# Hit by the Bus: QoS Degradation Attack on Android

Mehmet Sinan İnci
Worcester Polytechnic Institute
msinci@wpi.edu

Thomas Eisenbarth
Worcester Polytechnic Institute
teisenbarth@wpi.edu

Berk Sunar
Worcester Polytechnic Institute
sunar@wpi.edu

## ABSTRACT

Mobile apps need optimal performance and responsiveness to rise amongst numerous rivals on the market. Further, some apps like media streaming or gaming apps cannot even function properly with a performance below a certain threshold. In this work, we present the first performance degradation attack on Android OS that can target rival apps using a combination of logical channel leakages and low-level architectural bottlenecks in the underlying hardware.

To show the viability of the attack, we design a proof-of-concept app and test it on various mobile platforms. The attack runs covertly and brings the target to the level of unresponsiveness. With less than 10% CPU time in the worst case, it requires minimal computational effort to run as a background service, and requires only the UsageStats permission from the user. We quantify the impact of our attack using 11 popular benchmark apps, running 44 different tests. The measured QoS degradation varies across platforms and applications, reaching a maximum of 90% in some cases. The attack combines the leakage from logical channels with low-level architectural bottlenecks to design a malicious app that can covertly degrade Quality of Service (QoS) of any targeted app. Furthermore, our attack code has a small footprint and is not detected by the Android system as malicious. Finally, our app can pass the Google Play Store malware scanner, Google Bouncer, as well as the top malware scanners in the Play Store.

## Keywords

Mobile Security; QoS Attack; Mobile Malware; Performance Degradation

## 1. MOTIVATION

Smartphones are now integrated into all facets of our lives—facilitating our daily activities from banking to shopping and from social interactions and to monitoring our vital health signs. These services are supported by numerous apps built by an army of developers. The mobile ecosystem is growing at an astounding rate with more than 2.4 million apps as of September 2016 [3], running on billions of devices. According to [5], more than a million new Android devices are activated worldwide, downloading billions of apps and games each month. Moreover, app revenue totaled to 45 billion dollars in 2015, proving a lucrative business. App developers big and small are under enormous competition trying to get a foothold in this growing market and a share of the huge revenue. As expected, there is fierce competition amongst competing apps with similar functionality trying to earn a top ranking in the app store.

In such a cutthroat market, competing app vendors have a strong incentive to cheat to get ahead in the competition. In the mobile app market, a strong delivery channel is the app store where apps which have received high user ratings are featured on the main page, and ones with low ratings are essentially buried in the listings and thus have become invisible to the users. Therefore, if an app developer can force a negative user experience during the use of a competitor's app, that could essentially render the competing app invisible.

To prevent malicious interference mobile platform vendors commonly implement app level sandboxing. For instance, on the Application Fundamentals web page, Google states that: "Each process has its own virtual machine (VM), so an app's code runs in isolation from other apps." [6]. While the first part of this statement is correct, the second part is not. Android apps running on a system share the same underlying physical hardware and therefore are not truly isolated from each other. This lack of physical isolation can be exploited, giving an app advantage over a competitor's app. Victim app's operations can be manipulated, degraded or even brought to a halt. This would be particularly destructive in real-time applications such as trading, online gaming, and live streaming.

In this work, we show that such an exploit is indeed practical. We introduce a technique that exploits architectural bottlenecks, in combination with logical channel leakages to degrade the performance of victim apps. The attack app does not require any root or peripheral access privileges as needed in sensor-enabled attacks. Also, the performance footprint of the attack code is very low, the attack is not detected by any malicious code monitor. We quantify the performance degradation with wide variety of benchmarks on various platforms. Since our attack vector only employs a widely used feature of modern microprocessors, i.e. memory bus locking, the degradation attack is hard to detect and mitigate. Furthermore, our app passed the Google Play

*ASIA CCS '17, April 02-06, 2017, Abu Dhabi, United Arab Emirates*
© 2017 ACM. ISBN 978-1-4503-4944-4/17/04…$15.00
DOI: http://dx.doi.org/10.1145/3052973.3053028

Store malware scan and an additional 23 of the most popular malware scanners listed in the Play Store. This shows evidence that new threats do not necessarily fit into traditional malware definitions and require more in-depth analysis with a broader perspective.

## Our Contribution

This work presents and explores all the necessary steps to successfully implement a Quality of Service (QoS) attack on mobile devices running Android OS. Specifically, in this paper,

- We present the first QoS attack on Android OS that combines architectural bottlenecks and logical channel leakages to significantly degrade the performance of a victim app.
- We show that our attack is stealthy and hard to mitigate by showing that it cannot be detected by the Android OS, Google Play Store malware scan or malware scanner apps. Further, the attack exploits the memory bus locking, a widely used feature of modern microprocessors hence is hard to mitigate.
- We test and quantify the QoS degradation caused by our attack using the most popular benchmarks in the Play Store.

## 2. BACKGROUND

In this section, we provide the necessary background information to better understand our attack. More specifically, we go over the Android permission management system and atomic instructions.

### 2.1 Android Permission System

Android employs permission management system to give apps access to various specific operations, sensors and data. While some of these permissions require user consent, others do not. Before Android 6.0 (API 23), the permission consent was set to be given at install time. Granting permissions at the install time meant that apps listed all the permissions that they might require during execution and that the users raced to the install button without checking them. After Android 6.0, permission requests are presented when the app needs that specific permission for the first time. When the user is prompted, he/she can still deny the permission even though the app is already installed and running. In cases when the user declines to give the requested permission, some applications may keep working while others crash outright or prompt the user until the permission is granted. However, not all permissions require user consent. Depending on the importance of information or service that an app needs to access, permission might be granted automatically without prompting the user. In the following, we explain these different types of permissions and how they are handled by the Android operating system.

#### *2.1.1 Normal Permissions*

Normal permissions are used when apps require access to data and resources that are not deemed sensitive by Android. For instance, setting an alarm requires only a Normal Permission and does require user consent since there is no serious danger of a privacy leak by setting up an alarm. Because of this, Normal Permissions do not require explicit user consent and are automatically granted by the system. Note that the user can always review which permissions an app uses, normal or not. While deemed safe and trusted, normal permissions can also affect the operation of other apps. For instance, the `KILL_BACKGROUND_PROCESSES` permission allows an app to shut down other apps using only their package name. While this permission does not allow access to any sensitive data, it gives crucial control over other apps in the system.

#### *2.1.2 Dangerous Permissions*

As explained in [1], any permission that is needed to access a sensitive, private data of the user or a service of the device, is classified as Dangerous Permission. The data or the service that these permissions allow access to are sensitive, and therefore require explicit user consent. For instance, using the device camera requires a Dangerous Permission since an app can access the camera and take photos when the user did not intend to do so. In comparison to Normal Permissions, this type of permissions clearly carry a higher risk of privacy breach and has to be granted carefully. The following permission groups are considered dangerous permission by the Android and require explicit user consent; Calendar, Camera, Contacts, Location, Microphone, Phone, Sensors, SMS, and Storage.

#### *2.1.3 Signature Level Permissions*

These permissions are granted by the system to apps only if the app requesting the permission has the same signature as the app that declared that permission. If these signatures match, the system grants the permission without prompting the user. Signature Level Permissions are vendor dependent and are generally closed by hardware vendors.

### 2.2 Atomic Operations

Atomic operations are defined as indivisible, uninterrupted operations that appear to the rest of the system as instant. While operating directly on memory or cache, atomic operation prevents any other processor or I/O device from reading or writing to the operated address. This isolation ensures computational correctness and prevents data races. While instructions on single thread systems are automatically atomic, there is not guarantee of atomicity for regular instructions in multi-threaded systems. In these systems, an instruction can be interrupted or postponed in favor of another task. Hence the atomic operations are especially useful for multi-threaded systems and parallel processing.

