algorithm that can help eliminate the app-collusion-introduced hidden cost by deriving the user interaction behaviors on their mobile devices. Our algorithm can accurately predict those apps that an user is likely to interact with in a specific time window, and thus can decide whether the apps colluding with current active apps should be launched or not by estimating the potential system-wide overhead. We evaluate the approach over the same collected user behaviors, and the results show that AppConan can save about 8.2% CPU usage, 201 MB memory usage, and 8.4 MB daily network usage per day on average.

II. Problem Statement

Intuitively, hidden compositional launch is a directed relationship among two apps indicating that one can launch the other without users' notice. We formally define the hidden compositional launch as follows.

An app $app = \{com_i\}$ can be abstracted as a set of components com_i. Components can communicate with each other to realize the functionality of an app, denoted as *Inter-Component Communication* (ICC). Generally, ICC can occur between components either within the same app or from different apps. To study the app-collusion behavior, we define a special case for ICC named Inter-App Collusion (*IAC*) as $\langle com_i, com_j \rangle$, where three conditions are satisfied: 1) com_i can launch com_j, 2) com_i and com_j come from different apps, and 3) com_j is not an activity. It should be particularly noticed that the third condition constrains no existence of UI switch, so that IAC is invisible and un-perceived to users. At app level, hidden compositional launch is a binary relationship among apps, hidden compositional launch $= \langle app_i, app_j \rangle \subset APP \times APP$, where $\exists com_i \in app_i, com_j \in app_j, \langle com_i, com_j \rangle \in$ *IAC*.

The following code shows a simple implementation of hidden compositional launch. When `methodA` is invoked, a new `Intent` is initialized to target at a service from another

Authorized licensed use limited to: BEIJING UNIVERSITY OF POST AND TELECOM. Downloaded on July 28,2022 at 12:32:41 UTC from IEEE Xplore. Restrictions apply.

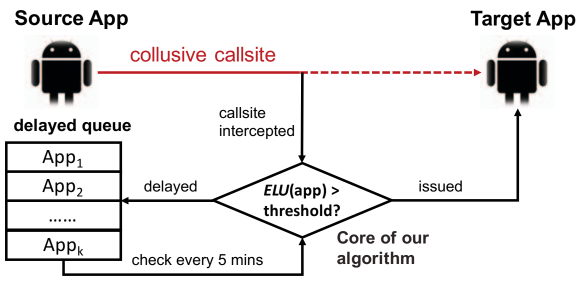

Fig. 1: Workflow of AppConan.

app (line 4∼5), and sent to the system to wake up that service (line 7). Since a service has no UI elements, users have no idea of its launch.

Listing 1: A code sample of app collusion

```
public class classA{
   public void methodA(Context context) {
      // com.example.target is not the
          current package;
      Intent intent = new Intent();
      intent.setClassName("com.example.target",
          "com.example.target.otherService");
      intent.setAction("some_action");
      context.startService(intent);
   }
}
```

III. AppConan: Eliminating Collusion-Related Hidden Cost

Motivated by the high prevalence and non-trivial hidden cost of hidden compositional launch, we aim to further propose an online optimization approach, namely AppConan, to eliminating the undesired overhead caused by those hidden compositional launch.

The major challenge of our goal is that we don't want to interfere too much with the original apps' normal behavior when suppressing the collusion behavior. Indeed, app collusion can have positive aspects sometimes, among which the probably to-be-used apps can be pre-launched by the currently used app in the background, so that users can quickly open and interact with this app later (we call it "warm launch" in this article). In other words, simply cutting-off all hidden compositional launch cannot be directly adopted since it can compromise the user experience. For instance disabling all hidden compositional launch can make apps fail to receive notifications the cloud. As a result, users could miss some important messages which they want to subscribe from some apps. Therefore, we need a more intelligent and moderate mechanism to identify those "useless" collusive apps that do not positively contribute to user experience, so that we can disable or delay their launch.

A. AppConan Design

Based on the preceding analysis, the design principles of the online optimization approach proposed in this article are as follows.

