RIP StrandHogg: A Practical Detection Method on Android

Jasper Stang
Julius-Maximilians-Universität Würzburg
Würzburg, Germany
jasper.stang@stud-mail.uni-wuerzburg.de

Alexandra Dmitrienko
Julius-Maximilians-Universität Würzburg
Würzburg, Germany
alexandra.dmitrienko@uni-wuerzburg.de

Sascha Roth
KOBIL Systems GmbH
Worms, Germany
sascha.roth@kobil.com

ABSTRACT

StrandHogg vulnerabilities affect Android's multitasking system and threaten up to 90% of Android platforms, which translates to millions of affected users. Existing countermeasures require modification of the OS, have usability drawbacks, or are limited to the detection of certain attack versions. In this work, we aim to develop a generic, efficient, and usability-friendly attack detection method, which does not require OS modifications and can be employed by apps installed on any vulnerable Android platform. To achieve our goal, we analyze StrandHogg attack techniques and develop two countermeasures, one using Machine Learning and the other one using ActivityCounter – a reliable attack indicator, which we could synthetically engineer. Our first approach achieves an average F1 score of 92% across all attack variations, while ActivityCounter shows superior performance and efficiently detects all attack versions without false positives. ActivityCounter is the first solution without practical limitations, which can be easily deployed in practice and protect millions of affected users.

CCS CONCEPTS

• **Security and privacy → Mobile and wireless security**; *Malware and its mitigation.*

KEYWORDS

Android, StrandHogg, StrandHogg Detection

ACM Reference Format:
Jasper Stang, Alexandra Dmitrienko, and Sascha Roth. 2021. RIP StrandHogg: A Practical Detection Method on Android. In *Conference on Security and Privacy in Wireless and Mobile Networks (WiSec '21), June 28–July 2, 2021, Abu Dhabi, United Arab Emirates.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3448300.3468288

1 INTRODUCTION

StrandHogg [28] is a task hijacking attack on Android, which abuses Android's multitasking system. It belongs to the class of User Interface (UI) deception attacks [5], where an attacker forces users to confuse malicious UI elements with the ones originating from a legitimate app. As a result of task hijacking, an attacker can, e.g., steal security-sensitive user input [2, 4, 21, 29], or achieve privilege escalation by deceiving the user into granting permissions.

UI deception attacks are difficult to address in a generic sense. While a few attempts exist [1, 5], they require significant modifications of the OS to such an extent that the resulting system might not anymore be considered as Android. Hence, defenses often target specific types of UI attacks (e.g., [22, 23]).

StrandHogg stands out from the line of other task hijacking attacks as, while being discovered in 2015, it was neither fixed in any Android version, nor other practical defense methods were developed. As of now, it is still feasible to launch this attack on Android versions ranging from 4.4 to Android 10, which corresponds to 90% of all Android platforms [24]. Furthermore, solutions proposed in the related work have various drawbacks, e.g., require OS-level modifications [27, 32], impose false negatives [32] or limit functionality [33], thus affecting usability.

Moreover, recently a new variation of the StrandHogg attack was discovered – it was described in the blog [26] and named as StrandHogg 2.0 - the 'evil twin' attack. This second version was recorded in a Common Vulnerabilities and Exposures entry (CVE-2020-0096) and fixed in the patch level 2020-05-05 [17]. This leaves devices with Android versions from 4.4 to 9.0, or 55% of all Android platforms [24] vulnerable to both attack versions. Even more worrying, many countermeasures [1, 20, 31] developed against the original StrandHogg attack appeared toothless against the second version.

In this paper, we aim to tackle the StrandHogg vulnerability of the Android OS and look for a countermeasure, which is able to deal with both StrandHogg attack versions at the same time. Furthermore, we aim at a solution that would be capable of protecting apps on all vulnerable Android versions, i.e., with versions ranging from 4.4 to 10. To ensure the practicality of our defense method, we avoid techniques that require modifications to the operating system. Therefore, the defense method must be integrated into each app that desires to be protected. A majority of vendors only support their devices for a limited period (typically three years), and, oftentimes, the devices no longer receive security fixes after this period[1].

Contributions: To achieve our goal, we develop two attack detection methods, both being capable of providing effective protection for apps executing on any Android platform against both StrandHogg versions. In particular, we make the following contributions:

WiSec '21, June 28–July 2, 2021, Abu Dhabi, United Arab Emirates

ACM ISBN 978-1-4503-8349-3/21/06…$15.00
https://doi.org/10.1145/3448300.3468288

[1] According to statistics from Malwarelytics [3], 20.5 percent of devices will never receive a patch for StrandHogg v2 (CVE-2020-0096).

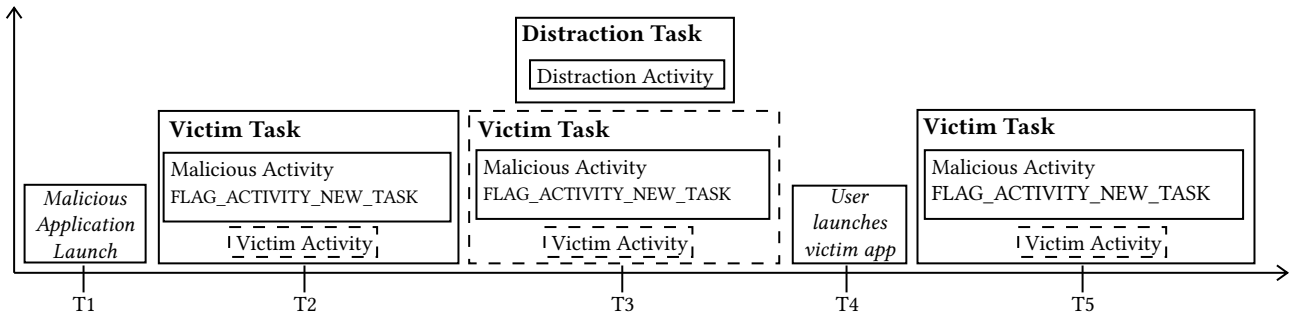

Figure 1: StrandHogg v1 Attack visualization. Dashed borders indicate a background task/activity. The topmost task/activity is displayed in the foreground.

- We analyzed both StrandHogg attack methods in order to gain a deep understanding of attack mechanics (cf. Section 2). Using the gained knowledge, we implement both attack versions and utilize them in the evaluation of defense methods. To the best of our knowledge no fully-working StrandHogg v2 attack was already available online. We will, therefore, make these implementations available for researchers[2].
- We developed a first attack detection approach, which enables a service provider to protect their mobile app from StrandHogg attacks through the application of Machine Learning (ML) (cf. Section 4). This method involves app monitoring, collection of execution traces and relies on the server's help for attack inference. It achieves an average F1 score of 92% on traces collected on Android platforms with versions from 4.4 to 10.
- Developing the ML-based method gave us knowledge gain necessary for developing a second, more effective detection method, which we named ActivityCounter (cf. Section 5). It relies on a single `numActivties` attribute for attack detection and does not require any server support. To realize ActivityCounter, we had to solve technical challenges, one to turn the `numActivties` attribute into a reliable attack indicator and the other one to eliminate any changes to the Android OS. ActivityCounter can detect both attack versions on all affected Android platforms, has no false positives and no false negatives, and is privacy-friendly, since it performs on-device attack detection. Furthermore, it is a vendor-independent solution as it does not rely on any vendor-specific components.