In order to ensure atomicity, platforms have different techniques. In old x86 systems, processor always locks the memory bus completely until the atomic operation finishes, whether the data resides in the cache or in the memory. While ensuring atomicity, the process also results in a significant performance penalty to the system. In newer systems prior to Intel Nehalem and AMD K8, memory bus locking was modified to reduce this penalty. In these systems, if the data resides in cache, only the cache line that the processed data resides on is locked. This, ‘cache lock’ results in a very insignificant system overhead compared to the performance penalty of memory bus locking. However, when the data surpasses cache line border and resides in two cache lines, more than a single cache line has to be locked. In order to do so, memory bus locking is again employed.

After Intel Nehalem and AMD K8, shared memory bus was replaced with multiple buses with non-uniform memory access bridge between them. While the system gets rid of the memory bottleneck for multiprocessor systems, it also invalidates memory bus locking. When a multi-line atomic cache operation needs to be performed, all CPUs has to coordinate and flush their ongoing memory transactions. This emulation of memory bus locking results in a significant performance hit and is the underlying mechanism exploited in this work.

ADD, AND, CMPXCHG and XOR are some of the instructions defined in x86 architecture that can be executed atomically with a lock prefix. Also, XCHG instruction executes atomically when operating on a memory location, regardless of the LOCK use.

On the ARM side of things, there are atomic instructions available in user space as in x86. Prior to ARM v6, SWP instruction was used to provide atomic read and writes. Later, ARM v6k and ARM v7 introduced the `LDREX` and `STREX` instructions to split the atomic memory update into two pieces and ensure atomicity [25]. When an atomic memory update has to be executed, first the `LDREX` instruction is called to load a word from the memory and tag the memory location as exclusive. This operation immediately notifies the `exclusive monitor`, a simple state machine with two states; open and exclusive [2]. After the memory location is tagged as exclusive, only the parties allowed by the `exclusive monitor` can store data to this location. If any other process or user attempts to store data to the location, the request is denied and an error state is returned. After the data is updated outside of the memory and the updated data needs to be stored, the `STREX` instruction is called to conditionally store to the memory location, condition being the right to store to the location.

**Exotic Atomic Operations:** Exotic Atomic Operations are ones that work on uncacheable memory and trigger system-wide memory bus locking. The fact that some addresses are uncacheable can be due to data in the operand spanning multiple cache or memory lines as in the case of a word-tearing or it can be due to the operand address corresponds to a reserved space on the physical memory. In any case, an Exotic Atomic Operation triggers memory bus locking or flushing of the ongoing memory operations in all of the CPUs of the system to ensure atomicity and data coherence. As expected, this results in heavy performance penalty to the overall system, especially the memory transactions. Moreover, since instructions might take different clock cycles to execute, in order to maximize the flushing penalty, all atomic instructions available to the platform should be tested to see how long each instruction takes to complete. Since the flushing is succeeded with the atomic operation itself, the longer the instruction executes, the worse the performance hit to the system becomes.

## 3. RELATED WORK

Malware in Android devices is generally separated into five categories as follows; Information Leakage, Privilege Escalation, Financial Charge, Ransomware, and Adware. Here, we introduce a new category, Quality of Service (QoS) attacks. QoS attacks aim to degrade and or disrupt the functionality of legitimate services. In this case, we aim to degrade the performance of other apps installed on the same mobile device using similar techniques as micro-architectural attacks.

Many applications being executed in the same shared hardware introduces a potential threat to the sandboxing techniques implemented by the OS. A malicious application might monitor hardware access patterns, to recover sensitive information from a potential victim. Micro-architectural side channel attacks have been studied over the last 20 years. The first practical attacks used the L1 cache as a covert channel to deduce cryptographic patterns and recover cryptographic keys [24, 10, 7]. Cache attacks usually establish a relationship between the positions accessed in the cache and the data used by the victim. However, L1 caches are core-private resources. Therefore these attacks are restricted to core co-resident processes. This scenario was deemed unrealistic. Consequently, micro-architectural attacks did not receive much attention after the first practical realizations. However, with the increasing popularity of shared hardware systems, i.e. cloud and mobile computing, the micro-architectural attack scenario of attacker and victim being able to run processes on the same hardware became realistic. Lately, interest in micro-architectural attacks started to rise again [30, 31]. With the attack scenario established, follow-up works overcame the issue of only targeting core-private resources: attacks targeting the Last Level Cache (LLC) have now widely been studied [29, 22, 19, 18]. Since the LLC is generally shared across cores multi-core processors it provides a suitable covert channel to run cross-core side-channel attacks. Further, the timing difference between LLC and memory accesses is higher, therefore, easier to distinguish. In addition to caches, other micro-architectural components have also been exploited to recover sensitive information. For instance, Branch Target Buffers (BTB) have also shown to provide a good covert channel to exploit non-constant code execution flows [9, 8]. In these attacks, authors monitor whether the outcome of a branch is miss-predicted, and use this information to deduce if the branch has been taken or not in a square and multiply operation. Moreover, architectural side-channels can be used to covertly communicate or signal presence of co-location as demonstrated in [17, 28, 16, 23].

Micro-architectural components have been widely exploited under non-virtualized and virtualized scenarios. However, little work has been done on exploiting embedded processors such as smartphones and tablets at the hardware level. Although time driven attacks have proven to be effective in ARM processors [26, 12], these attacks have not been demonstrated on mobile platforms i.e. devices running a mobile OS like Android or iOS. More recently, access-driven side-channel attacks have been exploited to detect ARM Trust-Zone code usage in the L1 cache [13], again not on a mobile platform. Finally in 2016, Lipp et al. [21] managed to run micro-architectural attacks such as Prime+Probe and Flush&Reload on various Android/ARM platforms, proving the practicality of these attacks on mobile devices. In the attack, authors exploited timing differences of accesses from cache and memory to recover sensitive information like keystrokes using a non-suspicious app. Also in 2016, Veen et al. [27] showed that the Rowhammer attack can be performed in Android platforms, without relying on software bugs or user permissions.

Other than cache attacks, there are also other OS based side-channel attacks targeting mobile platforms. These attacks take advantage of hardware related information provided by the OS to extract information. For instance, one can access Linux public directories to monitor the data consumption of each process to build a fingerprinting attack [32]. The network traffic is not the only feature that can be exploited; e.g. per process memory statistics are given by the OS can also be utilized to monitor what a victim application is doing [20] or even recover user's pictures [14].

As for countermeasures, very little work exists to eliminate side-channel attacks implemented in mobile devices. However, there are studies showing that detecting a malicious applications is possible. For instance, [11] utilizes static analysis to identify code that executes GUI attacks, whereas [15] focuses on preventing memory-related attacks (including memory bus side channel attacks) by using ARM-specific features.

# 4. THE QOS ATTACK

In order to perform the QoS attack on Android, we need to overcome two separate problems: 1) Detecting when the victim app is in active use i.e. in the foreground. 2) Performing an Exotic Atomic Operation and triggering a memory bus lock.

The first part of the attack is crucial to ensure that the user does not suspect the attacker app as the culprit behind the slowdown but rather blames it on the victim app. If the attacker was to trigger memory bus locking continuously, even from the background, it would appear as the system altogether has a performance problem. Or even worse, the user could trace the slowdown back to the attacker app and uninstall it, rather than the victim app and defeat the purpose of the attack. Therefore, the first problem the attacker has to overcome is detecting the victim app launch.

