Eavesdropping User Credentials via GPU Side Channels on Smartphones

Boyuan Yang
University of Pittsburgh
USA
by.yang@pitt.edu

Ruirong Chen
University of Pittsburgh
USA
ruirongchen@pitt.edu

Kai Huang
University of Pittsburgh
USA
k.huang@pitt.edu

Jun Yang
University of Pittsburgh
USA
juy9@pitt.edu

Wei Gao
University of Pittsburgh
USA
weigao@pitt.edu

ABSTRACT

Graphics Processing Unit (GPU) on smartphones is an effective target for hardware attacks. In this paper, we present a new side channel attack on mobile GPUs of Android smartphones, allowing an unprivileged attacker to eavesdrop the user's credentials, such as login usernames and passwords, from their inputs through on-screen keyboard. Our attack targets on Qualcomm Adreno GPUs and investigate the amount of GPU overdraw when rendering the popups of user's key presses of inputs. Such GPU overdraw caused by each key press corresponds to unique variations of selected GPU performance counters, from which these key presses can be accurately inferred. Experiment results from practical use on multiple models of Android smartphones show that our attack can correctly infer more than 80% of user's credential inputs, but incur negligible amounts of computing overhead and network traffic on the victim device. To counter this attack, this paper suggests mitigations of access control on GPU performance counters, or applying obfuscations on the values of GPU performance counters.

CCS CONCEPTS

• **Security and privacy → Systems security**; • **Human-centered computing → Ubiquitous and mobile computing**.

KEYWORDS

Mobile GPU, Side Channel, Input Eavesdropping, Smartphones, Performance Counters.

ACM Reference Format:
Boyuan Yang, Ruirong Chen, Kai Huang, Jun Yang, and Wei Gao. 2022. Eavesdropping User Credentials via GPU Side Channels on Smartphones. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22), February 28 – March 4, 2022, Lausanne, Switzerland.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3503222.3507757

ASPLOS '22, February 28 – March 4, 2022, Lausanne, Switzerland

ACM ISBN 978-1-4503-9205-1/22/02…$15.00
https://doi.org/10.1145/3503222.3507757

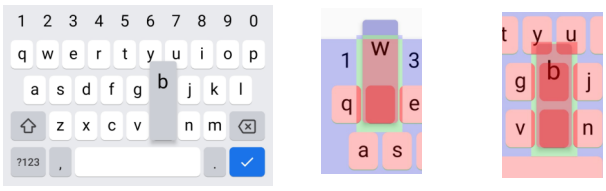
1 INTRODUCTION

Malicious attacks against smartphones have recently become a major technical concern [3, 41]. These attacks target on both system software and hardware. Software attacks exploit vulnerabilities in mobile OS and applications to steal users' text chats [9, 38], video calls [36], or gain device control [6, 19, 22], but can be mitigated through software updates. Hardware attacks exploit unintended information leakage from system hardware, and are difficult to eliminate as hardware upgrade cannot be easily done on commodity devices.

Existing hardware eavesdropping attacks on smartphones mainly focus on CPU and on-board sensors. For example, access time on CPU cache provides information about victim applications when they contend for cache accesses [13, 15, 53], and IMU sensor readings could be used to infer users' keystrokes [1, 2, 26, 33, 35, 39]. However, the correlations between these hardware data and user activities are usually weak and ambiguous, and are susceptible to various system factors and random noise in practice [25]. These attacks are hence limited to rough estimation of user activities such as the applications being launched [15, 49, 50], user locations [17, 34] and user identities [5, 7, 57].

Alternatively, GPU has been considered as another effective target for eavesdropping attacks. Existing GPU attacks mainly monitor the variation of GPU workloads that can be measured by either the GPU cache access time [37] or GUI performance metrics [32, 37], and infer user activities from such variations. The strengths of these attacks, however, are determined by the amount of GPU workload variation caused by user activities, and they are incapable of inferring user's keyboard inputs that result in only a negligible amount of GPU workload.

Instead, in this paper, we present a new eavesdropping attack on mobile GPUs that allows an unprivileged attacker to precisely infer the user's credential inputs through the on-screen keyboard. Our basic rationale is the explicit correlation between user inputs and screen display on smartphones: on one hand, user inputs are always reflected into screen display as visible feedback; on the other hand, display contents are always rendered by GPU, and GPU is solely used for graphics rendering in most cases. Based on this rationale, we found that *GPU performance counters (PCs)* in certain categories reflect the amount of screen display changes at the granularity of individual pixels. This explicit and fine correlation allows direct eavesdropping without any ambiguity.



(a) Popup with a key press (b) GPU overdraw

Figure 1: GPU overdraw on popups of key presses in Android. Blue: 1x overdraw; green: 2x overdraw; pink: 3x overdraw; red: 4x overdraw.

More specifically, our attack utilizes the popups that come by default with every key press[1]. Since a popup is rendered on top of the keyboard, popups with different key presses, as shown in Figure 1, result in different amounts of *GPU overdraw*, which is the number of draws on the same pixel when drawing keyboard UI in multiple layers. Such amounts of GPU overdraw is reflected by certain categories of GPU PCs, such as Low Resolution Z (LRZ) pass, Rasterization (RAS) and Vertex Cache (VPC) on Qualcomm Adreno GPUs. On the victim device, by reading these PCs that have unique variations for each key press, we can infer the key press without random guess.

The major challenge of our attack is how to access GPU PCs on Android. Unlike desktop systems with well-developed libraries (e.g., CUDA for Nvidia GPUs), most Android systems do not provide explicit interfaces for GPU status queries. Further, Android enforces many security protections, including permission requirements for system resource access and resource isolation by SELinux, making it harder for an unprivileged attacker to retrieve the global GPU information. Our approach to this challenge is to read the raw values of GPU PCs from the GPU device file. In Android, this device file is used as the interface to access GPU hardware by user-space drivers (e.g., OpenGL ES and Vulkan), which run as shared libraries with the same PID as the calling user application. Hence, it is accessible to unprivileged user programs. In particular, we target Qualcomm Adreno GPUs[2] and use their GPU drivers' open-source header file (`msm_kgsl.h`) as the reference to choose the right parameters of the `ioctl()` system call to access the GPU device file.

Besides, the eavesdropping accuracy could also be affected by many system and user factors. For example, one key press may result in irregular changes of GPU PC values, cursor blinking and OS notifications may also produce unexpected display changes that affect GPU performance counters. The users may also correct their past inputs or switch to other applications during inputs, causing confusions in eavesdropping. To address the system factors, we build classification models using the GPU PC data being collected offline, and further use these models online to distinguish between GPU hardware events caused by key presses and other system factors. To address the user factors, we identify specific GPU PCs that show strong features indicating input corrections and application switch, so as to exclude these factors from eavesdropping.

[1]These popups are used as visible feedback to user inputs, and help the user verify that the correct key is being pressed.

[2]Qualcomm Adreno GPUs are the most popular mobile GPUs, and are used on over 40% of mobile devices worldwide in 2020 [54].

To our best knowledge, our work is the first that allows eavesdropping of user credentials via on-screen keyboard inputs on smartphones. Our detailed contributions include:

- We identified the GPU PCs as reliable hardware indicators of user's key presses, and quantified the correlation between GPU PC values and different key presses.
- We allowed an unprivileged attacker to read the global values of selected GPU PCs on the victim device.
- We quantified the impacts of various system and user factors on the eavesdropping accuracy, and developed methods to eliminate these impacts in practical system settings.

We implemented our attack as an Android application running on the victim device, and evaluated the attack on Android smartphones with different Qualcomm Adreno GPUs. A demo video is provided to illustrate our attack[3]. From our experiment results, we have the following conclusions:

- Our attack is *accurate*. It can correctly eavesdrop more than 80% of user's login usernames and passwords over different target applications, and this accuracy retains even when user credentials contain 16 characters.
- Our attack is *adaptive*. It can well adapt to different device models, system settings and user's input behaviors.
- Our attack is *lightweight*. Since our attack does not need repetitive guesses, the inference latency is <0.1ms, and the attack incurs negligible computing overhead and network traffic on the victim device.

We also discussed possible mitigation methods to this attack. Some intuitive methods, such as disabling key press popups and malware detection, could be used to mitigate this attack, but still leave important security vulnerabilities that can be utilized by the attacker to infer useful information about the user's input credentials. Instead, we consider that using role-based access control is a more effective mitigation method, and such access control can be practically enforced in Android systems by either redesigning the Android graphics APIs or utilizing the SELinux Access Manager.

Disclosure: We have reported all of our findings to Qualcomm and Google following their disclosure requirements. We are informed that Google will provide a patch in future Android security updates to mitigate this vulnerability.