To summarize, both developed methods outperform existing work since they can detect both attack versions, do not require any changes to the OS, have little or no impact on usability, and do not limit functionality. ActivityCouter is superior to all existing defense methods including the ML-based approach, since it is more effective, privacy-, and user-friendly. It can be deployed on any vulnerable Android version and, hence, can vendor-independently protect millions of affected users.

[2]Download link: https://github.com/ActivityCounter/StrandHoggAttacks

2 BACKGROUND

In this section, we provide the necessary background information on Android and describe StrandHogg attacks and known mitigation techniques.

2.1 Android

Understanding of StrandHogg attack internals requires knowledge about several elements of the Android operating system: Activities, Tasks, and Task Affinities.

Activities Activity is the main User Interface component in Android. By means of displaying activities on the screen, Android apps interact with the user. An app can display multiple activities for different user interaction purposes such as searching in contacts or writing a message. All activities must be defined in the app's manifest – the file that describes essential information about the app like the package name, components or permissions [9].

Tasks To manage all opened activities of an application, the Android OS keeps them in a task. Each task, therefore, consists of a collection of activities. Usually, the user only interacts with one activity at a time – the foreground activity of the task. The task that contains the foreground activity, which the user is currently interacting with, is called the foreground task. All other tasks are therefore in a paused state, which makes them background tasks. The activities in a task are kept in a stack ordered by the time the activities were visited. That enables the user to switch back to previous activities by pressing, for instance, the back button on the device [6].

Task Affinity Each activity can specify to which task it prefers to belong. This can be done by setting the `android:taskAffinity` attribute, in the manifest file, to the package name of the preferred task. By default, each activity inherits the affinity of its own application. That means an activity has to explicitly specify if it wants to belong to a *foreign* task [7].

2.2 StrandHogg Attacks

Android's `android:taskAffinity` attribute enables a highly dangerous attack, where an adversary is able to hijack another app's foreground activity and display a malicious one instead. From 2015 to this day, there are 2 variations of the StrandHogg attack known, which we refer to as StrandHogg v1 and StrandHogg v2 attacks.

StrandHogg v1 Attack. The original attack was first theoretically described by Ren et al. in 2015 in the paper "Towards Discovering

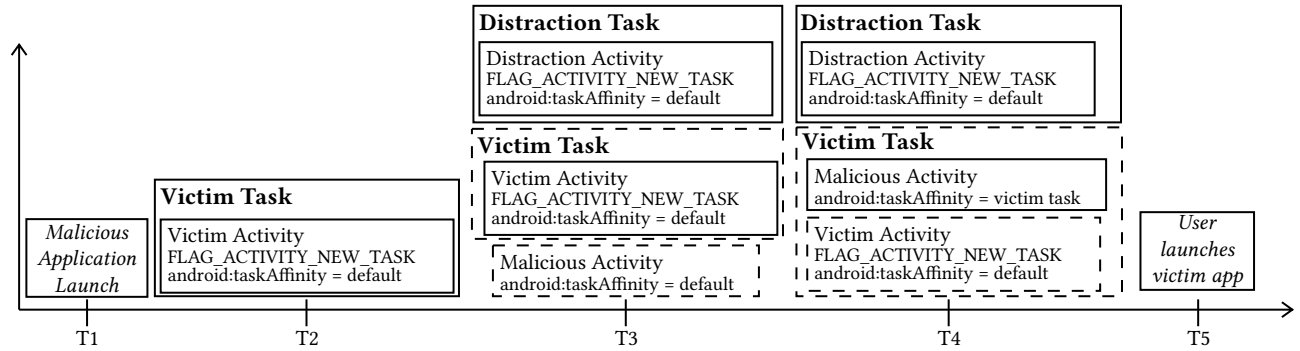

Figure 2: StrandHogg attack v2 visualization. Dashed borders indicate a background task/activity. The topmost task/activity is displayed in the foreground.

and Understanding Task Hijacking in Android" [28]. Høegh-Omdal et al. [25] presented a practical implementation approach for the attack and discovered that the vulnerability was actively exploited as early as in 2017. According to the authors, the attack is feasible on Android devices with versions from 4.4 up to 10 [25]. The attack does not work on the latest Android version 11.

A visualization of the attack is shown in Figure 1. Each step (**T1 - T5**) of the attack is marked on the x-axis. The attack involves a malicious and a victim app, each having their own activities. There are three activities: malicious activity, victim activity, and distraction activity. The malicious and distraction activity are parts of the malicious application, while the victim activity is the main activity of the victim app. We will call the task in which the victim activity resides as a victim task. Furthermore, the distraction activity is used during an attack as a distraction to the user, and it belongs to its own distraction task. The malicious activity is the core of the attack and will be displayed instead of the victim activity, hence, with this activity, an adversary can, for instance, steal passwords and other sensitive input.

As the first step in the attack, a user launches a malicious application on the device – this is shown as step **T1** in the Figure 1. This app will then launch the malicious activity with the flag `FLAG_ACTIVITY_NEW_TASK`. The flag will either create a new victim task, in case none is currently running, or otherwise the victim task will be brought to the front [13]. The malicious activity additionally declares that it wants to belong to the victim task, by setting the `android:taskAffinity` attribute (**T2**). To prevent that the malicious activity is instantly displayed, the distraction activity, belonging to the adversarial app, is launched right afterward. This results in the victim task becoming a background task, as the distraction activity is displayed in the foreground (**T3**). The foreground task only contains the distraction activity. The malicious activity will reside in the victim's task, which is in the background.

The user can now interact freely with the distraction activity such as minimizing or closing it. The malicious activity will stay in the victim task, despite the distraction activity's closing. The victim task stays in the background until the user resumes to it – it can be resumed by either clicking the victim's app icon or the activity in the list of recent activities (**T4**). In case the victim task is resumed by the user, the malicious activity will be displayed instead of the victim's activity (**T5**).

StrandHogg v2 The second version of the StrandHogg attack was discovered by Høegh-Omdal [26] and was named as an *evil twin* attack. This version does not rely on the `android:taskAffinity` attribute and, hence, can evade many detection methods that were developed for the first version. According to Promon [26], the StrandHogg v2 attack affects devices with versions ranging from 4.4 to 9.0. The attack does not work on devices running Android 10 or 11. The attack was recorded in a Common Vulnerabilities and Exposures entry (CVE-2020-0096) and fixed in the patch level 2020-05-05 [17].

A visualization of the attack can be seen in Figure 2. Each step (**T1 - T5**) of the attack is marked on the x-axis. Again, we introduce the three activities: malicious activity, distraction activity, and victim activity.

The attack exploits a confusion of the task affinity. In the first step, the malicious application is launched on the device. In the second step, the victim activity is launched (cf. Fig. 2 – **T2**). Right after the victim activity was created, it is instantly minimized by launching an array of activities (**T3**) consisting of the malicious activity which has no flags set, the victim activity with the flag `FLAG_ACTIVITY_NEW_TASK` and a seemingly benign distraction activity, again with the flag `FLAG_ACTIVITY_NEW_TASK`. By launching the victim activity for a second time, the order of foreground activities will be shifted. The absence of flags from the malicious activity misleads the Android OS to set the task affinity to the victim app (**T4**). The user can again interact freely with the distraction activity. The attack is identical behavior-wise to the first StrandHogg attack – as soon as the victim app's icon is clicked, the malicious activity will be launched instead of the victim activity (**T5**).