The second part of the attack is degrading the victim app's performance whenever it is active. The attacker can achieve this by performing Exotic Atomic Operations that trigger memory bus locking, resulting in significant performance overhead to the system. By triggering the lock while the victim app is running, the attacker can flush the ongoing memory operations in the CPU, disrupting the victim app's operation for over 10K cycles. By continuously doing this while the victim app is in the foreground, the attacker can lead the user to think that the victim app has suboptimal performance and consequently uninstall it.

**Attack Scenario:** Our attack consists of the following steps:

1. Launch the attacker app, create a `Sticky` background service meaning that the service will stay active even if the attacker app is closed or even shutdown by the user.

2. Run cache profiling tool to obtain spanning addresses to perform Exotic Atomic Operations.

3. Check for victim app launch from the background service. Wait until user puts the victim app in the foreground.

4. When the victim launch is detected, start the `Exotic Atomic Operation` loop, degrading the QoS of the targeted app.

5. Keep the loop running until the target app is no longer in the foreground. Stop the QoS degradation attack and release the system bottleneck as soon as the user quits the victim app.

6. Repeat until the user removes the victim app.

In the following, we describe the details of our attack as well as the design and implementation of our attacker app.

## 4.1 Detecting Victim Launch

In order to know when the targeted app is running, we use logical channels that are available to apps in Android OS. However, as Android OS evolved, some of these channels have been closed by the deprecated APIs. In the following, we show how to deduce the foreground app in different versions of Android OS, through various channels.

### *4.1.1 pre-Android 5.0 (API Level 21)*

In Android 5.0 (API 21), it is possible to get the list of running apps on the device as well as the foreground app. By using the runningAppProcess method from the `ActivityManager` class, an app can get the list as well as a binary value `LRU` that holds whether or not the app is the least recently used app. By continuously monitoring the LRU value of a process, an attacker detects when the victim app is in the foreground.

### *4.1.2 Android 5.0+ (API Level 21+)*

With Android 5.0 and forward, Android OS limited the access to other apps due to privacy and malware concerns. After the deprecation of the APIs to retrieve running apps, the background apps are now hidden to user level apps. Evidently, Play Store still has many applications like Task Managers, Memory Optimizers etc. that can detect apps running in the system. The question is, if the running apps are hidden, how do these Task Managers still retrieve the list of running. The answer lies in logical channel leakages. Here we discuss various methods that can be used to retrieve running apps and subsequently the foreground app.

**Package Usage Stats Permission:** With UsageStats class added in Android 5.0, apps can obtain various information such as the Last Time Used (LTU), package name and total time spent in the foreground about all the apps running on the device. Although there is no information provided about whether or not an app is in the foreground, it is still possible to infer this information using other data as follows. Using the `getLastTimeUsed` function, it is possible to check when an app, the victim in this case, was put in the foreground by the user. Note that this value is updated as soon as the user puts the app in the foreground. However, during the use of the app in the foreground, the LTU value remains constant. The value is again updated when the user changes activities and puts the victim app to the background. Using this information, one can monitor LTU value of an app as shown in Algorithm 1 and deduce the foreground activity of the victim app.

**Timing the killBackgroundProcesses Function:** After Android 5.0 (API Level 21), some of the Task Managers in the Play Store changed the way they worked to keep their functionality. Instead of getting the list of running apps, they now retrieve the list of all installed apps using the system provided `getInstalledApplication` function. Then

**Algorithm 1:** Victim app detection algorithm

```
functiin the foreground_Check();
while SwitchON// The start service switch do
    if inUse && !inUse_old then
        isActive = TRUE; // Victim app put in the
        foreground
    end
    if isActive then
        lock(); // Bus locking function
        inUse_old = inUse;
        inUse = Check_Usage();
        if inUse && !inUse_old then
            isActive = FALSE; // Victim app put off
            foreground
        end
    end
end
```

using the `killBackgroundProcesses`, go over the whole list and try to kill all listed apps using app package names. Note that this function requires the `KILL_BACKGROUND_PROCESSES` permission and kills only the apps that were running.

In our attack scenario, we can improve this method to detect whether the victim app is in the foreground. For the apps in the list that were not running, the function simply moves the next item. An attacker can periodically call this function, time each call, and kill the victim app. If the call of the function is above a certain threshold, the attacker can then deduce that the victim app was indeed running. Furthermore, if the victim app is in the foreground, the function cannot kill it at all. By detecting such unsuccessful kill requests, the attacker can deduce when the victim is in the foreground. Moreover, this method requires only the `KILL_BACKGROUND_PROCESSES` function that is a Normal Permission and does not require explicit user consent.

**Monitoring Hardware Resources:** Hardware resources in Android devices are shared hence accessible to all apps (given the permission). However, even with legitimate access rights, two apps cannot use certain resources simultaneously. For instance, the camera can only be accessed by a single app at any given time. So, when two apps try to access the camera API at the same time, only the first one is served while the second one receives a busy respond. This shared resource allows an attacker to monitor access to a hardware API and detect when a competitor app is in use.

**Getting the List of Running Services:** After Android 5.0, the list of running apps became hidden while the list of running services did not. Using this information, it is possible to check whether an installed application has any active service indicating a recent use. While a low-grain estimation, this information can still be used to detect whether an app has run recently. This method is especially useful in cases where victim app services launch after the app comes to foreground.

**Reading the System Logs:** In Android, many events including warnings, errors, crashes, and system-wide broadcasts are written to the system log. The operating system along with apps write to this log. Using this information, an attacker determines which apps have written to the log recently and monitor the victim app usage. Note that, since Android 4.1 (API level 16), system logs are accessible by only system/signature level apps meaning that third party apps cannot read system logs.

**Niceness of Apps:** Another way of detecting foreground app, is using `getRunningAppProcess` and to check niceness of apps. When the app is in the foreground, niceness value decreases to give the user a smoother experience. Therefore by constantly monitoring the niceness value of an app, one can defer foreground activity of a target app.

## 4.2 Cache Line Profiling Stage

Our attack is CPU-agnostic and employs memory bus locking regardless of the total cache size, cache line size or the number of cache sets. We achieve this by detecting uncacheable memory blocks with a quick, preliminary cache profiling stage. The profiling eliminates the need to know the CPU specifications e.g. the cache line size. Moreover, the Java code in Android apps is compiled to run on the JVM, resulting in changes to the cache addresses. By employing this profiling stage however, we can ignore address changes at the runtime.

In order to obtain a data block that spans multiple cache lines, we first allocate a block of page-aligned memory using `AtomicIntegerArray` from the `java.util.concurrent.atomic` class. Note that the size of this array should be large enough to contain multiple uncacheable addresses but not so large that it would trigger an Out-of-Memory (OOM) error and crash the app. In our experiments we have used *array_length* of 1024K to satisfy both conditions.

After the allocation, choose the ideal atomic operation to be used with the memory bus locking. While there are many atomic operations to choose from, it is most beneficial for the attacker to chose the operation that takes the longest time to perform. Since the memory bus lock remains active until the atomic operation is fully completed, longer operations result in stronger degradation to the system. In order to maximize the performance penalty, we have tested various operations such as `compareAndSet, decrementAndGet, addAndGet, AndDecrement` and `getAndIncrement`. Our results showed that the `getAndIncrement` operation was taking the longest time (10K-12K nanoseconds) hence was selected to be used in the attack. After choosing the atomic operation, we first operate continuously on a single address to get a baseline execution time. After the baseline execution time is established -without the bus locking- we start performing on the array as described in Algorithm 2. Starting from the beginning of the allocated array, we increment the array index by one in each loop and record the execution time. When a significantly longer execution time is detected, it is evident that the address is uncacheable or spanning multiple cache lines, therefore triggering the memory bus lock. After all the addresses are tested and the spanning ones recorded, we obtain a list of addresses that satisfy the `Exotic Atomic Operation` condition. We later use these addresses to lock the memory bus in our QoS degradation attack. Note that it is not necessary to obtain a long list since the attacker can de-allocate the array and reuse same addresses.