2 BACKGROUND AND MOTIVATION

To better understand our attack, we first introduce the basics of GPU overdraw in Android, and then motivate our design of attack by explaining the performance counters of Qualcomm Adreno GPUs that relate to GPU overdraw.

2.1 GPU Overdraw in Android

All graphics contents in Android are rendered in multiple layers based on their layout and components. As shown in Figure 2, GPU renders these layers in a back-to-front order to handle translucent effects and shadows. Parts of the bottom layers, hence, are occluded by the top layers and invisible to the user [23]. GPU overdraw, then, occurs when layers overlap and pixels in the overlapping portion are drawn multiple times.

[3]Link to the demo video: https://youtu.be/f40TvdDaxqw

On Android devices with Qualcomm Adreno GPUs, the screen is divided into equally sized tiles that are separately rendered. The tile sizes are automatically determined by the GPU hardware to reach optimal performance [24], and the amount of overdraw is counted as the number of involved tiles. For example in Figure 2, 3 tiles are involved in 3x overdraw.

The amount of overdraw varies with different screen contents. Hence, we can infer screen contents from values of GPU PCs that reflect such overdraw. Note that, the same screen content (e.g., a character being displayed) may correspond to different PC values in different applications, on different device models, or with different system configurations.

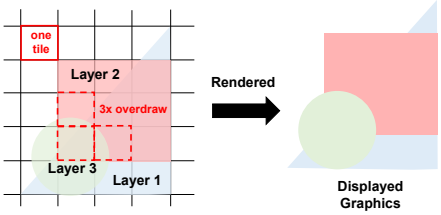


Figure 2: Android graphics rendering with Qualcomm Adreno GPUs. GPU overdraw occurs when layers overlap.

2.2 Performance Counters of Adreno GPUs

GPU performance counters (PCs) are special-purpose hardware registers that count occurrences of GPU hardware events and amounts of hardware resource usage [55]. By design, they are used for performance analysis in program and system debugging, and can also be used to adjust the system configuration based on GPU workload [44]. Hence, they are usually exposed to users and can be queried through GPU drivers.

On Qualcomm Adreno GPUs, there are 3 categories of GPU PCs that relate to GPU overdraw [24], as listed below.

- PCs about **Low Resolution Z (LRZ) pass** record the amount of tiles not rendered by GPU due to occlusion from higher layers. Occluded pixels are discarded early in the rendering pipeline to improve GPU performance. Since character shapes in popups of different key presses are different, they result in different occlusions and hence different values of LRZ-related PCs.
- **Rasterization (RAS)** converts vector graphics into pixels, and RAS-related PCs measure the amount of pixels as the rasterization output. These amounts are determined by vector shapes from all layers, and are different with popups of different key presses.
- **Vertex cache (VPC)** caches vertex data for faster reuse in GPU rendering, and VPC-related PCs measure the amount of such cache use. Popups of different key presses vary the tile occlusions and LRZ passes, and affect such cache use.

As illustrated in Figure 3, the popup of a key press will result in three GPU PC value changes, reflecting different screen changes: 1) when the key is pressed and the popup appears; 2) when the key is released and the text echo appears; 3) when the popup disappears. In most cases, the first PC value change exhibits significant but unique differences over popups of different key presses. Hence, in the rest of this paper, we will only use these first PC value changes for eavesdropping.

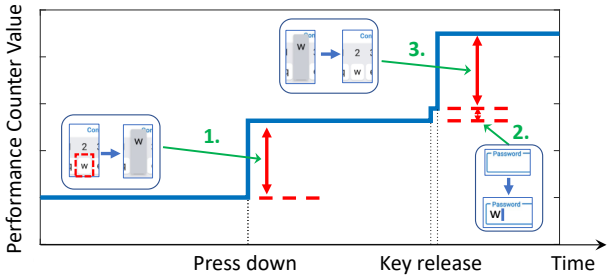


Figure 3: A key press results in 3 GPU PC value changes

3 OVERVIEW

In this section, we first introduce our threat model, and then provide an overview of our attack.

3.1 Threat Model

We assume that an eavesdropping application made by the attacker can be installed and launched on the victim device. The attacker can first build a benign application in popular categories and then embed the malicious codes into it. Since these malicious codes will only involve legitimate Linux system calls, the application can be safely published at Google Play Store and evade Google's malware detection or on-device scanning like Google Play Protect [4] (see Section 9.1). When being executed, the eavesdropping application will silently run in the background as an Android service to query GPU PC data, but will not require any special Android OS permission. As a result, the user will be totally unaware of the eavesdropping application being executed.

Our attack targets sensitive user credentials, i.e., usernames and passwords being used in target applications, such as online banking (e.g., Chase, American Express, Bank of America), investments (e.g., Fidelity, Robinhood, Charles Schwab), and personal credit report (myFICO, Experian, etc.). Our targets also include the corresponding webpages when being launched in Google Chrome. Due to security reasons, these applications and webpages do not remember user's credentials, and the user instead needs to input these credentials every time. In these cases, the login menu will be the first screen that the user sees, and we can eavesdrop credential inputs from the first set of key presses after application launch.

3.2 Attack Overview

As shown in Figure 4, our attack consists of an *Offline Phase* and an *Online Phase*. In the Offline Phase, the attacker emulates all key presses over different device models and configurations to collect a sufficient amount of GPU PC data. The data is used to identify the correlation between key press popups and GPU PC values, and knowledge about such correlation result in classification models that are preloaded to the attacking application for eavesdropping. A separate classification model will be built for each device model and configuration (e.g., the screen resolution and the on-screen keyboard being used).

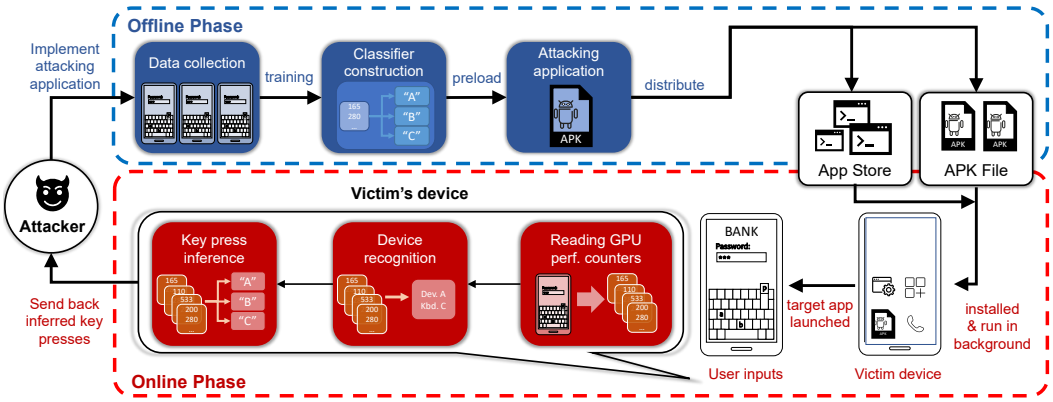


Figure 4: Attack overview

In the Online Phase, the attacking application will spawn a monitoring process, which runs as an Android service in background and uses the existing techniques [14, 15, 49, 50] to detect the launch of target applications[4]. If a target application is launched, the monitoring process will start reading the selected GPU PCs. These readings will be first used to recognize the current device model and configuration, and then applied to the corresponding classification model for eavesdropping. Only the results of eavesdropping are sent back to the attacker.

3.3 Accessing GPU Performance Counters

To access GPU PCs that relate to GPU overdraw, our first step is to identify the names of these PCs on Android systems with Adreno GPUs. We use the function calls in the `GL_AMD_performance_monitor` extension provided by Qualcomm to the standard OpenGL ES APIs [12]. These PCs are grouped with counter IDs in each group. We iterate through all PCs and use `GetPerfMonitorCounterStringAMD()` to get a string identifier for each PC as its description. By doing so, we select PCs in groups of `LRZ`, `RAS` and `VPC`, as listed in Table 1, for eavesdropping. These PCs are specified by the GPU manufacturer, and we have verified that they remain the same over all mainstream Adreno GPU models released after Adreno 540.

Next, the PC values need to be read by the attacking application. The `GL_AMD_performance_monitor` extension can only be used by the attacking application to read the local PC value changes caused by this application itself [28], but cannot provide any global GPU information about other applications. Instead, we access the global GPU information by directly reading the GPU PC values from the GPU device file. More details are described in Section 4.

3.4 Eavesdropping User Inputs

As described in Section 2, the selected GPU PCs reflect the amount of GPU overdraw at the granularity of pixels, and hence exhibit unique value changes for popups of different key presses. To verify this correlation, we conducted experiments over a Oneplus 8 Pro smartphone with Google Keyboard. Figure 5 shows that the PC values remain unchanged if the screen display does not change, and

[4]These techniques have been proved to be highly accurate, and achieves >90% accuracy in >100 target applications.