The attack works without setting the `android:taskAffinity` attribute in the app's manifest. This enables the attack to evade detection methods that rely on an analysis of the app's manifest. Furthermore, it is possible to execute the attack entirely in memory, as no prior preparations, such as declaring attributes in the manifest, are required. The Fix for the attack forces the malicious activity to declare the flag `FLAG_ACTIVITY_NEW_TASK`. Therefore, the task affinity is set to the correct app and the attack is prevented [18].

3 ADVERSARIAL MODEL AND REQUIREMENT ANALYSIS

In this section we present our attacker model, elaborate on assumptions and specify requirements for our detection method.

Adversary Model Our adversarial model is inherited from StrandHogg attacks – we consider an adversary who developed a malicious app, which, in turn, was downloaded by the victim's user and installed on Android 4.4 to 10 versions. The malicious app requests no permissions and does not collaborate with any other (potentially malicious) apps installed on the same platform. Furthermore, the attacker has no control of the underlying OS – hence, he is limited by restrictions imposed by Android's security mechanisms (e.g., Sandboxing and permission enforcement). The attacker's primary goal is to steal security-sensitive user input (such as passwords), which is achieved through deception and luring the user to enter sensitive input into a malicious activity, while the user believes he communicates with the benign app.

Goals The goal of our detection method is to detect the StrandHogg attack. Once detected, the user might be required to execute a password reset, which prevents account stealing. Alternatively or additionally, the user could be informed of the attack.

Requirements We formulate the following requirements for the detection method:

- **R1: Generalizability**. We require that the defense mechanism should be capable of defeating both StrandHogg attack versions. The development of separate defense strategies for each version is undesirable, as it would likely increase complexity and consumption of resources.
- **R2: Efficiency**. We also require the solution to be efficient in terms of run-time overhead and energy/memory consumption. Detection methods that impose high overhead have decreased chances for adoption.
- **R3: Compatibility**. Furthermore, we require that the detection method should not rely on any modifications of the operating system, since such modifications can only be implemented by (impossible to control) device vendors and even if implemented, often do not target older OS versions, leaving many devices in the field vulnerable. As such, our solution should be placed at the user-level rather than reside at the system-level.
- **R4: Usability**. We require that the method should not affect usability. In particular, it should not restrict or limit any existing functionality in Android.

In the following sections, we present two solutions, the one based on intrusion detection using machine learning and the second one based on dynamic attribute analysis. In Section 6, we will elaborate if these these methods fulfill the formulated requirements.

4 ML-BASED APPROACH

In this section, we explain our first attack detection method, which is based on intrusion detection using Machine Learning (ML).

General Idea The general idea of our first approach is to self-monitor app execution by the (potential victim) app, analyze execution traces – collected time series of monitored attributes – using machine learning (ML) methods, and identify suspicious behavior that might be indicative of the ongoing attack. The detection system must be individually implemented into each sensitive app, that needs to be protected. In particular, the victim app collects execution traces and transmits them to the server, where the traces are used for model training.

Methodology Our methodology includes the following steps: First, we constructed the dataset containing benign and malicious traces. Second, we trained and evaluated the model using the dataset and identified important features, and improved them through manual feature engineering. Third, we performed an additional evaluation of the best performing model to identify whether there is a correlation between model performance, attack version, and attack scenarios. The observations we make in this evaluation inspire us to build a second defense method which we present in Section 5.

4.1 Data Collection and Pre-Processing

Until now, there is no dataset available that includes execution traces of the StrandHogg attack. Hence, we implemented both StrandHogg attack versions and constructed our own dataset containing 900 benign and 1.032 malicious traces collected from a wide range of mobile devices and emulator instances.

DataCollector App The collection of data was done by implementing an Android app that will be referred to as DataCollector. The app collected 148 different attributes, using common Android API functions. Collected attributes included, for instance, tasks, intents, and application life cycle events. The collection of attributes was triggered every five seconds, which resulted in a time series of data, or trace. Furthermore, some attributes, like life cycle events, were collected every time a state change occurred. Intents were only collected once they have been received by the DataCollector. Each trace consists of several attribute-value pairs, i. e. observations.

Trace Collection StrandHogg v1 and v2 attacks were implemented in such a way that they target our DataCollector app. Therefore, the DataCollector corresponds to the victim app as shown in Figure 1 and Figure 2. Both attacks inject a malicious activity into the task structure of our DataCollector app. This simulates an ongoing StrandHogg attack as, for instance, used by password phishing attacks. We then proceeded to collect traces on different devices, such as Samsung Galaxy S2, Motorola G4, Xiaomi Redmi Note 9S, and Nokia 7.2, with Android versions ranging from 4.2 to 10. Additionally, we collected traces on Android emulators with versions ranging from 4.4 to 10.

(i) Benign traces Our data set contains two different types of traces. The first type is labeled as benign, those were collected without executing any attack and act as baselines. Normal app usage was simulated while collecting those traces, for instance, minimizing and resuming to the app via the recent apps list. The duration of the traces was roughly 30 seconds. Our benign data set contains 900 traces.

(ii) StrandHogg v1 traces The second type is labeled malicious, hence, those traces were collected while a StrandHogg attack was executed. We collected traces for both StrandHogg attacks. For the StrandHogg v1 attack we considered two scenarios, as we thought they may result in different attribute values in collected traces.

In the first scenario, traces were collected when the DataCollector was in a minimized state, i.e., the activity was still present in the recent apps list. In the second scenario, traces were collected with the DataCollector closed. Note that even though the app was closed,

the collection of attributes was continued, since the collection was implemented as a foreground service.

Our dataset includes 586 traces collected during a StrandHogg v1 attack with minimized DataCollector. For the scenario with a closed DataCollector we collected 242 traces.

(iii) StrandHogg v2 traces Due to the nature of the StrandHogg v2 attack, it is not possible to launch the attack with a closed DataCollector, as the attack starts the victim app in a minimized state in the first step. Furthermore, we could not collect StrandHogg v2 traces for Android 10 devices, since Android 10 is not anymore vulnerable (cf. Section 2). Hence, our collected StrandHogg v2 attack traces consider the scenario with the minimized DataCollector. There are 204 StrandHogg v2 traces in total that were collected on emulators with Android platforms ranging from 4.4 to 9 versions.

Data Pre-processing Before using our data set as training data for an ML model, we needed to transform the data set. This preprocessing was done using python scripts. Those merge all traces into a pandas data frame, a two-dimensional data structure provided by the pandas framework [30], while preserving the correct labels. Furthermore, we used *Label Encoders* to convert categorical values into numerical ones. We removed the timestamp values from our data set, as they most likely would have contributed to over-fitting. In case certain column values were missing, we replaced them with a default value.

4.2 Feature Engineering and Model Training

To build the best performing model, we had to go through an iterative process of model training, evaluation, and feature engineering to identify attributes that contributed the most to the classification results.

Model Training We split the entire data into two separate frames, one consisting of 75% of our complete data set and intended for training, and the second one holding 25% of data and used in the evaluation. During the process of splitting the data, we ensured, that the evaluation set contains traces from both attacks and both scenarios.

We then used two classifiers, *XGBoost* and *Random Forest*, to train models using the first data frame. Both classifiers were trained using default parameters provided by the *scikit-learn* framework. We trained the models to perform binary classification, therefore each observation from the traces were classified into either benign or malicious. Classification performance was evaluated using the second data frame holding 25% of data, that were not used in the training step. The results are depicted in Figure 3. The *XGBoost* performance is slightly better than the *Random Forest* performance, therefore, we focused on the *XGBoost* classifier for further testing.