**Timing the Operations:** To time the performed atomic operations, we use the system provided `nanotime()` function. In theory, this function returns the JVM's high-resolution timer in nanoseconds. However in practice, due to numerous delays stemming from both hardware and software, we have observed varying timer resolutions. With the test devices that are used in experiments, best timer resolutions

**Algorithm 2:** Cache line detection algorithm

```
arr = Atomic Integer Array of length array_length
Output: List of cache line spanning addresses
for each $i$ smaller than $array\_length$ do
    startTime = System.nanoTime();
    value = arr.getAndIncrement(i);
    operation_time = System.nanoTime() - startTime;
    if $operation\_time > Pre\_calculated\_average$ then
        exotic_address[index++] = i;
    end
end
```

that we have observed were 958, 468, 104 and 104 nanoseconds on Galaxy S2, Nexus 5, Nexus 5X and Galaxy S7 Edge respectively. Considering that the aforementioned devices have CPUs running in the range of 1.2 to 2.26 GHz and assuming that the devices were running at highest possible CPU speeds, we can estimate the timer resolution in CPU cycles. By multiplying each CPU clock with the minimum timer resolution in nanoseconds, we get 223, 187, 1057 and 1149 CPU cycles of timer resolution for each device. While the low-resolution timer would present a problem for cache attacks, it is sufficient to distinguish between regular and Exotic Atomic Operations. Remember that we are measuring the execution time of atomic operations on different memory addresses to detect uncacheable addresses where this operation will incur a heavy timing penalty. In average, the regular atomic operations take around 1686, 1610, 844 and 369 nanoseconds in our test platforms. While Exotic Atomic Operations take around 3000-20000 nanoseconds as shown in Figure 1. Since the gap between the two is large enough, we can distinguish between the two using the `nanotime()` timer.

## 4.3 Attacker App Design and Implementation

We have designed a simple, lightweight proof-of-concept app to turn the performance penalty into an attack and tested it on various platforms and target apps. The app is designed to work on all devices that have Android 5.0 (API level 21) or a newer version of Android. According to [4] this covers 58.4% percent of all Android devices as of November 2016. Since the app a proof-of-concept, it uses the Package Usage Stats permission. This permission allows the app to get the list of running apps their last active times. Note that this permission opens a prompt and requires the user to explicitly give permission to the app. Our app has a simple interface that includes two activities. The first activity prompts the user to give the necessary permission after which the user can open the second activity. As seen in Figure 2, the app opens with an activity that shows disclaimer. On this activity screen Figure 4.3, the user has to give the necessary Package Usage Stats permission to the app or otherwise the app will not enter the app selection activity. To give the permission, the user only has to click on the "Give Ordinary Permissions" button and will automatically be forwarded to the necessary system settings page. After the permission is obtained, the user can return to the opening activity and click on the Go To App Selection Screen. On the app selection activity, as shown in Figure 2(b), the user can select any of the apps that were used in the last 24 hours as the target. After that, the user clicks on the "Start Slowdown Service" switch and the selected app name is passed to the background service. Now, the background service of the attacker app continuously monitors the list of used apps. When the selected app is detected to be put in the foreground, the service starts the attack and degrades the target's performance. The QoS attack continues until the user exits the selected app.

# 5. EXPERIMENT SETUP AND RESULTS

In this section, we give the details of our experiment setup, the devices that we have tested our attack on and finally present the performance degradations observed by the selected benchmarks apps.

## 5.1 Experiment Setup

In order to test the level of QoS degradation that our attack can cause, we performed experiments on various smartphones. Also, since regular Android apps generally do not provide performance statistics, we have used benchmarks to quantify the QoS degradation. In our experiments, we have collected performance measurements with and without the attacker app running in the background.

As test platforms, we have used four different mobile devices namely Galaxy S2, Nexus 5, Nexus 5X and Galaxy S7 Edge. We have selected these devices to show the viability of the attack on different mobile CPUs. Also to add variety, we have updated these devices to different versions of Android. We have updated Galaxy S2, Nexus 5 and Nexus 5X and Galaxy S7 Edge to Android 4.1.2 (API Level 16), 5.1.1 (API Level 22), 6.0.1 (API Level 23), 6.0.1 (API Level 23) respectively. As expected, we were able to perform our attack on all test devices, regardless of what version of Android they had running. Note that we have not used any unofficial Android distribution, ROM or side-loaded any patches. All the test devices use their stock Android ROMs without any modification. We present the detailed specifications of the devices in Table 1.

Since all Android devices employ a type of battery optimization, we wanted to make sure that our experiments would not be affected by this in any way. Therefore, to ensure optimal performance, all the experiments were performed while the test devices were fully charged and connected to a power outlet. Furthermore, Android monitors the data from the temperature sensors and adjusts the system performance to prevent overheating. To prevent any slowdown due to high temperature, we have placed the test devices apart from each other made sure that they are properly ventilated.

## 5.2 Test App: Benchmarks

In order to quantify the level of degradation caused by the attack, we have used benchmarks apps, performing various tests. With these tests, we have measured the performance of the devices performing high-level operations such as 3D processing, 2D image processing, streaming as well as low-level tests like ALU computations, memory read and write access time, bandwidth and latency.

To prevent any bias and provide a fair comparison, we have used the top downloaded benchmarks in the Google Play store. In total, we have used 11 benchmark apps. Other than the CPU Prime Benchmark, all of these benchmarks has numerous available tests to score different aspects of the device. For instance, the AnTuTu Benchmark performs 3D computation, CPU, RAM and UX (User Experience)