Table 1: PCs on Adreno GPUs being used for eavesdropping

Group	ID	String identifier
LRZ	13	PERF_LRZ_VISIBLE_PRIM_AFTER_LRZ
	14	PERF_LRZ_FULL_8X8_TILES
	15	PERF_LRZ_PARTIAL_8X8_TILES
	18	PERF_LRZ_VISIBLE_PIXEL_AFTER_LRZ
RAS	1	PERF_RAS_SUPERTILE_ACTIVE_CYCLES
	4	PERF_RAS_SUPER_TILES
	5	PERF_RAS_8X4_TILES
	8	PERF_RAS_FULLY_COVERED_8X4_TILES
VPC	9	PERF_VPC_PC_PRIMITIVES
	10	PERF_VPC_SP_COMPONENTS
	12	PERF_VPC_LRZ_ASSIGN_PRIMITIVES

exhibits significant but different changes with popups of different key presses: 1637 for key 'w' and 1625 for key 'n'. Such difference is further exemplified in Figure 6, and we also verified that for each key, repetitive presses always result in the same change of PC values.

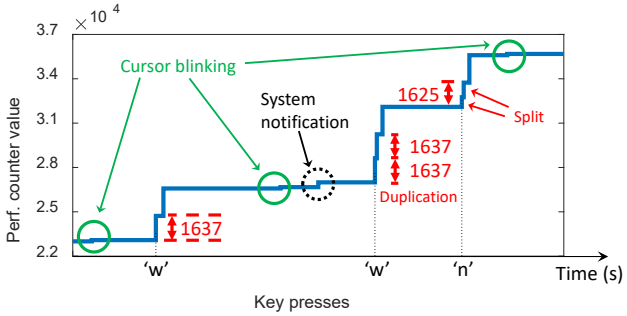


Figure 5: Variations of the `PERF_LRZ_VISIBLE_PRIM_AFTER_LRZ` PC (ID 13 in Table 1) due to different key presses and system factors. Note that only the first PC value change for each key press is measured and used for eavesdropping.

Intuitively, such explicit correlation allows accurate eavesdropping by jointly examining the value changes of all the selected GPU

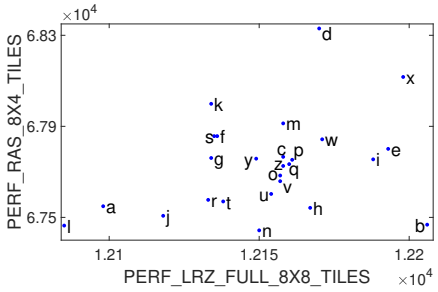
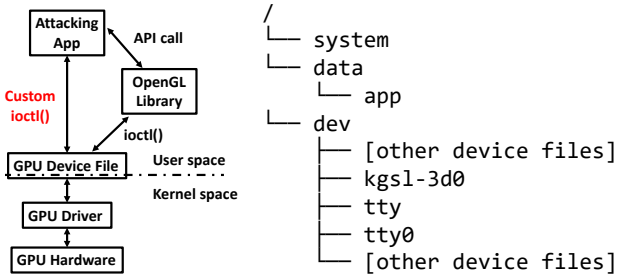


Figure 6: PC value changes with popups of different key presses, with one LRZ PC and one RAS PC as an example.

PCs. However in practice, the eavesdropping accuracy could be impaired due to various system factors:

- **Duplication**: Due to the rich animation of popups on some keyboard (e.g., Google Keyboard), one key press may result in two consecutive PC value changes with the same amount, and incorrectly results in two duplicated key presses being inferred (see the second key press 'w' in Figure 5).
- **Split**: If a PC is being read when the GPU is in the process of drawing the key press popup, the change of this PC could be split into multiple consecutive changes with smaller amounts (see key press 'n' in Figure 5), and hence prevents the key press from being correctly inferred.
- **System noise:** Irrelevant OS behaviors, such as cursor blinking and system notification icons, could also result in changes of GPU PC values.

In addition, the eavesdropping accuracy could also be affected by the heterogeneous user behaviors during credential inputs. The user may switch to other running applications, whose operations may result in irrelevant GPU PC value changes. The user may also press the backspace key to remove and correct the past inputs, which however, could have already been inferred and recorded. The impact of these system and user factors will be addressed in Section 5.



(a) Function call relationship (b) Android device file directories

Figure 7: GPU device file in Android OS

4 READING PERFORMANCE COUNTER VALUES

To effectively read the global values of GPU PCs from the unprivileged attacking application, our basic approach is to bypass the regular APIs provided by OpenGL ES and directly access the GPU device file `/dev/kgsl-3d0`. As shown in Figure 7, this file in Android is part of Qualcomm's *Kernel Graphics Support Layer* (KGSL) to provide an interface for userspace applications to access the GPU hardware [8]. Since this interface is also used by user-space drivers (e.g., OpenGL ES and Vulkan) that run as a shared library with the same PID and SELinux context as the calling user application, it is always accessible to unprivileged user applications [16].

As an interface, the device file could be used by user applications to query GPU PCs through the `ioctl()` Linux system call. As shown in Figure 8, this system call takes a GPU request code, which specifies the GPU PC being selected as its input parameter, and then writes the PC value into the provided memory block.

```
int ioctl (int fd, unsigned long request, void *ptr);
```

Reference to the GPU device file — GPU request code — Pointer to memory block

Figure 8: The `ioctl()` system call

The GPU request codes and data structures for reading GPU PCs, as shown in Figure 9, are specified by the `msm_kgsl.h` header file in the Qualcomm GPU KGSL driver[5]. Based on these specifications, Figure 10 demonstrates the procedure of reading the value of a specific GPU PC from the GPU device file. Before reading the device file, we will need to first notify the GPU hardware to prepare the I/O and make the current PC value available in the device file. Afterwards, the PC value is blockread through the wrapper `data_read` and eventually written into `data.value`.

```
/* Perf counter group IDs*/
#define KGSL_PERFCOUNTER_GROUP_VPC 0x5
#define KGSL_PERFCOUNTER_GROUP_RAS 0x7
#define KGSL_PERFCOUNTER_GROUP_LRZ 0x19
... ... ...
/* Data structure of a perf counter */
struct kgsl_perfcounter_read_group {
    unsigned int groupid;    // Group ID
    unsigned int countable;  // Counter ID
    unsigned long value;     // Perf counter value
    ... ... };
/* To initialize a perf counter query */
#define IOCTL_KGSL_PERFCOUNTER_GET \
    _IOWR(KGSL_IOC_TYPE, 0x38,
    struct kgsl_perfcounter_get)
struct kgsl_perfcounter_get {
    unsigned int groupid;       //   Group ID
    unsigned int countable;     // Counter ID
    ... ...};
/* Blockread to GPU performance counters */
#define IOCTL_KGSL_PERFCOUNTER_READ \
    _IOWR(KGSL_IOC_TYPE, 0x3B,
    struct kgsl_perfcounter_read)
struct kgsl_perfcounter_read {
    struct kgsl_perfcounter_read_group
        *reads;             // Pointer to rx buffer
    unsigned int count;     // Buffer size
    ... ...};
```

Figure 9: The GPU request codes and data structures for reading GPU performance counters, as specified in `msm_kgsl.h`

[5]This file is open sourced and can be found in the Android kernel's source codes from AOSP [20] or open-source graphics libraries such as Mesa [10].

```
/* Open GPU device file */
Int fd = open("/dev/kgsl_3d0", O_RDWR);

/* To initialize a performance counter query */
struct kgsl_perfcounter_get data_get;
data_get.groupid = KGSL_PERFCOUNTER_GROUP_LRZ;
data_get.countable = data_put.countable = 14;
ioctl(fd, IOCTL_KGSL_PERFCOUNTER_GET, &data_get);

/* Specify the perf counter for blockread */
struct kgsl_perfcounter_read_group data;
data.groupid = data_get.groupid;      //  group ID
data.countable = data_get.countable;  // counter ID

/* Specify the pointer to rx buffer in blockread */
struct kgsl_perfcounter_read data_read;
data_read.reads = data;
data_read.count = 1; // The amount of reads

/* Read the value of performance counter*/
ioctl(fd, IOCTL_KGSL_PERFCOUNTER_READ, &data_read);
```

Figure 10: An example of using the `ioctl()` system call to read the `PERF_LRZ_FULL_8X8_TILES` performance counter

In the Online Phase, the attacking application periodically invokes the `ioctl()` system call to read values of selected GPU PCs. By default, the interval of such readings is set to be equal to or slightly smaller than half of the screen refresh interval, to ensure that GPU status for each screen frame is at least covered by one PC reading. We will further investigate the best reading interval in Section 7.4.