- **Time-Aware**. If a collusive app is unlikely to be interacted by user within a given time window (e.g., 5 minutes), it should not be launched in background via hidden compositional launch. On the contrary, if the collusive app is likely to be used by the user, we view such a collusion as "useful" and allow its running in background. This principle is based on the intuitive observation: after being launched in the background, an app can prepare its data in advance and response to user action more quickly and smoothly.
- **Resource-Consumption-Wise**. The more resources an app consumes in background, the more probably it shall be disabled or delayed from being launched by hidden compositional launch. Indeed, this principle requires the trade-off against the preceding principle, i.e., keeping more apps alive and saving system resources.
- **Personalized**. The runtime optimization should be highly personalized for different users and configurable for different hardware device specifications. Such a flexibility is necessary as users have different usage patterns of apps and devices have different amount of available system resources.

To address the preceding three principles, we design AppConan as shown in Figure 1. The core component of AppConanis a run-time decision maker, which determines whether an app collusion should be delayed or issued when it happens. The decision is made by calculating the *ELU* value of the app to be launched, and compares it to a configurable threshold $\mathcal{T}$ that can be defined by either developers or users. The *ELU* value indicates how much the app x is *eager to be launched by the user*, and is the key indicator of AppConan. It is directly derived from a probability model $\mathcal{P}$ and a resource weighting model $\mathcal{R}$ in the following formula. We will explain the definition and the design details of $\mathcal{P}$ and $\mathcal{R}$ later in this section.

$$ELU(x, context) = \frac{\mathcal{P}(x, context)}{\mathcal{R}(x)}$$

If the *ELU* of an app is higher, we enable the collusion to launch the app. Otherwise, we delay this app in a queue that maintains a list of apps to launch. In this article, we update the *ELU* of all apps in the queue for every 5 minutes, and enable the collusion if the updated *ELU* is higher compared to the threshold. We then explain how the two models are defined.

1) Probability Model: $\mathcal{P}(x, context)$ means the probability of app x being used by users in the foreground within a given time window τ from now on given the current *context*. A lower $\mathcal{P}(x)$ value indicates less necessity of x being launched in the background since it is unlikely to be used by users. Therefore, we can delay this app to save more resources.

Intuitively, such a prediction problem can be treated as a binary-classification problem, i.e., given the current context information and a particular app, whether this app will be used or not within a given time window τ. A straightforward approach is recording the actual user interaction after

Authorized licensed use limited to: BEIJING UNIVERSITY OF POST AND TELECOM. Downloaded on July 28,2022 at 12:32:41 UTC from IEEE Xplore. Restrictions apply.

hidden compositional launch as positive instances and the non-happened interaction as negative instances to train a classifier. However, due to the sparsity of the positive instances in the whole data span (any context at an arbitrary time point combined with any possible app that can be an observed instance), it is hardly known which app will never be used given a particular context.

Theoretically, such a problem is known as a typical One-Class Classification problem [4]. Here we adopt a state-of-the-art algorithm *PU* proposed by Liu et al. [5] [6] to solve the problem. The basic idea of the algorithm is to use the positive data to identify a set of informative negative samples from the whole data span, in order to keep the positive and negative samples balanced.

The overall classification process is described in Algorithm III.1. At the beginning, the algorithm generates the whole data span U by enumerating every combination of target apps a and context. Then the algorithm identifies the informative negative samples according to the positive training data. For prediction, given the current context c, we can calculate the probability of any candidate target app ta that will be used by users within a given time window τ.

```
Input: PU Method $\mathcal{PU}$ , Positive dataset $P$, Candidate App
       Set $S$=($a_1, a_2, ..., a_N$)
Output: Recommendation List $L$
1 $U \leftarrow geneate\_whole\_data\_span()$
2 $N \leftarrow extract\_negative(U, P)$ according to [5]
3 $\mathcal{PU}.classifier.fit(P \cup N)$
4 foreach $ta \in S$ do
5   | $prob[tp] = \mathcal{PU}.classifier.predict\_probability((c, tp))$
6 end
7 $L \leftarrow sort\_by\_probability(P, prob)$
8 return $L$
```

Algorithm III.1: One-class classification algorithm for prediction.

To achieve effective prediction results, we carefully choose some meaningful context features, including information of the previously launched app (id, elapsed time, and category), network condition, battery level, and local system time. We adopt a variant of the PU algorithm, namely SPY+SVM, to accomplish the classification task.

2) Resources Weighing Model: $\mathcal{R}(x)$ measures how many system resources app x can consume in background. A high value of $\mathcal{R}(x)$ indicates that we should more aggressively disable or delay x from being launched since it consumes more resources than other apps. In detail, it is calculated as follows,

$$\mathcal{R}(x) = \sum_{i=0}^{n} \mathcal{W}_i \cdot \frac{r_i(x)}{R_i}$$

where $r_i(x)$ is the consumption of the i-th resource type for app x, R_i is the average consumption of the i-th resources type for all installed apps on the device, and $\mathcal{W}_i$ is the importance weight of the i-th resources type. In this article, we consider three resource types: CPU, memory, and network usage.