Feature Analysis To identify important features, we generated a feature importance distribution. The generation of feature importance distributions is a standard benefit when using decision tree methods. In this ranking, we noticed that out of 148 attributes, the attribute `android:numActivities` had, by far, the highest contribution (cf. Appendix Table 2). By evaluating this attribute manually, we discovered that it is usually either set to one or two in case of an attack. Interestingly, the value was never set to zero for both benign and malicious traces. We discovered that the `android:numActivities` attribute only works as expected in case

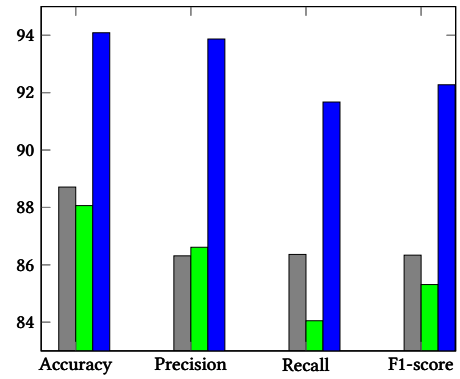

Figure 3: XGBoost and Random Forest classifier results. The y-axis shows percentage results. The gray/green bar chart shows the XGBoost/Random Forest results, respectively, before adding the manually engineered feature (`globalNumActivities`). The blue bar chart shows XGBoost classification results with the additional feature.

the Data Collector is in a minimized state. In case the Data Collector is closed the attribute is always set to one, regardless of an ongoing attack. Other features coincidentally divide the data by detecting design differences between the Data Collector and malicious apps or distinguish different devices by, for instance, their display identifier or screen height. Other attributes such as task identifier are not reliable for detection as they are not persistent between consecutive runs.

Feature Engineering and Classification Improvement Based on previous observations, we engineered an additional feature, based on `android:numActivities` attribute. In particular, it was set to one, for each observation in the entire trace, in case `android:numActivities` was greater or equal two at least one time in the trace. This means, in case one or more observations have an increased `android:numActivities` value, all observations in the entire trace will have value one for the newly added feature, otherwise the value is set to zero. Using this additional feature, we were able to additionally improve the performance of our classifier as seen in Figure 3. We refer to the newly added feature as `globalNumActivities`. It has an importance of 57.68%. The accuracy of the classifier improved by 5.38%, the Precision improved by 7.56%, the Recall improved by 5.31%, and the F1-score improved by 5.94%.

4.3 Additional Evaluation of the Best Performing Model

Our best performing model, as described in previous paragraphs, is the one trained using *XGBoost Classifier* and when using the `globalNumActivities` feature. The goal of the additional evaluation is to identify if the model performs similarly well when detecting various attack versions and/or scenarios. To perform such evaluation, we again used our second data frame which was not used in training. It was randomly sampled to select 50 benign traces and 50 traces of each attack type and scenario. These samples were then combined into three validation datasets: (1) 50 benign traces and 50 StrandHogg v1 traces with minimized DataCollector, (2) 50 benign traces and 50 StrandHogg v1 traces with closed DataCollector, (3) 50 benign traces and 50 StrandHogg v2 traces. Each of those validation

set samples were individually classified. Our performance metrics, depicted in Figure 4, show that the classifier is able to achieve very good scores when classifying StrandHogg v1 traces that were collected in a minimized state. The classification performance for StrandHogg v2 traces is very good as well. However, for traces that were collected in a closed state, the classification performance significantly drops. For instance, the accuracy drops by 12.45% compared to the StrandHogg v2 classification accuracy. The low accuracy for closed app state is due to the `android:numActivities` attribute only working reliably in a minimized app state.

The very good performance we observed for both StrandHogg versions for the scenario with the minimized app state inspired us to develop a second defense approach based on `android:numActivities` and minimized app state, which we discuss in the next section.

5 SECOND APPROACH: ACTIVITY COUNTER

In this section, we describe our second detection method, which we call ActivityCounter. It performs detection locally on the device and is efficient, deterministic, and yields 100% detection with no false positives. The idea behind the detection method emerged during the evaluation of our ML-based method and is based on monitoring the `numActivties` attribute and forcing one activity of the defending app to stay in a minimized state. The detection method must be integrated into each app that needs to be protected, for instance, online banking apps or apps that are potential targets for phishing attacks. Furthermore, the detection method can be used vendor-independently on any Android device.

5.1 Detection Method using Activity Count

First, we aim to gain an understanding, why the `numActivties` attribute becomes a better attack indicator when the app is forced to stay in minimized state.

We recall, that both versions of the StrandHogg attack inject a malicious activity into the task structure of the victim app. We previously observed, that the internal amount of activities (activity count), in the current process, can be accessed. The first step is to get a list of `RunningTaskInfo` elements via the method `getRunningTasks(int maxNum)`. According to the Android documentation, the

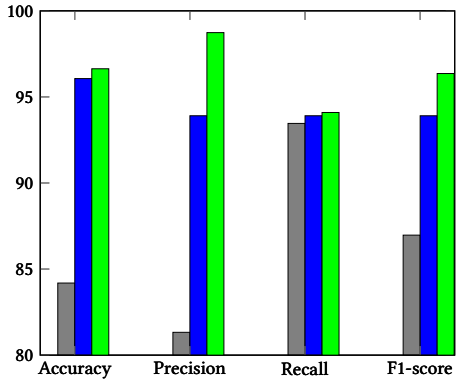

Figure 4: StrandHogg v1 trace (Test-Set) classification results for closed (gray) and minimized DataCollector (blue). Y-axis shows percentage results. Green bar chart shows StrandHogg v2 trace (Test-Set) results.

method is deprecated as of Android API level 21, which corresponds to Android version 5.0. However, for backwards compatibility reasons, the method still works up to Android 10. The documentation states: "For backwards compatibility, it will still return a small subset of its data: at least the caller's own tasks, and possibly some other tasks such as home that are known to not be sensitive" [10]. Each returned task inherits the attribute `numActivities` from the `TaskInfo` class. This attribute specifies the number of activities in the particular task [15]. Furthermore, we observed that this attribute only works reliably in a range above one, as described in Section 4.2. Hence, the attribute only works reliably if at least one activity is present, i.e., the app is minimized. That implies the attribute will never drop to zero.

We could verify that the number of activities accessed via the `numActivities` attribute increases by one, in case the app is minimized and a StrandHogg attack is launched. However, due to Android's internal logic, this attribute will not drop to zero. That means, in case no activity is present, the value will still be set to one. Furthermore, in case of an attack, the attribute will again be one, which is indistinguishable from the state of origin. To make this attribute a perfect indicator for an ongoing attack, one needs to guarantee that at least one activity is always present.

In case this condition is met, the StrandHogg attacks could be reliably detected – one would need to count activities and, in case the activity count is greater than the amount of activities that were legitimately created, one can infer that a StrandHogg attack was launched against the app.

5.2 Technical Challenges

In the previous sections, we elaborated on how the activity count could be used for attack detection. However, we face two technical challenges to realize this approach which we describe in the following.

CH1: Efficient Monitoring First of all, continuous background execution is necessary to periodically validate the amount of activities in our task. It is challenging to realize such monitoring efficiently – constant monitoring is prohibitively expensive and is likely to have a noticeable negative impact on the battery life of mobile devices. This problem violates the requirement **R2: Efficiency** described in Section 3.