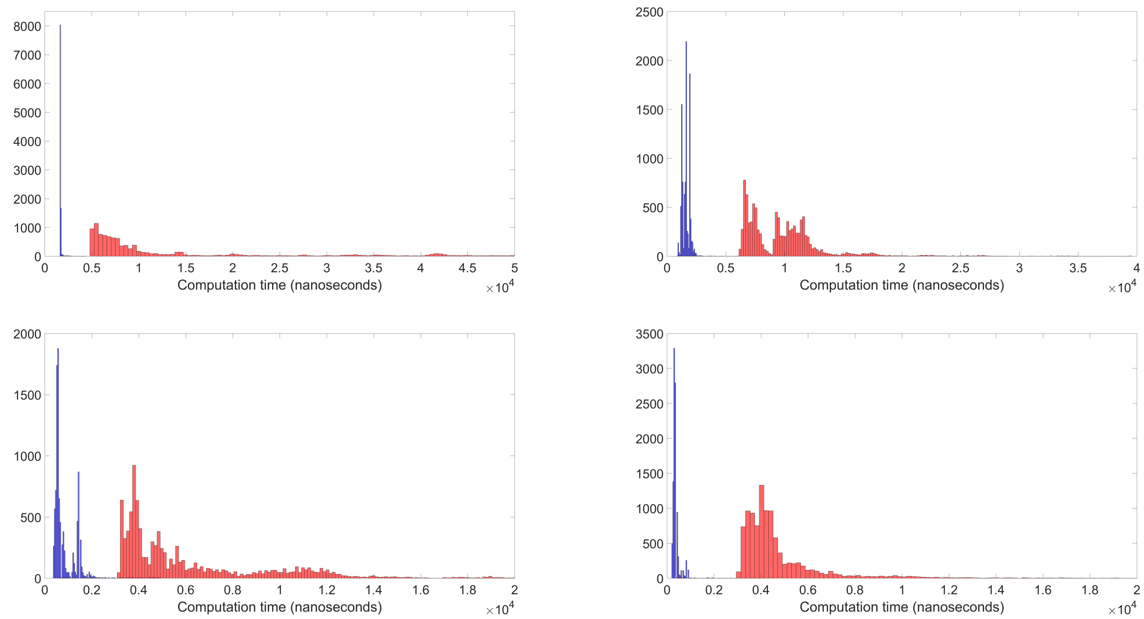
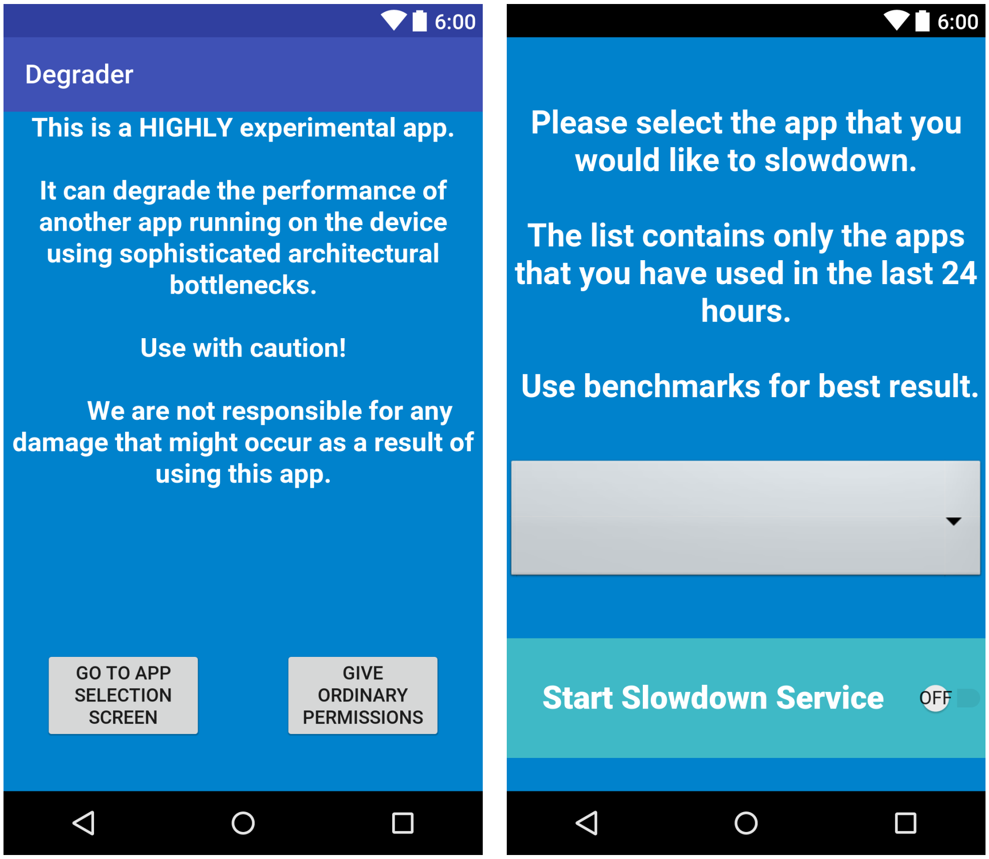


**Figure 1: Histograms of regular and Exotic Atomic Operation of the test devices, Galaxy S2, Nexus 5, Nexus 5x and Galaxy S7 Edge respectively. X-axis represents the computation time while the Y-axis shows the number of samples obtained at each point. Also, blue and red bars represent regular and exotic atomic operations respectively.**



This is a HIGHLY experimental app.

It can degrade the performance of another app running on the device using sophisticated architectural bottlenecks.

Use with caution!

We are not responsible for any damage that might occur as a result of using this app.

Please select the app that you would like to slowdown.

The list contains only the apps that you have used in the last 24 hours.

Use benchmarks for best result.

**Figure 2: Attacker app interface**

**Table 1: Specifications of the test devices used in experiments**

| | Galaxy S2 | Nexus 5 | Nexus 5X | Galaxy S7 Edge |
|---|---|---|---|---|
| Android Version | 4.1.2 | 5.1.1 | 6.0.1 | 6.0.1 |
| API Level | 16 | 22 | 23 | 23 |
| SoC | Exynos 4 | Snapdragon 800 | Snapdragon 808 | Snapdragon 820 |
| ARM Core | Cortex-A9 | Krait 400 | 4xCortex-A53 & 2xCortex-A57 | 4x Kryo |
| Number of Cores | 2 | 4 | 4+2 | 2+2 |
| CPU Clock (GHz) | 1.2 | 2.26 | 1.4 & 1.8 | 1.6 & 2.15 |
| Big-Little? | no | no | yes | yes* |
| 32/64-bit? | 32 | 32 | 64 | 64 |
| ARM version | v7-A | v7 | v8-A | v8-A |

tests to measure different aspects of the device performance, providing a separate score for each category. Including these tests, we have monitored the QoS strength of our app with 45 different tasks. Note that while some of the benchmark apps like 3DMark, GeekBench, GFXBench, and PCMark were not available for the older Galaxy S2 running Android 4.1.2, rest was available to all of the tested devices.

Complete list of benchmarks that we have used is as follows; 3DMark (Ice Storm Unlimited, Slingshot Unlimited), AnTuTu Benchmark (3D, CPU, RAM, UX), Benchmark&Tuning, CF-Bench (Java, Native, Overall), CPU Prime Benchmark, Geekbench 4, GFXBench GL (ALU 2, Driver 2, Manhattan 3.0, Texturing, T-Rex), PassMark Performance Test Mobile (2D Graphics, 3D Graphics, CPU, Disk, RAM, System), PCMark (Storage, Word, Work 2.0), Quadrant Standard (CPU, I/O, Memory, Overall), Vellamo (Browser - Chrome, Metal, Multi-Core).

## 5.3 Degradation Results

As mentioned earlier, regular Android apps generally do not output performance statistics. While the slowdown result of the attack and other effects are visible to the human eye, the visual slowdown is not quantifiable. To overcome this problem, we have decided to use benchmarks that can output system performance at any given time. To demonstrate and quantify the performance degradation caused by our QoS attack on different functions of the system and apps, we have used numerous benchmarks. This allowed us to measure system performance both with (degraded performance) and without (baseline performance) the attack running in the background, giving us a clear contrast for each test device. Here, we present these results.

Our results show that, we can significantly degrade the QoS of various apps. As shown in Figure 3, test benchmarks show varying levels of performance degradation, up to 90.98% compared to the baseline results. Also, it is evident that different test devices show diverse levels of degradation due to the difference in the underlying hardware. For instance, Nexus 5 has average degradation about 20% while Nexus 5X has about 45%.

Figure 4 shows how much performance degradation was observed by each benchmark when targeted by the attack. It is evident that almost all benchmarks show strong degradation in performance. Also, as mentioned before, one of our test devices, the Galaxy S2 with Android 4.1.2 did not support all of the benchmarks. These unavailable tests are shown with 0% degradation in the figure. In Figure 5, we represent normalized benchmark results i.e. ratio of degraded performance to baseline results. Note that like Figure 4, benchmarks are numbered as in Table 2.

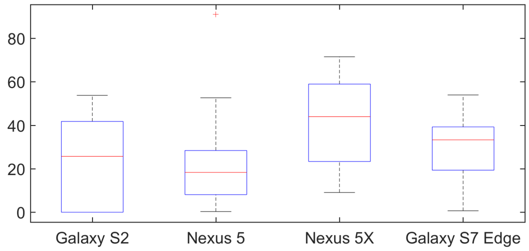


**Figure 3: Quartile representation of benchmark performance degradations. Vertical axis represents performance degradation percentage compared to the baseline and red lines mark the average degradation.**

Names of the benchmark suites and the specific tests are given in Table 2. Note that, while many of these benchmarks have subtest, they are not represented for clarity. Instead of giving results for each subtest, we only represent the average degradation. Finally, benchmark scores vary greatly depending on the computation power of the device as expected. Therefore, we represent degradation percentages rather than actual scores.