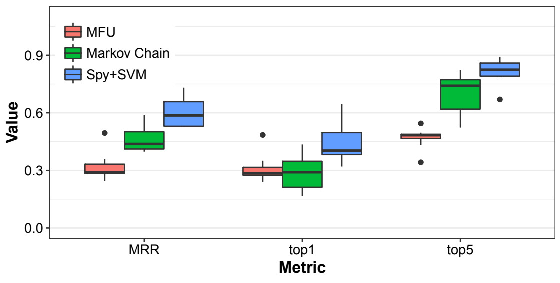

Fig. 2: Prediction results with different algorithms.

B. Choosing Parameters

The online optimization approach has three sets of configurable parameters: resources importance weight $\mathcal{W}_i$, threshold $\mathcal{T}$, and time window τ.

Importance Weight. $\mathcal{W}_i$ indicates how significant the i-th resources is under the current conditions of the user's device. For instance, if the foreground app is very CPU-intensive, the weight of CPU resources should be assigned with a higher value. If the app is running out of network data, the weight of network needs to be higher accordingly.

Threshold and Time Window. The parameters $\mathcal{T}$ and τ are designed for the flexible trade-off between keeping more apps alive and delaying more apps to be launched to save system resource. A higher threshold $\mathcal{T}$ and a smaller time window τ indicates more likelihood to disable or delay the hidden compositional launch.

C. Evaluation

To evaluate how well AppConan performs, we collected the device usage data from 10 volunteers for 20 days. We use the data train our probability model $\mathcal{P}$ and evaluate its accuracy. Then, we demonstrate how much system resources AppConan can save along with very small impacts on the user experience.

1) Accuracy of Probability Model: We first evaluate the performance of our proposed prediction model, i.e., whether it can precisely predict whether an app will be used. We use two metrics to measure the model's performance, i.e., the accuracy and the Mean Reciprocal Rank (MRR) score. The MRR of a recommendation list is the multiplicative inverse of the rank for the correct answer, which is a promising indicator in the ranking problem.

We compare the proposed model with two straightforward algorithms including the Most-Frequently-Used (*MFU*) model that ranks the apps according to their usage frequency, and the *Markov Chain* based model that ranks the apps by the static transition probability for a target app to be navigated from the current app.

We use 30% of the collected data as the test set, and the remaining 70% data as the training set. We train the model for every single user and evaluate the mean accuracy. Fig 2 illustrates that our proposed model outperforms the naive

Authorized licensed use limited to: BEIJING UNIVERSITY OF POST AND TELECOM. Downloaded on July 28,2022 at 12:32:41 UTC from IEEE Xplore. Restrictions apply.

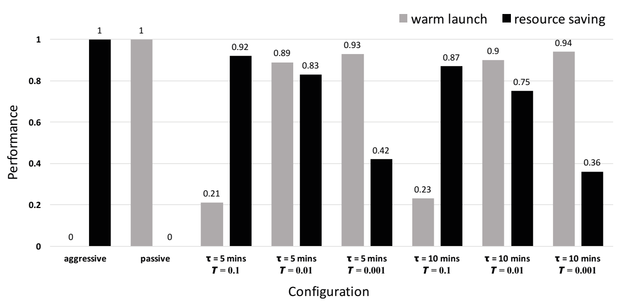

Fig. 3: Evaluation of our proposed algorithm in consideration of how it can keep apps launched by users quickly (warm launch) and can save the system resource. Both metrics are normalized. We have two ground truth: ***aggressive*** means disabling all hidden compositional launch, while ***passive*** means keeping the default strategy as Android and killing no hidden compositional launch.

MFU and the *Markov Chain* algorithms, with its average top-1 accuracy of 0.43, top-3 accuracy of 0.74, and the average MRR score of 0.60. Such a result indicates that the actual most used app can be accurately predicted by our model.

Our proposed method is targeted for predicting whether an app will be used within a given time window. Indeed, there have been some efforts on app-usage prediction [7] [8] [9]. Our prediction result is comparable to these state-of-the-art methods. More specifically, *APPM* [7] achieves around 80.9% accuracy for top-5 prediction, while our model achieves around 81.7%. In the future, we plan to incorporate these existing prediction strategies into our model to further improve the accuracy.