CH2: No App Close The user is not allowed to close the application, as the `numActivities` attribute is only a reliable indicator if at least one activity of the app is opened/minimized. Preventing the app from being closed may have a high impact on the usability of the device, which does not go along with **R5: Usability** requirement (cf. Section 3).

5.3 Solving Challenges

In this section, we present our solutions to the previously described technical challenges.

Our idea for solving the **CH1: Efficient Monitoring** challenge is to place a hook that notifies our detection system in case the number of activities changed in the particular task. Finding the appropriate place in code to hook in is not straightforward, since we need to eliminate any changes to the operating system to fulfill the requirement **R3: Compatibility**.

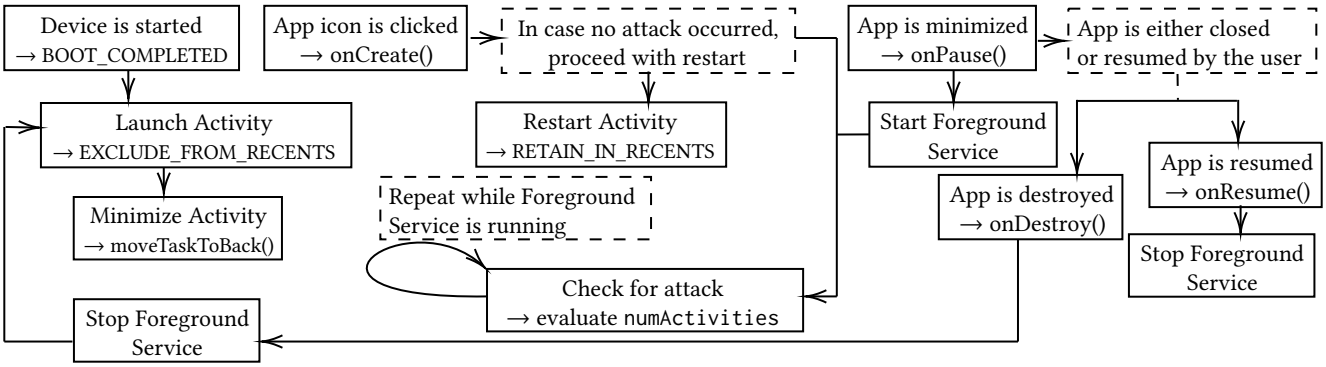

Figure 5: StrandHogg Attack detection Concept

The key to solving the second challenge **CH2: No App Close** is to keep a minimized activity instance. To fulfill the requirement **R5: Usability**, the minimized activity should not be displayed to the user. In the following paragraphs, we present our solutions to the challenges.

Launching minimized activity We could launch a minimized activity unnoticeable to the user by using the flag `FLAG_ACTIVITY_ EXCLUDE_FROM_RECENTS`. This flag prevents the activity from being displayed in the recent apps list [12]. As soon as the activity is created, it will instantly be sent to the background, i.e., into a minimized state, by calling the method `moveTaskToBack(boolean)` [8]. As the activity specified the exclude from recents flag, it will lead the Android OS to destroy the activity. However, a reference to the minimized activity is still kept in the Android OS, for the case if the app is opened again.

We could verify that the placed reference is sufficient to permanently increase the `numActivities` attribute by one. Therefore, by placing this minimized activity reference, every time the user closes the last activity instance, we can ensure that activity count based on the `numActivities` attribute works reliably.

Hooking the code for efficient monitoring Ideally, we want to place a hook into the Android OS, which notifies us of changes in the amount of activities in a particular task. However, to avoid any changes to the operating system, we had to come up with a workaround.

We observed an interesting behavior when placing minimized activity instance references, as described in the previous paragraphs – the placed activity reference automatically goes into a destroyed state, once it is minimized. However, every time the user clicks on the app's icon, the previously destroyed activity is re-created. That implies, that the `onCreate()` method of the activity is called. This is due to the task being revived and as the activity was destroyed earlier, it must now be re-created to be available to the user again. This behavior is very fortunate for our detection system, as we now have a callback each time the user clicks on our app's icon. We observed, that the `onCreate()` method of the destroyed activity will be called regardless of the foreground activity of the task. Therefore, even in case an adversarial activity is in the foreground, the `onCreate()` method will still be called.

We can now count the number of activities in our task via the `numActivities` attribute, each time the user clicks our app's icon, i.e., when the `onCreate()` method is called. Based on these observations we can now build the final detection method.

5.4 Detection Method Concept

In this section, we describe the concept of the novel detection method, based on the solutions we came up with, in the previous paragraphs. We describe the approach for two scenarios. The first scenario considers the closed app state, here we describe how a callback is placed to get notified of potential attacks. The second scenario considers the app in a minimized state, here we describe how to periodically check for ongoing attacks.

Detection while the app is closed As previously elaborated, we start by describing the detection technique while the app is in a closed state. The concept is visualized in Figure 5. The essential step is to guarantee that the app always receives a `onCreate()` callback, in case the app's icon is clicked. This callback needs to work independently of the current foreground activity in the task. This is achieved, as described in Section 5.3, by placing a minimized activity reference. We place the reference when the device is started, to guarantee that our detection method is launched at the earliest possible time. It works by registering for the Android broadcast `BOOT_COMPLETED` and launch the minimized activity as soon as the broadcast is received. This can be seen in the step *Device is started* in Figure 5. Now every time the user launches the app, by clicking on the app's icon, the `onCreate()` method of the destroyed activity will be called. This process corresponds to the step *App icon is clicked* in the Figure. In this callback, we check for an ongoing attack, as specified in Section 5.1. Therefore, we evaluate if the `android:numActivities` attribute is greater than the amount of our own created activities. If no attack is detected, we need to restart the app with the flag `FLAG_ACTIVITY_RETAIN_IN_RECENTS` which is the inverse of the flag `FLAG_ACTIVITY_EXCLUDE_FROM_RECENTS`. Otherwise, the user would not be able to see the activity in the recent apps list, which drastically reduces the user experience. Therefore, the flag does not affect the attack detection but is critical to maintaining the usability of the defense method.

Detection while the app is minimized In this paragraph, we describe attack inference while the app is in a minimized state. Assume no attack occurred and the app is in the foreground. The user can minimize the application, at any time, and start another app. This corresponds to step *App is minimized* in Figure 5. The other app potentially launches a StrandHogg attack. The malicious activity

will then be displayed, once the user resumes our app via the recent apps list. That means, we need to periodically check whether an attack occurred, while the app is intentionally minimized *by the user*. We can do that by starting a foreground service, which periodically evaluates the `android:numActivities` attribute. Empirically we chose to evaluate the attribute every 2 seconds, as we were able to detect every attack by using this interval.

In case no attack occurred, we need to handle two different possibilities. Either the app is resumed by the user or it gets closed. In case of resuming, the activity's `onResume()` method will be called. This process is described in the step *App is resumed* in the Figure. If this callback is received, we stop the foreground service. However, in case the user does not resume to the activity but rather closes it, we need to place our activity reference again. We observed that the `onDestroy()` method of our activity is always called, in case there is a running foreground service present. This step is illustrated in the Figure as *App is destroyed*. Therefore, we can rely on this method to detect when the user closes our activity. In case the activity's `onDestroy()` method is called, we stop the foreground service and place the minimized activity reference.