5 ACCURATE EAVESDROPPING IN PRACTICAL SYSTEMS

In this section, we tackle the system and user factors that may impair the eavesdropping accuracy in practical systems.

5.1 Addressing the Impacts of System Factors

As described in Section 3.4, inferring user's key presses from the changes in GPU PC values could be affected by various system factors, including duplication, split and system noise. The impacts of these system factors are further illustrated in Figure 11, where a key press may not always result in a single and constant change of the PC value. To demonstrate the impacts of these system factors on inference accuracy, we investigated the value changes of our selected GPU PCs over 3,485 key presses, using Google keyboard on a Oneplus 8 Pro smartphone. Among these key presses, we found 633 duplication cases, 316 split cases and 21 cases with high system noise, indicating that these system factors will make 28% of key presses to be incorrectly inferred.

First, our basic insight on identifying duplications is that the interval between two key presses of a human user is at least hundreds of milliseconds [43], and is much longer than our interval of GPU PC readings (<10ms as described in Section 4). As a result, for every change of the GPU PC value, we will backtrace a time period t_w in the past, and only consider this change as indicating a key press if no key press has been recently inferred within t_w in the past. In practice, the value of t_w should be the shortest possible interval between two key presses, and we choose t_w to be 75ms as suggested in [43].

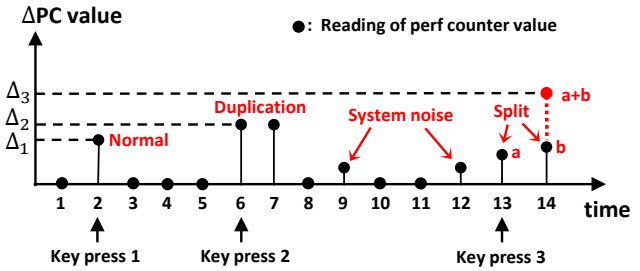


Figure 11: Illustrations of duplication, split and system noise when inferring user's key presses. Δ_1, Δ_2 and Δ_3 indicate the change of PC value corresponding to the key press 1, 2 and 3, respectively, in normal cases.

Second, to tackle split and system noises, we collect GPU PC data about the normal cases of different key presses during the Offline Phase, and then use such data to build a classification model for online eavesdropping: online readings of PC value changes could be considered as valid key presses only if they are close enough to the collected offline data. For example, as shown in Figure 12, readings A are inferred as key press 'w' and readings B are considered as system noise.

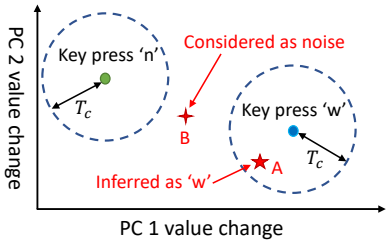


Figure 12: Illustration of the classification model for online eavesdropping, with an example of two GPU PCs. In practical systems, all the selected GPU PCs will be involved to build the classification model in a high-dimension space.

In practice, a classification model is separately built for each device model and device configuration, and all the classification models are preloaded to the attacking application. The classification threshold (T_C) will be decided accordingly to eliminate any false positives. For example, on a Oneplus 8 Pro smartphone with Google Keyboard, our collected offline data shows that the maximum difference in PC value change between cases of split or system noises and normal key presses is 370, which is then used as the value of T_C.

Then, our solution to split and system noises is shown in Algorithm 1. For each new change R of PC values, we first examine if it can be classified to a key press with our classification model. If not, we backtrace to combine R and the previous PC change $R_{previous}$, and see if $[R_{previous}, R]$ is a split (like the readings at time 13 and 14 in Figure 11). If R cannot be inferred as a key press in both steps, it will be considered as system noise (like readings at time 9 and 12 in Figure 11).

Algorithm 1: Online algorithm to tackle split and system noises

Output: Set of inferred key presses **E** and timestamps of these key presses **M**

```
while new PC readings R received at time t do
    key, d_min ← SearchMinDist(R) ;          // Apply to classification model
    if d_min < T_C then
        E ← E ⋃ {key} ;                      // Not system noise
        M ← M ⋃ {t};
    else
        key, d_min ← SearchMinDist(R + R_previous) ;
        // Check if it is split by with the previous PC reading R_previous
        if d_min < T_C then
            E ← E ⋃ {key} ;                  // Not split
            M ← M ⋃ {t_previous};
```

Algorithm 1 is a greedy algorithm and may make mistakes, because it will combine two consecutive PC value changes and infer the combination into a key press whenever possible. For example in Figure 11, at time 13, the algorithm combines PC value changes at time 12 and 13, and may incorrectly infer a key press at time 12. Addressing this limitation requires knowledge about the entire trace of GPU PC readings, meaning that eavesdropping can only be done after the user input finishes. In Section 7, we will experimentally investigate such tradeoff between the accuracy and timeliness of eavesdropping.

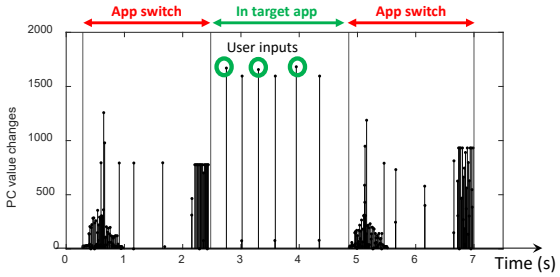


Figure 13: PERF_LRZ_VISIBLE_PRIM_AFTER_LRZ value changes when switching between applications

5.2 Recognizing Application Switch

In practical scenarios, before finishing the credential input, the user may switch to and use other applications at any time. Operations in these applications may also create irrelevant GPU PC value changes and hence confuse our eavesdropping. The key to addressing this issue is how to correctly decide if a GPU PC value change is caused by our target application (e.g., the ones listed in Section 3.1), and we find that specific GPU PCs, such as `PERF_LRZ_VISIBLE_PRIM_AFTER_LRZ`, show strong features when the user switches to another app.

As shown in Figure 13, due to the rich animation and screen display changes[6], there will be fierce value changes of these PCs at the beginning and end of app switch procedure, and the interval between these value changes (e.g., <50ms) will be much smaller than that between human typings. Hence, we could reliably use these PCs to detect application switches and only eavesdrop when the user types in the target application.

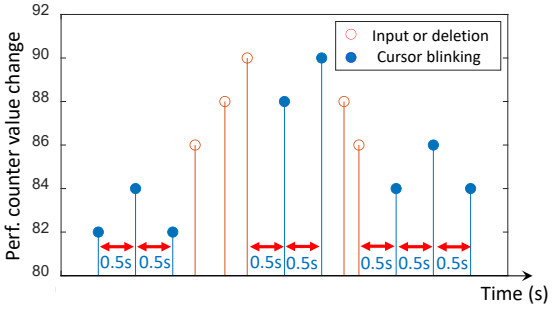


Figure 14: PERF_LRZ_VISIBLE_PRIM_AFTER_LRZ value changes with 3 letter inputs and then 2 letter deletions

5.3 Eavesdropping with Input Corrections

Deleting and correcting the past input by pressing the backspace key is also a common behavior during user input, and it is necessary to correctly detect such input corrections to make sure that the deleted inputs are not included in the eavesdropping results. The difficulty of such detection is that pressing the backspace key does not trigger a popup on most on-screen keyboards. Instead, we observed that the value changes of the `PERF_LRZ_VISIBLE_PRIM_AFTER_LRZ` have strong correlation with the current input length: as shown in Figure 14, the PC value strictly increases by 2 with a new input character and decreases by 2 whenever an input character is deleted by backspace. We can hence use the value changes of this PC to detect input corrections.

In particular, the values of this PC may also be changed by cursor blinking. However, since cursor blinking in most systems has a fixed interval of 0.5 seconds, these PC value changes can be recognized according to their timestamps.

6 IMPLEMENTATION

The source codes of our implementations are released in a GitHub repository[7].

Offline Phase: We implement a bot program in Python to automatically emulate all on-screen key presses by injecting screen input commands through the Android `/dev/input/eventX` interface, so as to collect GPU PC data. This program runs in the Termux emulator on rooted Android devices under the attacker's control, and the collected data is stored in the device's local storage.

During data collection, we launch a target application and run the bot program in background to generate text inputs in the target application. As shown in Figure 15, we connect the device to a host

[6]To switch between apps, the user first presses the bottom app switcher button to show the app overview screen, and then select the app to switch.
[7]https://github.com/perfinfer/code

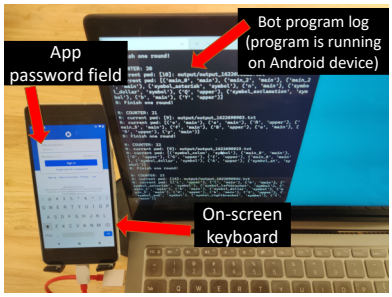


Figure 15: The bot program for offline data collection

system and remotely log into the device's Android system via ADB to monitor the data collection process.