2) Performance of AppConan: We then evaluate the overall performance of AppConan from two aspects: 1) How it affects the warm launch? Inaccurately delaying apps from being launched can lead to slow cold launch, which should be avoid. 2) How much resources AppConan can save? Here we simply present the saved resources for CPU, memory, and network data on average.

We emulate the app behavior by performing AppConan on the collected data. The result of warm launch can be simply obtained via looking up the background app list every time when user launches an app. To figure out how many resources are saved, we check the amount of resources consumed by the apps that are delayed from being launched, at every 5-minutes interval. For the configurable parameters, we simply set $\mathcal{W}_i$ (i=1, 2, 3) for all three types of resources equally as 0.33, and iterate the different values of time window τ and the threshold $\mathcal{T}$.

As observed from Figure 3, when setting the parameters properly (e.g., $\mathcal{T}$=0.01, τ=5 mins), AppConan can significantly help save system resources (83%) consumed by the background apps that are launched via hidden compositional launch, while imposing quite little on the user experience (keeping 89% warm launch). In detail, this configuration can save about 8.2% CPU usage, 201 MB memory usage, and 8.4 MB cellular usage per day on average. Essentially, the key reason of such an improvement with little side-effect is due to our probability model $\mathcal{P}$, which accurately helps identify "useless" hidden compositional launch at runtime.

Figure 3 also indicates that different configurations can lead to high divergence among warm launch and resource saving. Instead of fixing $\mathcal{T}$ and τ as constants, a more intelligent way is making the two parameters adaptive to user preferences and device status with better trade-offs. For example, for users who care more about system resource, or for devices that are running out of battery, we can dynamically assign higher $\mathcal{T}$ and lower τ.

In summary, we have demonstrated current AppConan is effective to optimize the undesirable overhead caused by collusion.

IV. Conclusions

In this work, we develop a runtime algorithm namely AppConan to mitigate the negative impacts from hidden compositional launch. AppConancan learn the user interaction behaviors over apps, monitor the hidden compositional launch online, and automatically suppresses those colluded apps affecting user-interaction experiences.

V. Acknowledgement

We thank all the researchers providing support for the paper. This work was partially supported by the Key-Area Research and Development Program of Guangdong Province (No.2018B010113001)

References

[1] M. Martins, J. Cappos, and R. Fonseca, "Selectively taming background android apps to improve battery lifetime," in *Proc. of ATC'15*, 2015, pp. 563–575.

[2] X. Chen, N. Ding, A. Jindal, Y. C. Hu, M. Gupta, and R. Vannithamby, "Smartphone energy drain in the wild: Analysis and implications," in *Proc. of SigMetrics'15*, 2015, pp. 151–164.

[3] M. Xu, Y. Ma, X. Liu, F. X. Lin, and Y. Liu, "Appholmes: Detecting and characterizing app collusion among third-party android markets," in *Proceedings of the 26th international conference on World Wide Web*, 2017, pp. 143–152.

[4] D. M. J. Tax, "One-class classification," *Applied Sciences*, 2001.

[5] B. Liu, Y. Dai, X. Li, W. S. Lee, and P. S. Yu, "Building text classifiers using positive and unlabeled examples," in *Proc. of ICDM 2003*, 2003, pp. 179–188.

[6] X. Li and B. Liu, "Learning to classify texts using positive and unlabeled data," in *Proc. of IJCAI 2003*, 2003, pp. 587–594.

[7] A. Parate, M. Hmer, D. Chu, D. Ganesan, and B. M. Marlin, "Practical prediction and prefetch for faster access to applications on mobile phones," in *Proc. of UbiComp 2013*, 2013, pp. 275–284.

[8] T. Yan, D. Chu, D. Ganesan, A. Kansal, and J. Liu, "Fast app launching for mobile devices using predictive user context," in *Proc. of MobiSys 2012*, 2012, pp. 113–126.

[9] R. Baeza-Yates, D. Jiang, F. Silvestri, and B. Harrison, "Predicting the next app that you are going to use," in *Proc. of WSDM 2015*, 2015, pp. 285–294.

Authorized licensed use limited to: BEIJING UNIVERSITY OF POST AND TELECOM. Downloaded on July 28,2022 at 12:32:41 UTC from IEEE Xplore. Restrictions apply.