Detection with multiple activities In case the app needs multiple activities, we need to guarantee that the detection system can not be bypassed by launching a different activity than the main activity. We can ensure that the adversary needs to attack the main activity of our app, by protecting each following activity with a custom signature permission. Therefore, only our app is allowed to launch an activity different from the app's main activity.

Necessary Permissions In this paragraph, we describe the necessary permissions for the detection method. An overview of the necessary permissions can be seen in Appendix Table 3. To place the minimized activity at the earliest possible time, the app needs to receive a broadcast once the boot process is completed. This needs the `RECEIVE_BOOT_COMPLETED` permission. Furthermore, we need a running foreground service that periodically evaluates the number of activities in our task while the app is intentionally minimized by the user. To do that, one needs the `FOREGROUND_SERVICE` permission for Android versions 9.0 and higher. Furthermore, for the Android versions below 5.0, which corresponds to API level 21, the permission `GET_TASKS` is necessary to receive the list of running tasks [14]. According to the Android documentation, as of Android 10, which corresponds to API level 29, there are certain restrictions on starting activities while the app is in the background [19]. The restrictions are ignored when certain exceptions apply. One of those exceptions is that the app was granted the `SYSTEM_ALERT_WINDOW` permission by the user. Therefore, for Android 10 devices our method additionally needs this permission to be able to always place a minimized activity, even when being in the background.

5.5 Evaluation

In this section, we evaluate the effectiveness of our activity count method through developing a proof-of-concept implementation and empirical tests.

Proof-of-Concept Implementation We implemented the concept as described in Section 5.4 within an app we refer to as ActivityCounter. For development, we used Android Studio and the programming language Java. Our implementation has a main activity that handles the attack detection. Furthermore, we implemented one permission-protected activity as described in Section 5.4. We will provide the source code upon request.

Testbed We tested the implementation by use-case testing it on different emulator devices. Each of the tests were executed on emulator devices with versions: 4.4, 5.0, 6.0, 7.0, 8.0, 9.0, and 10.

Usability Tests The first test cases were performed to ensure the usability of the detection method and were executed while no attack was ongoing. Test cases included active interaction of the user with the app and included such actions as starting, minimizing, and closing the ActivityCounter app and switching between different app states. We additionally verified that the placed minimized activity reference is not visible in the recent apps list. Therefore, the user is not confused by some ActivityCounter activity still being in the recent apps list, despite closing the ActivityCounter app.

We noticed, that the placement of a minimized activity reference, in the case of ActivityCounter's closing, is sometimes observable by the user. This can be mitigated by using a (temporarily) transparent main activity. We applied this mitigation technique in the ActivityCounter implementation and the flickering was no longer visible on a 8 year old device (Samsung S4 Mini).

Effectiveness Evaluation The second line of tests was intended to ensure the reliability of the detection method. In this part of the evaluation, we launched both StrandHogg attack versions, that were implemented in such a way that they targeted our ActivityCounter app. We tested two attack scenarios:

- *Attack Scenario #1* - The ActivityCounter app is closed by the user. (Therefore, a minimized activity reference was placed.)
- *Attack Scenario #2* - The ActivityCounter app is intentionally minimized by the user. (Therefore, the foreground service is periodically evaluating the `android:numActivities` attribute.)

To test the first attack scenario, we launched the ActivityCounter app on the emulator device. We then proceeded to close the ActivityCounter app by opening the recent apps list and closing the ActivityCounter activity. (Pressing the back button to close the activity was considered as well.) In the next step, the StrandHogg v1 attack was launched. After closing the distraction activity we opened the ActivityCounter app by clicking on the ActivityCounter app's icon. We could verify that the attack was successfully detected. We repeated the evaluation for the StrandHogg v2 attack. We could again verify that the attack was reliably detected.

The second attack scenario was tested by opening the ActivityCounter app and minimizing it via pressing the home button on the emulator device. We verified, that the foreground service is successfully started once the recent apps list is shown or the home button is pressed. We then proceeded to launch the StrandHogg v1 attack against our ActivityCounter app. We verified that the app can detect the attack reliably. We then repeated the test for the StrandHogg v2 attack and ensured that this attack version can also be reliably detected.

Reliability Tests Our further evaluation concerns the reliability of our ActivityCounter app, which might be affected by the OS that can kill background activities at any time, due to various reasons such

as memory shortage. During the test, the first step was to launch the ActivityCounter activity and close it right afterward, such that a minimized activity reference will be placed. We then retrieved a process list via the `adb shell` command `ps`. In this process list, we identified the *PID*, belonging to our ActivityCounter app. We then killed this process via the `kill` *PID* command. We then proceeded to test the reliability of the app by launching the StrandHogg v1 attack. We could verify that the killing of the ActivityCounter's app process is not sufficient to bypass our detection method, as the attack was successfully detected. We repeated this evaluation for the StrandHogg v2 attack as well and achieved the same results.

Resilience against attack evasion Our last line of tests targets the resilience of our detection method against evasion. One potential evasion strategy for the attacker would be to utilize the Android flag `FLAG_ACTIVITY_CLEAR_TASK`. The general idea behind this approach is to clear all activities from the victim task. This would remove the minimized activity reference from the task and therefore evade our detection scheme, as we no longer receive a callback once the app's icon is clicked. The Android documentation states: "This flag will cause any existing task that would be associated with the activity to be cleared before the activity is started. That is, the activity becomes the new root of an otherwise empty task, and any old activities are finished" [11]. The test was done by adding the `FLAG_ACTIVITY_CLEAR_TASK` flag to the launch intent of both StrandHogg attack versions. We could confirm, that the additional launch flag does not affect the reliability of the detection technique.

6 DISCUSSION

In this section, we explain, how our approaches fulfill the targeted requirements presented in Section 3. We start by discussing the requirements for the ML-based approach presented in Section 4. We then elaborate on how the ActivityCounter, presented in Section 5, fulfills the requirements.

ML-based Approach Our detection method based on machine learning is capable of detecting both StrandHogg attack versions. Hence, we consider the requirement **R1: Generalizability** fulfilled. However, the approach uses a background monitoring component that periodically collects attributes to detect ongoing attacks. Continuous background monitoring is expensive in terms of run-time overhead and energy consumption. The increased energy consumption may drain the battery of the device faster, which will result in a reduced user experience. Therefore, we consider the requirement **R2: Efficiency** not fulfilled. The approach only uses existing Android API methods to collect the attributes. This makes it applicable to a wide range of devices while being independent of any modification of the operating system. Due to this fact, we consider the requirement **R3: Compatibility** fulfilled. The last requirement **R4: Usability** is not fulfilled, as we previously discussed that the battery duration of the device may be reduced. Furthermore, the approach suffers from false positives and negatives, due to the `android:numActivities` attribute not working reliably for closed app state. This could lead to undetected attacks and unjustified password resets, hence negatively impacting security and usability.

ActivityCounter The novel detection method, called ActivityCounter, can detect both versions of the StrandHogg attack. Therefore, we consider the requirement **R1: Generalizability** fulfilled. Generally, this method does not require continuous background monitoring, as it is only necessary once the app is intentionally minimized by the user. The resource overhead of the background monitoring component is acceptable as it only validates if an attack occurred every two seconds. In our tests, the interval of two seconds was sufficient to detect all StrandHogg attacks against the ActivityCounter app. Furthermore, the malicious activity must be present on the screen for a sufficiently long time to successfully steal sensitive user input, such as passwords. The process of entering credentials often takes longer than two seconds, therefore, we consider the interval appropriate. In case the app is in the foreground or closed, background monitoring is not necessary as we placed a hook that notifies us in case the app is started. Usually, the app is either in the foreground or minimized for short times by the user, therefore, we consider the second requirement **R2: Efficiency** fulfilled as well. We further required that the detection method should not rely on any modification of the OS. Our approach works by only using common Android API methods, that are available on Android versions ranging from 4.4 to 10. Therefore, the defense technique is applicable to any vulnerable Android platform without the need of modifying the OS, as a consequence we consider the requirement **R3: Compatibility** fulfilled. The last requirement **R4: Usability** is fulfilled as well, as the detection method does not negatively impact the usability of the device, e.g. by draining the battery or suffering from false positives. As we elaborate in the next section, no other proposed defense method fulfills all the above-mentioned requirements.