## 5.4 Stealthiness of the Attacker App

Our attack is hard to detect and runs with a minimal footprint on the system. During our experiments, we have continuously monitored the CPU usage of our app through Android Monitor provided by the `Android Studio`. We have observed that our app has low CPU usage, even at the times of performing Exotic Atomic Operations continuously. For all the tested devices, CPU usage of the attacker app never exceeded 10% mark, showing light CPU usage.

To show that our app is stealthy and can pass modern malware scanners, we have used 23 of the most popular malware scanners on the Google Play Store. At the time of testing, none of the malware scanners (see Appendix A) were able to detect our attacker app as malware even though it was causing significant distress to the operation of the device. We believe that this is due to following reasons:

1. Unlike other micro-architectural attacks e.g. Rowhammer and cache attacks (Flush&Reload, Evict&Reload, Prime+Probe etc.), our QoS attack does not require evicting memory blocks from the cache. Since it does not re-

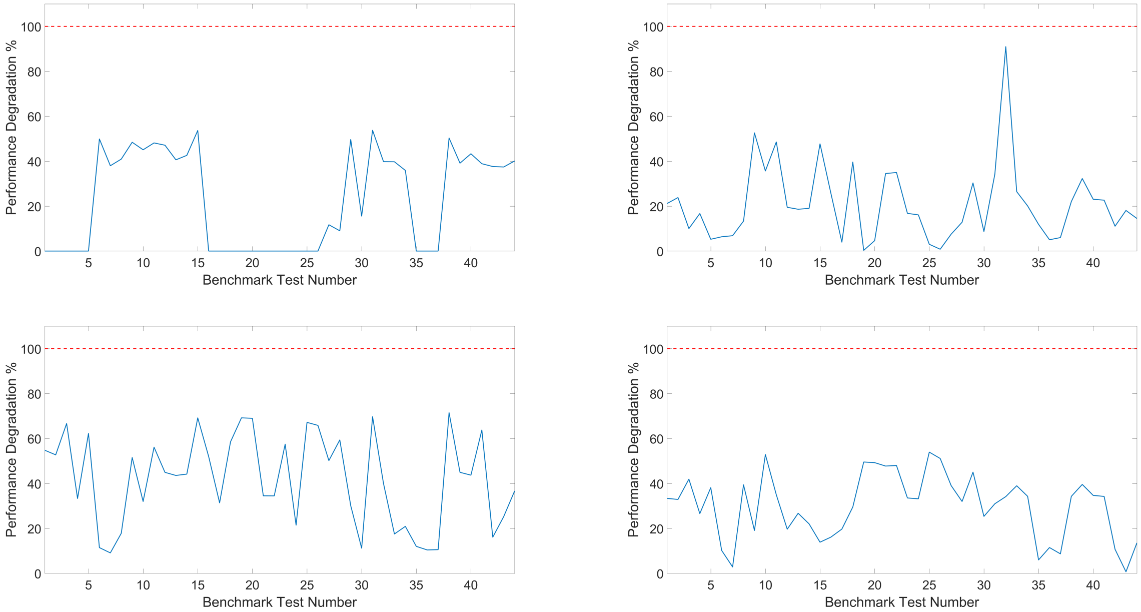


**Figure 4: Performance degradation percentages of benchmarks on test devices, Galaxy S2, Nexus 5, Nexus 5x and Galaxy S7 Edge respectively. Red dashed lines represent the maximum possible degradation, i.e. 100%. Benchmarks are numbered as in Table 2.**

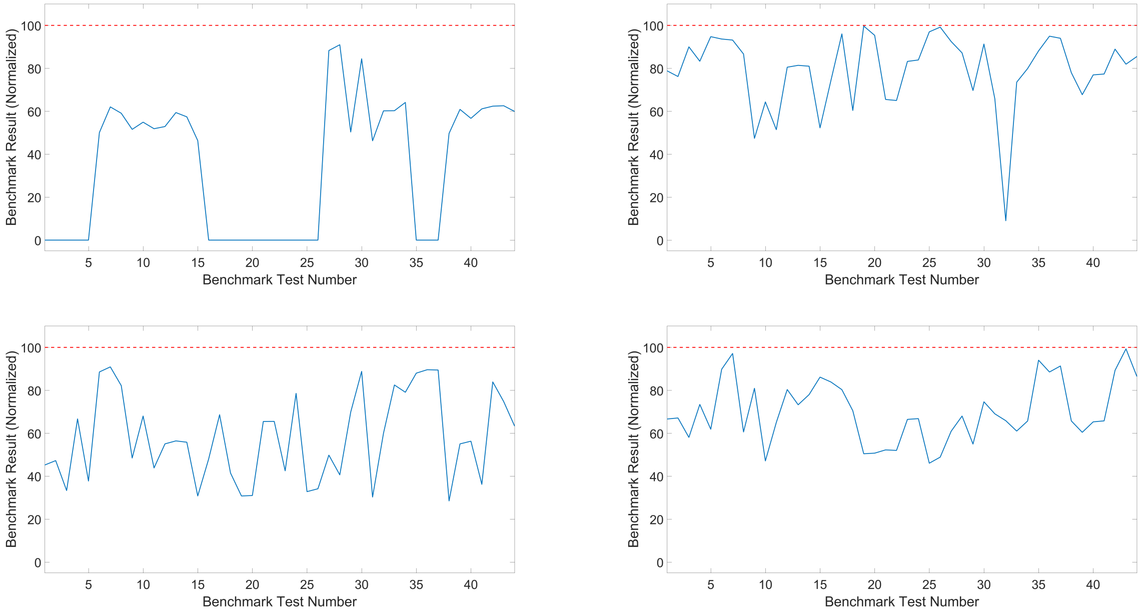


**Figure 5: Normalized benchmark results of test devices, Galaxy S2, Nexus 5, Nexus 5x and Galaxy S7 Edge respectively. Red dashed lines represent baseline performance for each benchmark while blue lines represent results under attack. Benchmarks are numbered as in Table 2.**

quire eviction or continuous monitoring of a specific memory block, there is no continuous data access.

2. The memory bus trigger does not have to run at a high frequency to achieve performance degradation. In fact, a single bus lock results in a performance bottleneck for over 12395, 9766, 6128 and 4693 nanoseconds in average on our test devices, as shown in Figure 1. So, as long as the attacker can trigger the bus lock about every 10K+ cycles, the system will stay in a continuous state of a bottleneck. The fact that the attacker issues only 1 CPU instruction every 10K+ cycles, keeps the CPU load minimal. If a user was to use a task manager to inspect CPU usages of different apps or check the system logs to do the same, he/she would only see CPU use of unsuspecting 10% by the attacker app.