Online Phase: The attacking application launches an Android service in background to continuously read GPU PC values. To maximize the performance and ensure timely readings of GPU PC values, we also incorporate the functionality of online inference from GPU PC values into this service, and implement this service in C++ using Android NDK [29, 48].

7 EVALUATION

We evaluate the accuracy, timeliness, overhead and adaptability of our attack on Android smartphones with different Qualcomm Adreno GPUs. Different system configurations (e.g., the screen refresh rate and on-screen keyboard being used) are applied and evaluated on these devices. By default, the values of the selected GPU PCs are read every 8ms, and we will also evaluate the impacts of different reading intervals.

Our experiments are conducted by running target applications on the smartphone and using the bot program implemented in Section 6 to emulate different key presses. The attacking application will then be running on the smartphone to read GPU PCs and infer the emulated key presses.

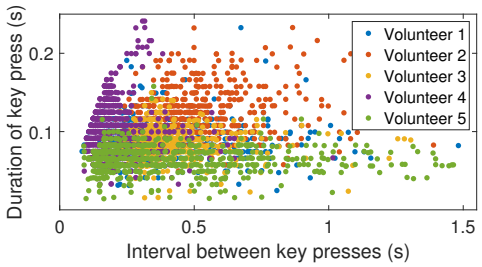


Figure 16: Durations and intervals of key presses

To mimic real human inputs when emulating these key presses, we collect human input data by having 5 student volunteers to randomly type text strings on a Oneplus 8 Pro smartphone with Google Keyboard. The lengths of these text strings randomly vary between 8 and 16, and each student volunteer types 50 times. The durations[8] of these key presses and intervals between key presses, as shown in Figure 16, exhibit noticeable heterogeneity across different student volunteers, and will be used to emulate key presses in our experiments.

7.1 Inference Accuracy

We first evaluate the accuracy of our attack over the Chase Mobile application[9]. Our experiments are being conducted on a Oneplus 8 Pro smartphone with Google Keyboard, and the length of username and password ranges between 8 and 16. For each length, 300 random texts are emulated using the durations and intervals of key presses shown in Figure 16. Results in Figure 17(a) show that the accuracy of our attack is always higher than 75% and the average accuracy is 81.3%. Furthermore, Figure 17(b) shows that only 1 key press is incorrectly inferred for most text inputs, and our average accuracy of inferring individual key presses is 98.3%. In practice, such single errors in inference could be addressed with a small number of guesses.

The accuracy over different groups of key presses is shown in Figure 17(c), and the accuracy over individual key presses is shown in Figure 18. These results show that most errors happen on a few key presses, such as the symbols ',' and '.' that result in the minimum amount of GPU overdraw.

In comparison, we also evaluated the eavesdropping accuracy using the GPU PCs suggested in [37] over desktop Nvidia RTX 2070 GPU. The Nvidia 470.57 driver is used in evaluation on Linux Ubuntu 20.04, and we use the Nvidia CUPTI interface to read GPU PCs every 10ms. In our experiments, a bot program repeatedly type characters into the `gedit` text editor, Gmail login webpage in Chrome and the login fields in Dropbox client app, with an interval of 0.5s for 10 times. As shown in Table 2, when the collected PC traces are applied to different classification algorithms, the eavesdropping accuracy is lower than 14%. These results demonstrate the ineffectiveness of existing work [37] on eavesdropping user's keyboard inputs that incur negligible amounts of GPU workloads.

Table 2: The eavesdropping accuracy of existing work [37] that uses PCs of desktop Nvidia GPUs

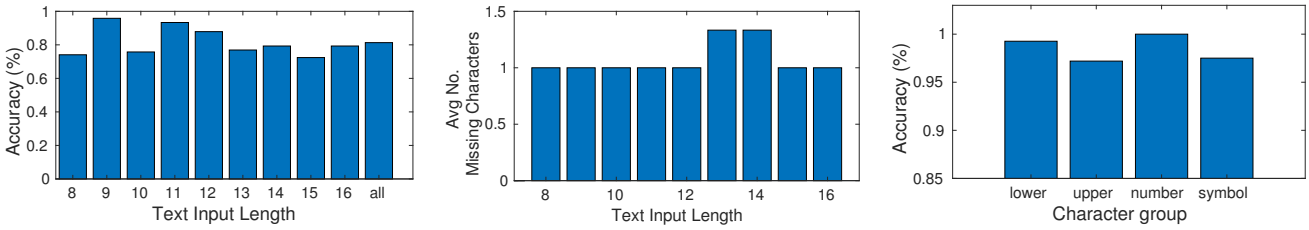
	gedit	Gmail web	Dropbox client
Naive Bayers	8.7%	9.5%	8.9%
KNN3	8.4%	9.2%	9.7%
Random Forest	13.7%	14.2%	14.0%

Accuracy over different target applications: We evaluate the accuracy on target applications of online banking (Chase and Amex), online investment (Fidelity and Charles Schwab) and personal credit report (MyFICO and Experian). Website access to some target applications in Google Chrome are also evaluated. Results in Figure 19 show that our attack can be applied to target applications in all categories, and its accuracy of inference over these applications is always higher than 80%.

Accuracy on different on-screen keyboards: We use a Oneplus 8 Pro smartphone to evaluate the accuracy on 6 popular on-screen keyboards: Microsoft Swift, Google Keyboard, Sogou Keyboard, Google Pinyin Keyboard, Go Keyboard and Grammarly Keyboard.

[8]The duration of a key press starts when the key is being pressed and ends when the key is being released.

[9]https://play.google.com/store/apps/details?id=com.chase.sig.android



(a) Accuracy of inferring text inputs

(b) Average number of incorrectly inferred key presses in each text input

(c) Inference accuracy in different groups

Figure 17: Accuracy of inferring user's text inputs

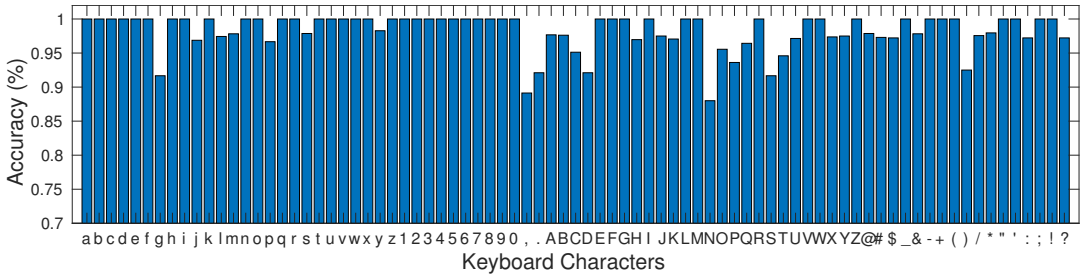


Figure 18: Inference accuracy over individual key presses

Figure 20 shows that, even though these keyboards have different UI designs, our attack retains high inference accuracy on all cases, with <5% variation.

7.2 Impact of User Input Speed

To evaluate the impact of different user inputs speeds, we split the key presses that we collected from 5 student volunteers into 3 parts with the same amount, based on the interval between key presses. We then use these 3 parts to emulate different speeds of user inputs: fast (typing interval <0.24s), medium (typing interval between 0.24s and 0.4s) and slow (typing interval >0.4s), and apply each of these parts to 300 random text inputs in target applications.

The impact of different input speeds is shown in Figure 21. While the accuracy of inferring individual key presses remains constant, Figure 21(a) shows that the accuracy of inferring user's text inputs drops to 60% when the input speed decreases. The main reason is that when the interval of reading GPU PCs remains unchanged, infrequent key presses have higher chances of involving random system noise. However, as shown in Figure 21(b), the average number of errors is <1.3, allowing low-cost correction with few guesses.

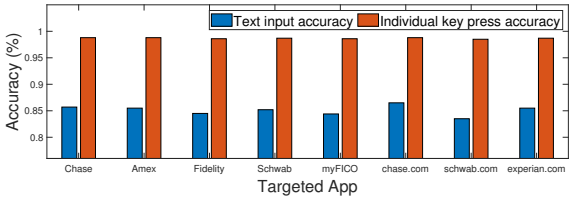


Figure 19: Inference accuracy on different target apps

7.3 Impact of CPU and GPU Workloads

Our attack runs a monitoring process in background, and its access to GPU PCs could be affected by other concurrent CPU and GPU workloads. To evaluate the impacts of these concurrent system workloads, we emulate different CPU workloads by running a multi-threaded process that occupies all CPU cores by a varying percentage, and emulate different GPU workloads by running a custom program that invokes OpenGL ES APIs to render 3D objects in background using GPU[10].