Limitations One limitation of both detection methods is that it must be integrated manually into each sensitive app that desires to be protected. This approach is limited, since it only protects applications if app developers have chosen to integrate it. Also, this choice is not under control of end-users. Another limitation is that the detection methods are specific to the StrandHogg attacks. Further research is necessary to identify if the methods can be utilized to detect other types of activity hijacking attacks.

7 RELATED WORK

In this section, we overview related work on countermeasures against StrandHogg attacks. They can be classified into three categories: runtime detection, static analysis, and enforcement of attributes.

Runtime Detection Defense methods [27, 32] are based on the Xposed framework. In particular, Yan et al. [32] proposed a new security model "Android Window Integrity". This model specifies the capabilities for each app in the system and analyzes activity transitions. In case an activity transition is against the rules imposed by the security model, such as starting an activity while being in the background, the user is asked for confirmation. Ren et al. [27] identified the possibility of false positives when using the proposed "Android Window Integrity" security model. The authors proposed a new system called "ActivityShielder", that reduces the risk of false positives. It works by validating the integrity of the source that causes a switch between activities. By doing so, the entire foreground task integrity is validated.

Both techniques are capable of detecting activity hijacking attacks, hence, both StrandHogg versions can be detected. Those

	Prevents/Detects v1	Prevents/Detects v2	No negative app behaviour change	No usability impact	No False Positives	No OS modification required
Static Analysis [1, 20, 31]	✓	✗	✓	✓	✓	✓
Runtime Detection [27, 32]	✓	✓	✓	✓	✗	✗
Enforcement of Attributes [33]	✓	✓	✗	✗	✓	✓
ML-Based Approach	✓	✓	✓	(✗)	✗	✓
ActivityCounter	✓	✓	✓	✓	✓	✓

Table 1: StrandHogg Countermeasures Overview

methods, however, can not be implemented without vendor support as they both require modification of the OS.

Static Analysis This category of countermeasures relies on static analysis of the bytecode, resources, and permissions of potential malicious Android apps. Bianchi et al. [1] developed a tool intended for static analysis of Android apps. It works by decompiling the Dalvik ([16]) intermediate code as well as other app resources, in order to detect risks indicating a task hijacking attack. Additionally, a runtime defense mechanism is included, that uses an indicator in the navigation bar. Zhou et al. [31] proposed a solution intended to detect malicious apps by permission-based footprinting. Hwang et al. [20] proposed a novel tool to detect activity hijacking attacks, especially the first version of the StrandHogg attack, based on analyzing used intents and the `taskAffinity` attribute in the app's manifest.

The StrandHogg v1 attack can easily be spotted using static analysis techniques, which inspect the `android:taskAffinity` attribute. "Attackers exploiting StrandHogg have to explicitly and manually enter the apps they are targeting into Android Manifest" [26]. Therefore, the attack can be detected by checking if an app has a foreign package name set as task affinity. However, there are legitimate use cases for declaring task affinities to foreign applications, hence, the approach could lead to false positives.

The StrandHogg v2 attack can not be detected using static analysis. This is due to its independence of attributes. The attack does not require to specify any suspicious attributes in the app's manifest, prior to attack execution. Therefore, it is possible to launch the attack entirely in memory, which circumvents static analysis techniques.

Enforcement of Attributes Another mitigation for both StrandHogg attacks was proposed by Promon [33]. The idea of the defense method is setting the `android:launchMode` of the victim app to `singleInstance`. While maybe effective, this mitigation technique has severe usability drawbacks, which will likely confuse users, such as preventing multitasking or recreating the activity instead of resuming to it [7]. Furthermore, developers have to consider the changed app behaviour, as the functionality of the app is limited.

Summary We summarize the discussed countermeasures in Table 1. All already existing methods have drawbacks – for instance, the runtime detection requires OS modification, which can only be done with the support of the mobile platform vendors. However, many devices are no longer supported, therefore, those defense techniques can no longer be implemented. Other countermeasures, such as static analysis, are only capable of detecting the StrandHogg v1 attack. Moreover, countermeasures based on the enforcement of attributes are capable of defending against both StrandHogg attack versions, however, they negatively impact the behaviour of the app. Those behaviour changes have extremely severe consequences such as prevention of multitasking. Furthermore, those changes negatively affect the development process of an app, as changed behaviour has to be considered.

Our first approach, presented in Section 4, is based on machine learning and capable of detecting both StrandHogg attack versions. However, the approach suffers from false positives. Furthermore, the usability of the device is slightly reduced as continuous background monitoring is necessary, which may shorten the battery of the mobile device. We consider the approach more suitable than the *Enforcement of Attributes* solution, as there is no negative behaviour change, hence the implementation effort for developers is reduced. Furthermore, we consider the constant monitoring a less severe usability drawback than prevention of multitasking. Our second approach – ActivityCounter, presented in Section 5, aims at eliminating the drawbacks of the first approach. By placing a hook into the OS we were able to get rid of the continuous background monitoring. The ActivityCounter is capable of reliably detecting both StrandHogg attack versions without any false positives. Furthermore, it does not require any modification of the operating system, as the approach only relies on native Android API methods.

8 CONCLUSION

The StrandHogg attacks affect up to 90% of all Android devices around the globe and existing countermeasures cannot be straightforwardly applied for their protection – they either require OS modification, or have poor usability, or are helpless against some attack variations.

In this work, we aimed at a challenging goal to design a generic, efficient and usable method that does not require vendor support for deployment. To achieve our goal, we studied, analyzed, and implemented both StrandHogg attack versions. Using the gained knowledge, we were able to develop two novel detection techniques. The first technique, based on machine learning, is capable of detecting attacks with an average F1 score of 92 percent. It works by collecting traces on Android devices and classifying them with the help of the developed ML model. The second technique is based on counting activities in the particular task and shows outstanding performance – no false positives, no false negatives, and no drawbacks such as negative usability impact or requirement of vendor support. It can be applied to all vulnerable Android versions ranging from 4.4 to 10 and can detect all attack versions.

Overall, ActivityCounter is the first practical method that can protect apps on millions of vulnerable Android devices against StrandHogg attacks.

REFERENCES

[1] A. Bianchi, J. Corbetta, L. Invernizzi, Y. Fratantonio, C. Kruegel, and G. Vigna. 2015. What the App is That? Deception and Countermeasures in the Android User Interface. In *2015 IEEE Symposium on Security and Privacy*. 931–948. https://doi.org/10.1109/SP.2015.62

[2] Qi Alfred Chen, Zhiyun Qian, and Z. Morley Mao. 2014. Peeking into Your App without Actually Seeing It: UI State Inference and Novel Android Attacks. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, San Diego, CA, 1037–1052. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/chen

[3] Petr Dvorak. 2020. *StrandHogg 2.0: Explained.* https://www.youtube.com/watch?v=avElCFVuXvo – 17:00.