**Table 2: Performance degradation of the tested benchmarks on test platforms.**

| | Galaxy S2 | Nexus 5 | Nexus 5X | Galaxy S7 Edge |
|---|---|---|---|---|
| **Benchmark** | % | % | % | % |
| **1. 3DMark (Ice Storm Unlimited Graphics Average)** | NA | 21.15 | 54.78 | 33.36 |
| **2. 3DMark (Ice Storm Unlimited Physics Average)** | NA | 23.82 | 52.73 | 32.87 |
| **3. 3DMark (Slingshot Unlimited Graphics Average)** | NA | 10.00 | 66.67 | 41.91 |
| **4. 3DMark (Slingshot Unlimited Physics Average)** | NA | 16.67 | 33.33 | 26.57 |
| **5. AnTuTu Benchmark (3D)** | NA | 5.25 | 62.25 | 38.12 |
| **6. AnTuTu Benchmark (CPU)** | 49.90 | 6.36 | 11.44 | 10.18 |
| **7. AnTuTu Benchmark (RAM)** | 37.97 | 6.83 | 9.08 | 2.84 |
| **8. AnTuTu Benchmark (UX)** | 40.93 | 13.34 | 17.81 | 39.40 |
| **9. Benchmark & Tuning (CPU)** | 48.43 | 52.61 | 51.53 | 19.06 |
| **10. Benchmark & Tuning (I/O)** | 45.08 | 35.65 | 31.97 | 52.87 |
| **11. Benchmark & Tuning (Memory)** | 48.13 | 48.56 | 56.14 | 34.96 |
| **12. CF-Bench (Java)** | 47.11 | 19.48 | 44.94 | 19.63 |
| **13. CF-Bench (Native)** | 40.61 | 18.61 | 43.55 | 26.73 |
| **14. CF-Bench (Overall)** | 42.59 | 19.01 | 44.16 | 22.00 |
| **15. CPU Prime Benchmark** | 53.66 | 47.73 | 69.19 | 13.85 |
| **16. Geekbench 4 (Compute Overall)** | NA | 25.61 | 51.96 | 16.12 |
| **17. Geekbench 4 (Multi-C Overall)** | NA | 3.96 | 31.36 | 19.69 |
| **18. Geekbench 4 (Single-C Overall)** | NA | 39.62 | 58.52 | 29.50 |
| **19. GFXBench GL (ALU 2 (Frames))** | NA | 0.31 | 69.20 | 49.54 |
| **20. GFXBench GL (ALU 2 Offscreen (Frames))** | NA | 4.63 | 68.99 | 49.26 |
| **21. GFXBench GL (Driver 2 Overhead (Frames))** | NA | 34.50 | 34.50 | 47.74 |
| **22. GFXBench GL (Driver 2 Overhead Offscreen (Frames))** | NA | 34.98 | 34.47 | 47.98 |
| **23. GFXBench GL (Manhattan 3.0 (Frames))** | NA | 16.78 | 57.48 | 33.53 |
| **24. GFXBench GL (Manhattan 3.0 Offscreen (Frames))** | NA | 16.11 | 21.43 | 33.17 |
| **25. GFXBench GL (Texturing (MTexels/s))** | NA | 3.01 | 67.18 | 53.94 |
| **26. GFXBench GL (Texturing Offscreen (MTexels/s))** | NA | 0.81 | 65.84 | 51.11 |
| **27. GFXBench GL (T-Rex Offscreen)** | 11.71 | 7.50 | 50.17 | 39.06 |
| **28. GFXBench GL (T-Rex)** | 9.02 | 12.82 | 59.37 | 31.98 |
| **29. PassMark Performance Test Mobile (2D Graphics)** | 49.66 | 30.33 | 30.12 | 45.03 |
| **30. PassMark Performance Test Mobile (3D Graphics)** | 15.55 | 8.69 | 11.18 | 25.34 |
| **31. PassMark Performance Test Mobile (CPU)** | 53.75 | 34.23 | 69.71 | 30.95 |
| **32. PassMark Performance Test Mobile (Disk)** | 39.80 | 90.98 | 39.98 | 34.14 |
| **33. PassMark Performance Test Mobile (RAM)** | 39.74 | 26.45 | 17.50 | 38.98 |
| **34. PassMark Performance Test Mobile (System)** | 35.89 | 20.09 | 20.89 | 34.28 |
| **35. PCMark (Storage)** | NA | 11.89 | 12.00 | 5.96 |
| **36. PCMark (Work 2.0)** | NA | 5.03 | 10.40 | 11.47 |
| **37. PCMark (Work)** | NA | 6.00 | 10.54 | 8.66 |
| **38. Quadrant Standard (CPU)** | 50.31 | 22.03 | 71.50 | 34.26 |
| **39. Quadrant Standard (I/O)** | 39.12 | 32.28 | 44.94 | 39.56 |
| **40. Quadrant Standard (Memory)** | 43.30 | 23.06 | 43.69 | 34.68 |
| **41. Quadrant Standard (Overall)** | 38.88 | 22.65 | 63.78 | 34.23 |
| **42. Vellamo (Browser - Chrome)** | 37.62 | 11.06 | 16.08 | 10.70 |
| **43. Vellamo (Metal)** | 37.42 | 18.07 | 25.24 | 0.68 |
| **44. Vellamo (Multi-Core)** | 40.08 | 14.50 | 36.66 | 13.54 |

In addition to that, remember that the memory bus locking part of our attack is active only when the target app is in the foreground. So, unless the target device supports split screen and multitasking, and both the task manager and the victim app support split screen, the user cannot even observe this 10% percent load. While the bus locking is not active, the load of our background service is nominal, between 0.01% and 1%.

3. The current malware and anti-virus scanners are not adequately suited to detect micro-architectural attacks or bottlenecks. In these types of attacks, the attack surface and threat models are completely different than those of classical malware.

4. The attack is hard to detect with dynamic or static analysis techniques because the attacker app performs legitimate data operations and does not attempt to access any unauthorized APIs or data. Any app can perform atomic operations and trigger the bus lock without an ill-intent. Therefore it is not practical to simply detect these instructions. Further, the attacker app does not contain any information like the package name of the victim app which could give away the attack during a static analysis. As for dynamic analysis, the scanner would have to trigger a bus

lock which would require running the victim app which may not be possible if the victim is later provided to the app via the web connection. Finally, since no compute intensive operation like increment counters are used in the attack, it would be to very hard to observe the degradation during the scan and tag it as malware.

## 6. DETECTION AND MITIGATION

In order to detect and mitigate this attack, we propose the following countermeasures.

**Atomic Operation Address Check:** The operating system can inspect atomic operation operands for uncacheable memory addresses. When such an address is detected, the OS then can increment the count of recent atomic operations on uncacheable addresses. If the count reaches a preselected threshold, then the OS can give the app issuing these atomic operations a timeout or shut it down altogether. Though this countermeasure would incur a performance overhead, the penalty would be smaller than bus locking. Alternatively, the OS can move the operated data to a cacheable address and prevent bus locks.

**Memory Bandwidth Monitoring:** The operating system can periodically monitor memory bandwidth and stop or slow down any process that triggers bus locking frequently. In the case of an active attack, this method would allow the OS to detect the culprit and stop the process/app while allowing non-attack locks to be performed as usual.

**Closing Inter-app Logical Channel Leaks:** As discussed in Section 4.1 in detail, there are numerous ways to know which apps are installed or running on an Android device. When this information is obtained by the attacker, it is a matter of resource monitoring (through logical or side-channels) to detect victim launch. So, if an attacker cannot obtain any information about installed or running apps on the device, then the detection stage would rely only on more noisy side-channels like shared hardware monitoring. While this countermeasure would strongly impact existing apps using information like installed apps, it would also make the detection part of the attack more difficult.

**Restricting Access to UsageStats:** UsageStats permission is used by numerous apps that need to monitor user's app usage. For instance, a mobile data operator might want to check which apps is currently active so that they can disable the data usage counter for a specific app e.g. Youtube or Netflix. However, it is also a dangerous permission in the sense that it looks innocent to a naive user but has crucial consequences as demonstrated in this paper. So, instead of removing this permission altogether, making it a signature/system level permission would protect this critical information while allowing trusted parties e.g. data operators to keep the functionality.

## 7. CONCLUSION

In conclusion, we show that low-level architectural attacks are a real threat in mobile devices. Battery, compute power and storage restrictions of mobile devices require strong optimizations and create very suitable candidates for these types of attacks. Further, combining architectural attacks with logical channel leakages open a wide range of exploits for malicious parties. In this work, we have shown that malicious parties can exploit the underlying shared hardware to degrade or even halt operations of other apps using this combination and inter-app interactions have to be controlled carefully.