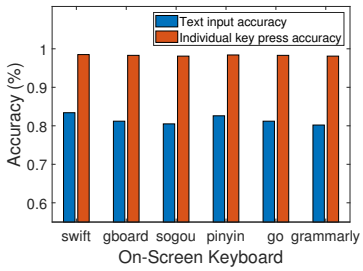
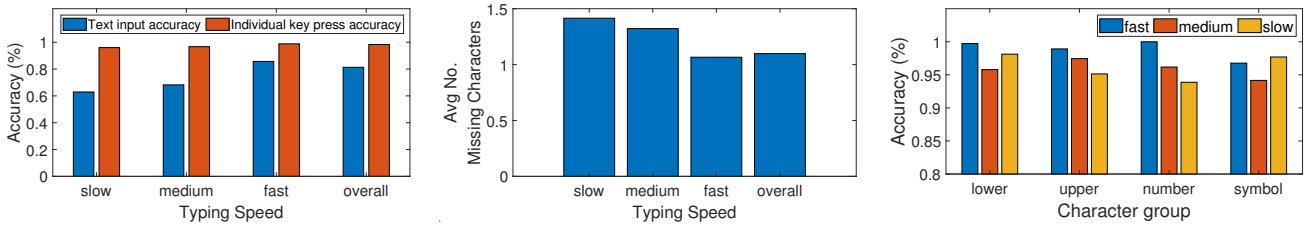


Figure 20: Inference accuracy on different keyboards

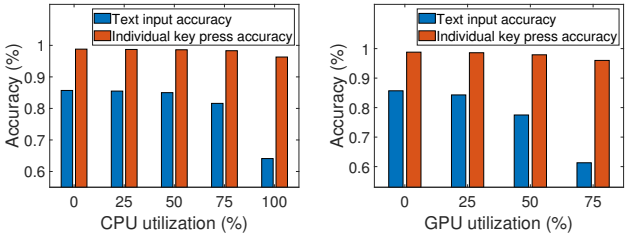
As shown in Figure 22, our accuracy experiences negligible reduction when the CPU workload is <50% or the GPU workload is <25%, but will drop to 60% if such workloads both increase to 75%. The basic reason is that when the Android system has heavy CPU or GPU workloads, our attacking application has to compete CPU or GPU access with other applications and hence may be unable to

[10]The current GPU utilization ratio is retrieved through the `/sys/class/kgsl/kgsl-3d0/gpu_busy_percentage` interface.



(a) Inference accuracy with different speeds (b) Average number of incorrectly inferred key presses in each text input (c) Inference accuracy in different groups

Figure 21: The impact of different speeds of user inputs



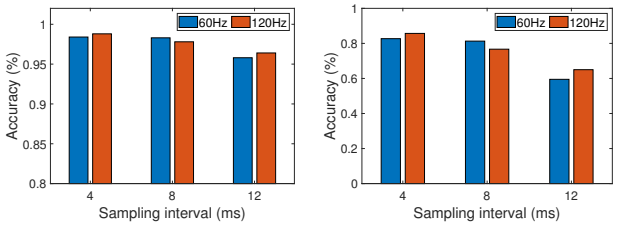
(a) Inference with CPU workloads (b) Inference with GPU workloads

Figure 22: Impacts of CPU and GPU workloads

timely read GPU performance counters. However in practice, most of our target applications produce only a small amount of CPU or GPU workloads in their login menus.

7.4 Impact of Interval Reading GPU PCs

As discussed in Section 4, the GPU PCs should be at least read once for each screen frame being rendered, to ensure timely inference of any user input. To evaluate the impact of interval reading GPU PCs, we adjust the screen refresh rate on a Oneplus 8 Pro smartphone between 60Hz and 120Hz. The inference accuracy with different intervals of reading GPU PCs, in these cases, is shown in Figure 23. While the accuracy of inferring individual key presses can be generally retained to be >95%, the accuracy of inferring text inputs drops by 20% if such interval increases to 12ms. As a result, we recommend that such interval should not be longer than 8ms if the screen refresh rate is 60Hz, and should be at most 4ms for 120Hz.

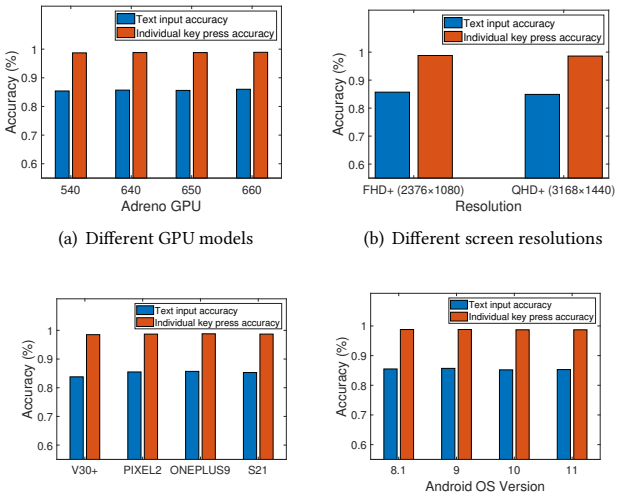


(a) Accuracy of infering key presses (b) Accuracy of inferring text inputs

Figure 23: Accuracy with different reading intervals

7.5 Adaptability of Attack

In this section, we evaluate the accuracy of our attack over different Android device models and configurations. Our experiments are conducted over the following smartphone models: LG V30+ (Adreno 540 GPU and Android 9), Google Pixel 2 (Adreno 540 GPU and Android 10), Oneplus 7 Pro (Adreno 640 GPU and Android 11), Oneplus 8 Pro (Adreno 650 GPU and Android 11), Oneplus 9 (Adreno 660 GPU and Android 11), and Samsung Galaxy S21 (Adreno 660 GPU and Android 11). Results in Figure 24 show that, since our attacking application carries preloaded classification models for each device model and configuration, it retains similar accuracy over all the different system situations. In particular, note that in Figure 24(c), we compared the inference accuracy over different smartphone models with the same GPU (e.g., LG V30+ and Google Pixel 2 with Adreno 540, Oneplus 9 and Samsung Galaxy 21 with Adreno 660), and demonstrated that different smartphone manufacturers and software systems have negligible impacts on the eavesdropping accuracy.



(a) Different GPU models (b) Different screen resolutions

(c) Different phone models with the same GPU (d) Different Android OS versions

Figure 24: Adaptability of attack

7.6 Timeliness and Overhead

The amount of computing time needed for our inference is shown in Figure 25, which shows the histogram of inference time over 3,300 key presses on a Oneplus 8 Pro smartphone. These results show that more than 95% of key presses can be inferred within 0.1ms.

These results also verified that our attack produces a negligible amount of computing overhead on the victim device and is hence hard to be detected.

We also evaluated the device power being consumed by our attack. Results in Figure 26 show that our attack consumes a maximum of 4% extra power after two hours of use.

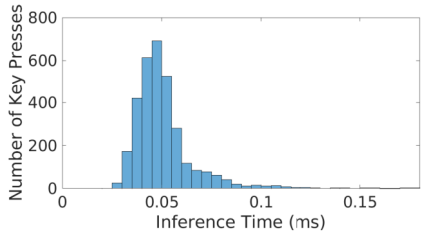


Figure 25: Computing time needed for eavesdropping

The data size of one classification model in our attack is 3.59 kilobytes on average. The size of our attacking application, hence, is at most 13.40 megabytes[11] with 3,000 preloaded classification models to cover 100 smartphone models, 15 different on-screen keyboards on each smartphone model, and 2 screen resolutions on these smartphone models.

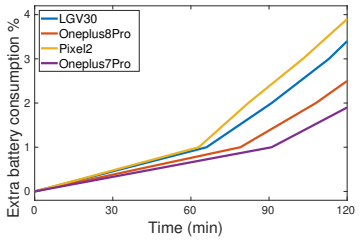


Figure 26: Power consumption for inferring user inputs

8 EXPERIMENTATION WITH PRACTICAL USE

In this section, we further evaluate the accuracy of our attack in practical usage sessions, where the user is actually using our attacking application on the victim device and will perform random app switch and input corrections. In our experiments, 5 student volunteers use a Oneplus 8 Pro smartphone to input random texts over 3 different apps[12] for 3 minutes. They are requested to randomly switch between these apps, make corrections to their inputs and perform other UI interactions. After each input, they are also asked to freely use other apps installed on the device within the 3 minutes. Each experiment is repeated for 10 times, and some sample traces of these user behavior events are shown in Figure 27.