[4] A. P. Felt and D. Wagner. 2011. Phishing on Mobile Devices. In *IEEE Workshop on Web 2.0 Security and Privacy*.

[5] Earlence Fernandes, Qi Alfred Chen, Justin Paupore, Georg Essl, J. Alex Halderman, Z. Morley Mao, and Atul Prakash. 2016. Android UI Deception Revisited: Attacks and Defenses. In *Financial Cryptography and Data Security*.

[6] Google. 2019. *Android Documentation - Understand Tasks and Back Stack.* Retrieved Feb 11, 2021 from https://developer.android.com/guide/components/activities/tasks-and-back-stack

[7] Google. 2020. *Android Documentation - <activity>.* Retrieved Feb 11, 2021 from https://developer.android.com/guide/topics/manifest/activity-element

[8] Google. 2020. *Android Documentation - Activity.* Retrieved Feb 17, 2021 from https://developer.android.com/reference/android/app/Activity#moveTaskToBack(boolean)

[9] Google. 2020. *Android Documentation - App Manifest Overview.* Retrieved Feb 11, 2021 from https://developer.android.com/guide/topics/manifest/manifest-intro

[10] Google. 2020. *Android Documentation - getRunningTasks.* Retrieved Feb 15, 2021 from https://developer.android.com/reference/android/app/ActivityManager#getRunningTasks(int)

[11] Google. 2020. *Android Documentation - Intent (FLAG_ACTIVITY_CLEAR_TASK).* Retrieved Mar 8, 2021 from https://developer.android.com/reference/android/content/Intent.html#FLAG_ACTIVITY_CLEAR_TASK

[12] Google. 2020. *Android Documentation - Intent (FLAG_ACTIVITY_EXCLUDE_FROM_RECENTS).* Retrieved Feb 17, 2021 from https://developer.android.com/reference/android/content/Intent#FLAG_ACTIVITY_EXCLUDE_FROM_RECENTS

[13] Google. 2020. *Android Documentation - Intent (FLAG_ACTIVITY_NEW_TASK).* Retrieved Feb 11, 2021 from https://developer.android.com/reference/android/content/Intent#FLAG_ACTIVITY_NEW_TASK

[14] Google. 2020. *Android Documentation - Manifest.permission.* Retrieved Mar 8, 2021 from https://developer.android.com/reference/android/Manifest.permission#GET_TASKS

[15] Google. 2020. *Android Documentation - TaskInfo.* Retrieved Feb 15, 2021 from https://developer.android.com/reference/android/app/TaskInfo#numActivities

[16] Google. 2020. *Android Runtime (ART) and Dalvik.* Retrieved Feb 11, 2021 from https://source.android.com/devices/tech/dalvik

[17] Google. 2020. *Android Security Bulletin - Mai 2020.* Retrieved Feb 13, 2021 from https://source.android.com/security/bulletin/2020-05-01

[18] Google. 2020. *GoogleGit - ActivityStarter.java.* Retrieved May 17, 2021 from https://android.googlesource.com/platform/frameworks/base/+/a952197bd161ac0e03abc6acb5f48e4ec2a56e9d

[19] Google. 2020. *Restrictions on starting activities from the background.* Retrieved Mar 8, 2021 from https://developer.android.com/guide/components/activities/background-starts

[20] Sungjae Hwang, Sungho Lee, and Sukyoung Ryu. 2020. All about activity injection: Threats, semantics, detection, and defense. *Software: Practice and Experience* 50 (01 2020). https://doi.org/10.1002/spe.2792

[21] C. C. Lin, H. Li, X. Zhou, and et al. 2014. ScreenMilker: How to Milk Your Android Screen for Secrets. In *Network and Distributed System Security Symposium*.

[22] B. Liu, S. Nath, R. Govindan, and J. Liu. 2014. DECAF: Detecting and Characterizing Ad Fraud in Mobile Apps. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[23] D. Liu, E. Cuervo, V. Pistol, R. Scudellari, and L.P. Cox. 2013. ScreenPass: Secure Password Entry on Touchscreen Devices. In *Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*.

[24] S. O'Dea. [n.d.]. *Mobile Android operating system market share by version worldwide from January 2018 to January 2021.* Retrieved March 13, 2021 from https://www.statista.com/statistics/921152/mobile-android-version-share-worldwide/

[25] Promon. 2019. *The StrandHogg vulnerability.* Retrieved Feb 11, 2021 from https://promon.co/security-news/strandhogg/

[26] Promon. 2020. *StrandHogg 2.0 - The 'evil twin'.* Retrieved Feb 13, 2021 from https://promon.co/strandhogg-2-0/

[27] C. Ren, Peng Liu, and S. Zhu. 2017. WindowGuard: Systematic Protection of GUI Security in Android. In *NDSS*.

[28] Chuangang Ren, Yulong Zhang, Hui Xue, Tao Wei, and Peng Liu. 2015. Towards Discovering and Understanding Task Hijacking in Android. In *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, Washington, D.C., 945–959. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/ren-chuangang

[29] Z. Wang, C. Li, and et al. Y. Guan. 2016. ActivityHijacker: Hijacking the Android Activity Component for Sensitive Data. In *International Conference on Computer Communication & Networks*.

[30] Wes McKinney. 2010. Data Structures for Statistical Computing in Python. In *Proceedings of the 9th Python in Science Conference*, Stéfan van der Walt and Jarrod Millman (Eds.). 56 – 61. https://doi.org/10.25080/Majora-92bf1922-00a

[31] Wu Zhou Xuxian Jiang Yajin Zhou, Zhi Wang. 2012. Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. *North Carolina State University* (2012). https://www.researchgate.net/publication/267787299_Hey_You_Get_Off_of_My_Market_Detecting_Malicious_Apps_in_Official_and_Alternative_Android_Markets

[32] F. Yan, Y. Li, and L. Zhang. 2018. ActivityShielder: An Activity Hijacking Defense Scheme for Android Devices. In *2018 27th International Conference on Computer Communication and Networks (ICCCN)*. 1–9. https://doi.org/10.1109/ICCCN.2018.8487367

[33] Wander Z. 2020. *StrandHogg 2.0 Exploit Explained – Why Users and Android App Developers should care.* Retrieved Feb 27, 2021 from https://www.xda-developers.com/strandhogg-2-0-android-vulnerability-explained-developer-mitigation/

A APPENDIX

A.1 Section 4 – Feature Importance

Feature	Importance (in %)
numActivities	**23,57%**
displayId	7,43%
isFullscreen	7,05%
baseIntent:mForceLaunchOverTargetTask	6,50%
isConventionalMode	6,30%
configuration:screenHeightDp	4,57%
userId	4,50%
configuration:seq	4,35%
taskId	3,05%
resizeMode	2,87%

Table 2: *XGBoost Classifier* feature importance ranking.

A.2 Section 5 – Required Permissions

Android Version	Required Permissions
4.4	RECEIVE_BOOT_COMPLETED GET_TASKS
5.0, 6.0, 7.0, 8.0	RECEIVE_BOOT_COMPLETED
9.0	RECEIVE_BOOT_COMPLETED FOREGROUND_SERVICE
10	RECEIVE_BOOT_COMPLETED FOREGROUND_SERVICE SYSTEM_ALERT_WINDOW

Table 3: Necessary permissions of ActivityCounter for different Android versions.