## 8. ETHICAL CONCERNS AND RESPONSIBLE DISCLOSURE

The attacker app that we have designed is uploaded to the Google Play Store and since it is not detected as malware by the Play Store, it is currently available for download. Since the app causes performance degradation to the overall system, we have clearly warned users in the install page that this is an experimental app aiming to degrade the performance of the overall system and should be used with caution. We hope that this warning will be sufficient to prevent any accidental installations that might result in unwanted performance degradation. Further, to prevent hogging of the system resources, we have significantly limited the performance degradation power of the app by decreasing the frequency of memory bus lock triggers.

Finally, we have informed the Android security team of our findings in advance to this publication and made our bug submission through the AOSP bug report portal. Since then, we have been in contact with the Android security team and currently working on possible detection and mitigation strategies for the attack.

## 9. ACKNOWLEDGMENTS

This work is supported by the National Science Foundation, under the grant CNS-1618837. Also, we would also like to thank anonymous reviewers for their insightful comments and suggestions for improving this paper. In particular we thank our shepherd, Mohammad Mannan.

## 10. REFERENCES

[1] Android system permissions. https://developer.android.com/guide/topics/security/permissions.html.

[2] ARM Synchronization Primitives. http://infocenter.arm.com/help/topic/com.arm.doc.dht0008a/DHT0008A_arm_synchronization_primitives.pdf.

[3] Number of available applications in the Google Play Store from December 2009 to September 2016. https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/.

[4] Android Dashboards. Online, Nov 2016. https://developer.android.com/about/dashboards/index.html.

[5] Android, the world's most popular mobile platform. Online, Nov 2016. https://developer.android.com/about/android.html.

[6] Application Fundamentals. Online, Nov 8 2016. https://developer.android.com/guide/components/fundamentals.html.

[7] Aciiçmez, O. Yet Another MicroArchitectural Attack: Exploiting I-Cache. In *Proceedings of the 2007 ACM Workshop on Computer Security Architecture*.

[8] Aciiçmez, O., Gueron, S., and Seifert, J.-P. New branch prediction vulnerabilities in OpenSSL and necessary software countermeasures. In *IMA International Conference on Cryptography and Coding* (2007).

[9] Aciiçmez, O., K. Koç, c., and Seifert, J.-P. Predicting secret keys via branch prediction. In *Topics in Cryptology CT-RSA 2007*, vol. 4377. pp. 225–242.

[10] Bernstein, D. J. Cache-timing attacks on AES, 2004. URL: `http://cr.yp.to/papers.html#cachetiming`.

[11] Bianchi, A., Corbetta, J., Invernizzi, L., Fratantonio, Y., Kruegel, C., and Vigna, G. What the App is That? Deception and Countermeasures in the Android User Interface. In *IEEE S&P 2015*.

[12] Bogdanov, A., Eisenbarth, T., Paar, C., and Wienecke, M. Differential cache-collision timing attacks on aes with applications to embedded cpus. In *CT-RSA 2010*.

[13] Brumley, B. Cache storage attacks. In *CT-RSA 2015*.

[14] Chen, Q. A. and Qian, Z. and Mao, Z. M. Peeking into Your App without Actually Seeing It: UI State Inference and Novel Android Attacks. In *USENIX Security 2014*.

[15] Colp, P., Zhang, J., Gleeson, J., Suneja, S., de Lara, E., Raj, H., Saroiu, S., and Wolman, A. Protecting data on smartphones and tablets from memory attacks. ASPLOS 2015.

[16] Inci, M. S., Gulmezoglu, B., Eisenbarth, T., and Sunar, B. Co-location Detection on the Cloud. In *COSADE 2016*.

[17] Inci, M. S., Irazoqui, G., Eisenbarth, T., and Sunar, B. Efficient, Adversarial Neighbor Discovery using Logical Channels on Microsoft Azure. In *ACSAC 2016*.

[18] Irazoqui, G., Eisenbarth, T., and Sunar, B. S$A: A Shared Cache Attack that Works Across Cores and Defies VM Sandboxing and its Application to AES. In *IEEE S&P 2015*.

[19] Irazoqui, G., Inci, M. S., Eisenbarth, T., and Sunar, B. Wait a Minute! A fast, Cross-VM Attack on AES. In *RAID 2014*.

[20] Jana, S., and Shmatikov, V. Memento: Learning secrets from process footprints. In *IEEE S&P 2012*.

[21] Lipp, M., Gruss, D., Spreitzer, R., Maurice, C., and Mangard, S. ARMageddon: Last-Level Cache Attacks on Mobile Devices. In *USENIX Security 2016*.

[22] Liu, F. and Yarom, Y. and Ge, Q. and Heiser, G. and Lee, R. B. Last-level cache side-channel attacks are practical. In *IEEE S&P 2015*.

[23] Maurice, C. and Weber, M. and Schwarz, M. and Giner, L. and Gruss, D. and Carlo, A. B. and Mangard, S. and Römer, K. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In *NDSS 2017*.

[24] Osvik, D. A., Shamir, A., and Tromer, E. Cache Attacks and Countermeasures: The Case of AES. CT-RSA 2006.

[25] Rusling, D. A. ARMv7a Architecture Overview. presentation, 2010. https://wiki.ubuntu.com/Specs/M/ARMGeneralArchitectureOverview?action=AttachFile&do=get&target=ARMv7+Overview+a02.pdf.

[26] Spreitzer, R., and Plos, T. On the Applicability of Time-Driven Cache Attacks on Mobile Devices. In *NSS 2013*.

[27] van der Veen, V., Fratantonio, Y., Lindorfer, M., Gruss, D., Maurice, C., Vigna, G., Bos, H., Razavi, K., and Giuffrida, C. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In *CCS 2016*.

[28] Varadarajan, V., Zhang, Y., Ristenpart, T., and Swift, M. A placement vulnerability study in multi-tenant public clouds. In *USENIX Security 2015*.

[29] Yarom, Y., and Falkner, K. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security 2014*.

[30] Zhang, Y., Juels, A., Oprea, A., and Reiter, M. K. HomeAlone: Co-residency Detection in the Cloud via Side-Channel Analysis. In *IEEE S&P 2011*.

[31] Zhang, Y., Juels, A., Reiter, M. K., and Ristenpart, T. Cross-VM Side Channels and Their Use to Extract Private Keys. In *CCS 2012*.

[32] Zhou, X., Demetriou, S., He, D., Naveed, M., Pan, X., Wang, X., Gunter, C. A., and Nahrstedt, K. Identity, location, disease and more: Inferring your secrets from android public resources. In *CCS 2013*.

# APPENDIX

## A. COMPLETE LIST OF MALWARE SCANNERS

1. 360 Security - Antivirus Boost
2. Anti Spy (SpyWare Removal)
3. AntiVirus FREE 2016 - Android
4. Avira Antivirus Security
5. Bitdefender Antivirus Free
6. Clean Master (Boost & AppLock)
7. CM Security AppLock AntiVirus
8. Comodo Mobile Security
9. Dr. Safety - BEST FREE ANTIVIRUS
10. Droidkeeper - Free Antivirus
11. FREE Spyware & Malware Remover
12. GO Security
13. Antivirus AppLock
14. Hidden Device Admin Detector
15. Kaspersky Antivirus & Security
16. Lookout Security & Antivirus
17. Malwarebytes Anti-Malware
18. McAfee SpyLocker Remover
19. Mobile Security & Antivirus
20. Norton Security and Antivirus
21. Open Vaccine
22. Security & Power Booster -Free
23. Stubborn Trojan Killer