As shown in Figure 28, the average accuracy of eavesdropping individual key press inference is 97.1%, and average accuracy of input eavesdropping is 78.0%. This accuracy slightly drops compared to those results in Section 7.1 due to the handling of input corrections, but is still sufficient to ensure practical eavesdropping with a few guesses.

[11]The maximum application size allowed by Google Play Store is 100 megabytes.
[12]The apps used in each experiment are randomly chosen from six apps, including Chase, Amex, Fidelity, Charles Schwab, MyFICO and Experian.

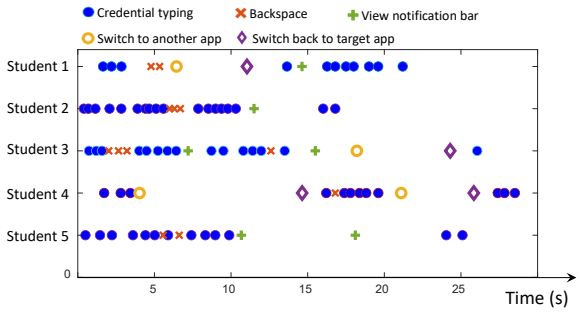


Figure 27: User behavior events during experiments

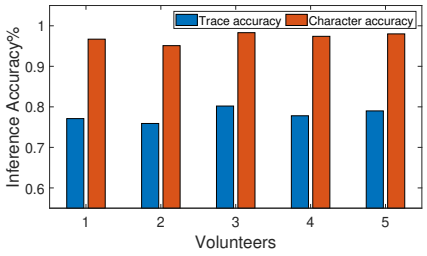


Figure 28: Accuracy in practical usage

9 ATTACK MITIGATION

9.1 Simple Mitigation Methods

We will first discuss some simple and intuitive methods to mitigate our attack.

Disabling popups of key presses. The most straightforward mitigation method is to disable keyboard popups of key presses in Android settings. For example, on a Google Pixel 6 smartphone with Android 12 and Google Keyboard, this disabling option can be found at System→Languages & Input→On-screen keyboard→Gboard→Preferences→Popup on keypress. Although this method can prevent direct eavesdropping of user inputs, it may cause inconvenience for many users, especially those with vision problems, as popups are designed as feedback to help verify that the correct key is being pressed. Furthermore, since this method did not disable user applications' access to GPU PCs, the attacker can still infer useful information about the user's input credentials, such as the input length, from GPU PCs as described in Section 5.3.

Malware detection. Another simple approach to attack mitigation is to rely on Google's malware detection on Play Store [4] or on-device detection such as Google Play Protect [21] to detect abnormal behaviors in our attacking application, such as frequent `ioctl()` calls. However, since `ioctl()` is a standard Linux system call and the only Android interface to access the GPU hardware [52], it is invoked at a very high frequency in normal OS operations (e.g., thousands of invocations per second are expected in such normal cases [46]). As a result, frequent invocations of this call for GPU PC queries will not be considered by Android as abnormal. Other existing approaches on run-time malware detection that are based on behavior recognition and classification [42, 45], on the other hand, are usually considered difficult to be integrated into commodity Android OS systems.

To further verify this, we built a testing application that periodically queries GPU PCs using the `ioctl()` call and submitted it to Google Play Store. The application passed all Google's checks and has been published[13].

9.2 Mitigation through GPU PC Access Control

A more viable solution to mitigating this attack is to apply access control on GPU PCs in Android. The most intuitive approach is to completely disable any smartphone application from accessing any GPU PC, but is practically infeasible due to its big impact on many applications' executions. First, access to GPU PCs is the foundation of GPU profiling and debugging tools provided by Qualcomm and Google, and disabling such access will prevent these tools from being usable. Second, access to GPU PCs allows many run-time tuning and adaptations for applications to optimize their performance. For example, the run-time information about GPU overdraw could be used to adjust the application's 3D rendering strategy and save GPU resources.

Instead, a better mitigation is role-based access control (RBAC) that limits the access to GPU PCs within a safe scope. RBAC has been adopted for desktop GPUs (e.g., by Nvidia [37]) at a coarse granularity, such that unprivileged applications are prohibited from accessing any GPU PCs and privileged applications have unrestricted access to GPU PCs. However, this coarse-grained RBAC cannot be applied on Android systems, where the Android security model prohibits escalating any user application's privilege to root or administrator [47, 56]. In these cases, RBAC should be enforced at a finer granularity, so that only listed applications are allowed to access the global values of GPU PCs and all other applications can only access their local values of GPU PCs. In practical Android systems, such RBAC can be possibly implemented in multiple ways, which are discussed in detail as follows.

First, as discussed in Section 3.3, the access to each application's local GPU PCs is provided as part of OpenGL ES APIs. Hence, RBAC on such local GPU PC access can be enforced by redesigning the Android interfaces provided for GPU hardware access, so that the graphics APIs are running as separate processes and the Android OS kernel can effectively identify whether the `ioctl()` calls are made through standard system interfaces. Details about such redesign, however, is out of the scope of this paper.

Second, RBAC on GPU PCs can be enforced through Android's security-enhanced Linux (SELinux), which has been used in Android OS to enforce mandatory access control (MAC) overall processes [18]. In SELinux, all the running processes are assigned roles based on their functionality, and their accesses to system resources and functions are then monitored and controlled by the SELinux Access Manager, based on a set of pre-defined policies provided by the OS. As a result, the SELinux-based `ioctl()` command whitelisting [52] can be used to filter suspicious `ioctl()` calls that try to illegally access GPU PCs, by adding GPU PC access rules to SELinux policies. For example, unprivileged user applications (with the `untrusted_app` role in Android) can be prohibited from accessing any global value of GPU PCs. Development of such RBAC based on SELinux will be our future work.

[13] App link: https://play.google.com/store/apps/details?id=me.alittletool.rgbdisplaycolor

9.3 Other Mitigations

Some other mitigation methods could be more effective, but would require collaborations and significant efforts from app/OS developers or user experience changes in authentication procedure.

Using password manager or biometric login. Using password manager or biometric login could avoid text typing using on-screen keyboards. However, this approach needs wider deployment of biometric sensors on smartphones and high adoption of password managers in the big population [40, 51]. In addition, users also need to manually enter their credentials on first-time login, which is still vulnerable to our attack.

Obfuscation on GPU PC values. Obfuscating the GPU PC values could confuse the attacker. Some target applications contain decorative animations on their login menu and could hence defend against this attack. For example, the PNC Mobile Bank application, as shown in Figure 29, reduces our eavesdropping accuracy to 30.2% with such animation. Obfuscation could also be more effectively applied from the OS, by randomly executing small GPU workloads in background. The major challenge, however, is how to decide the appropriate amount of these workloads, as excessive GPU workloads impair the system's performance. This is an open research question, and we believe that it is worth further investigation.

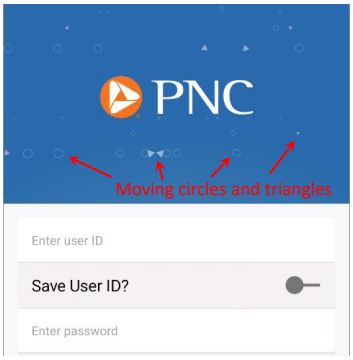


Figure 29: Animations on the PNC Mobile Bank application

10 RELATED WORK

Side channel attacks on desktop GPUs. Attacks on Nvidia GPUs use CUDA APIs to infer various types of user activities [32] or system computing information [37]. These attacks, however, are not applicable to mobile systems due to the large difference in GPU hardware architecture and availability of APIs. Some other approaches include exploiting sensitive data left in uninitialized GPU memory [30] and using OpenGL and OpenCL libraries for microarchitectural attacks on GPU cache contents [11, 37]. However, as we have shown in Section 7.1, the strengths of these attacks depend on the amount of GPU workloads caused by user activities. They are hence limited to inferring coarse-grained user activities, and cannot be used to eavesdrop users' keyboard inputs that incur negligible amounts of GPU workloads.

Side channel attacks on mobile devices. IMU sensor readings on mobile devices have been used to infer user inputs such as PIN

inputs [33], graphical patterns [1] and password inputs [26, 35, 39], but are known to be very sensitive to random noise and heterogeneous human behavior patterns. Mobile CPU caches have also been exploited for side channel attacks, to infer coarse-grained user activities [15, 31]. Some recent work further demonstrated the possibility of inferring users' PIN inputs from knowledge about cache behaviors [53], but only achieved low accuracy (30%). In contrast, our proposed attack achieves much higher accuracy (>80%) without requiring any guess, and hence has higher applicability in practical systems.

Side channel attacks on mobile GPUs. Existing attacks on mobile GPUs used GPU timing information to infer ciphertexts in AES encryption [27], and reverse engineered GPU hardware to implement Rowhammer attack on GPU memory [11]. These attacks, however, are not focusing on eavesdropping keyboard inputs and are orthogonal to our work. Google's Project Zero recently reported a security vulnerability of Qualcomm Adreno GPUs [16], which allows the attacker to overwrite device memory or obtain root control through the `ioctl()` system call. However, this attack does not involve eavesdropping user inputs. The fix provided by Google, on the other hand, did not disable user applications from using the `ioctl()` system call or accessing the GPU device file.

11 CONCLUSION

In this paper, we present new side channel attacks on mobile GPUs that allow unprivileged applications to eavesdrop the user's input credentials typed through the on-screen keyboard. Our attack builds on the different amounts of GPU overdraw caused by popups of key presses, and infers key presses from the corresponding changes of GPU PC values. Experiment results demonstrate that this attack can achieve >80% accuracy of eavesdropping and can be widely applied to smartphones with Qualcomm Adreno GPUs.

ACKNOWLEDGMENTS

We thank our shepherd, Nandita Vijaykumar, and the anonymous reviewers for their comments and feedback. This work was supported in part by the National Science Foundation (NSF) under grant number CNS-1812407, CNS-2029520 and IIS-1956002.

A ARTIFACT APPENDIX

A.1 Abstract

We provide the source codes and the compiled Android application executable (i.e., the .apk file) to launch our proposed attack of eavesdropping the user login credentials on Android devices. With the eavesdropping application running in background, the user's credential inputs using the on-screen keyboard can be eavesdropped, and the eavesdropping results will be displayed in real-time at the notification bar of the device.

As described in the paper, our attack can be applied to a wide range of Android devices, with the attacking application's parameters being customized for each specific device hardware model and software configurations. To facilitate artifact evaluation, we have customized the source codes and application executable being provided with respect to the Google Pixel 5 smartphone with Android 11, Google keyboard and certain user interface configuration being used, as described below. More details about how to customize parameters for different Android devices can be found in the paper.

A.2 Artifact Checklist

- **Program:** Android application for the eavesdropping attack.
- **Compilation:** Android Studio 2020.3.1 Patch 3.
- **Transformations:** No transformation tools required.
- **Binary:** Source codes are included to build binaries.
- **Run-time environment:** Android 11.
- **Hardware:** Google Pixel 5 smartphone (GD1YQ, unlocked).
- **Run-time state:** Not sensitive to runtime state.
- **Execution:** App execution on the Android device described above.
- **Metrics:** Qualitative result evaluation.
- **Output:** Eavesdropping result being displayed in the Android notification bar.
- **Experiments:** Start the eavesdropping App, and then type login credentials in the selected user applications or webpages.
- **How much disk space required (approximately)?:** 200 MB.
- **How much time is needed to prepare workflow (approximately)?:** Approximately 1 hour.
- **How much time is needed to complete experiments (approximately)?:** Approximately 40 minutes.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** MIT License.
- **Workflow framework used?:** No.
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.5733423.

A.3 Description

A.3.1 How to Access. This artifact can be downloaded from the DOI link https://doi.org/10.5281/zenodo.5733423.

A.3.2 Hardware Dependencies. To build the Android app from the source codes, a generic PC with Android Studio being installed is needed. Running the artifact requires a certain model of Android smartphone device (Google Pixel 5 GD1YQ unlocked).

A.3.3 Software Dependencies. The attacking app needs to be built using Android Studio 2020.3.1 Patch 3. The runtime environment of the app requires that the smartphone uses its factory-set OS and software configurations as listed below:

- Android 11 (RQ3A.210605.005)
- Google Keyboard (GBoard 11.1.04.397969183-release-arm64-v8a)

The attacking app being provided has been tested to be functional over the following user applications:

- Google Chrome (the latest version on Google Play store)
- Chase Mobile Banking App (version 4.256)
- Amex Mobile App (version 6.48.1)

A.4 Installation

This section describes the steps of building and installing the eavesdropping app and setting up the runtime environment.

To build the eavesdropping app, the provided artifact needs to be retrieved and unpacked onto a PC with stable Internet connection and Android Studio installed. Use Android Studio to open the extracted artifact directory with name `androidapp`. Select "Build – Make Project" in the menu to build the App. To install the built App, enable the Developer options submenu on Android device and

enable USB Debugging option under Developer options submenu. After connecting the Android device to the PC through USB cable, select "Run – Run app" in Android Studio to install the built app into Android device.

Alternatively, a prebuilt copy of the app executable is also provided in the artifact, as `androidapp/app-release.apk` file in the artifact. To install the prebuilt App, copy .apk file onto Android device, and click the file in the file browser on Android system.

Third-party applications being involved (e.g., Google keyboard, Chase Mobile App, Amex Mobile App) are also provided as .apk packages with the artifact. To install them, copy the .apk files under `3rdparty` directory of the extracted artifact onto Android device. Finish installation by clicking the .apk files in the file browser on Android system.

The following default Android OS configurations on Pixel 5 device are required in reproducing the paper's experiment results. Please check these settings before the experiment in the Settings App of Android OS:

- System dark theme enabled (in Settings – Display)
- Smooth Display enabled (in Settings – Display)
- Gesture navigation enabled (in Settings – System – Gestures)

A.5 Experiment Workflow

After the runtime environment has been properly set up, the following steps can be performed for each eavesdropping experiment:

(1) Launch the installed eavesdropping app.
(2) Click "START SERVICE" button on top-left corner of the eavesdropping app.
(3) Launch a victim app (e.g., Chase Mobile or Amex Mobile as listed above), or visit certain webpages (e.g., https://m.facebook.com, https://instagram.com) in Chrome.
(4) In the victim app's password input field, type numbers and alphabet letters at your choice.

A.6 Evaluation and Expected Results

After each experiment, the eavesdropped user credential input is expected to show up in the notification bar of Android. To verify the eavesdropping accuracy as described in Section 7.1 of the paper, the user can repeat the experiments described above with different keyboard inputs.

A.7 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html

REFERENCES

[1] Adam J Aviv, Benjamin Sapp, Matt Blaze, and Jonathan M Smith. 2012. Practicality of accelerometer side channels on smartphones. In *Proceedings of the 28th annual computer security applications conference (ACSAC '12)*. Association for Computing Machinery, New York, NY, USA, 41–50. https://doi.org/10.1145/2420950.2420957

[2] Liang Cai and Hao Chen. 2011. TouchLogger: Inferring Keystrokes on Touch Screen from Smartphone Motion. *USENIX Summit on Hot Topics in Security (HotSec)* (2011).

[3] Sen Chen, Lingling Fan, Guozhu Meng, Ting Su, Minhui Xue, Yinxing Xue, Yang Liu, and Lihua Xu. 2020. An empirical assessment of security risks of global android banking apps. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 1310–1322.

[4] Xiao Chen, Chaoran Li, Derui Wang, Sheng Wen, Jun Zhang, Surya Nepal, Yang Xiang, and Kui Ren. 2019. Android HIV: A study of repackaging malware for evading machine-learning detection. *IEEE Transactions on Information Forensics and Security* 15 (2019), 987–1001. https://doi.org/10.1109/TIFS.2019.2932228

[5] Anupam Das, Nikita Borisov, and Matthew Caesar. 2016. Tracking Mobile Web Users Through Motion Sensors: Attacks and Defenses.. In *NDSS*.

[6] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. 2010. Privilege escalation attacks on android. In *international conference on Information security*. Springer, 346–360. https://doi.org/10.1007/978-3-642-18178-8_30

[7] Sanorita Dey, Nirupam Roy, Wenyuan Xu, Romit Roy Choudhury, and Srihari Nelakuditi. 2014. AccelPrint: Imperfections of Accelerometers Make Smartphones Trackable.. In *NDSS*.

[8] Mohammad Javad Dousti, Majid Ghasemi-Gol, Mahdi Nazemi, and Massoud Pedram. 2015. ThermTap: An online power analyzer and thermal simulator for Android devices. In *IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. 341–346. https://doi.org/10.1109/ISLPED.2015.7273537

[9] Facebook. 2019. Facebook advisory for CVE-2019-3568. https://www.facebook.com/security/advisories/cve-2019-3568.

[10] Freedesktop.org. 2021. The Mesa 3D Graphics Library. https://mesa3d.org/.

[11] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2018. Grand pwning unit: Accelerating microarchitectural attacks with the GPU. In *2018 IEEE Symposium on Security and Privacy (SP)*. 195–210. https://doi.org/10.1109/SP.2018.00022

[12] Dan Ginsburg. 2007. AMD_performance_monitor. https://www.khronos.org/registry/OpenGL/extensions/AMD/AMD_performance_monitor.txt.

[13] Daniel Gruss, Erik Kraft, Trishita Tiwari, Michael Schwarz, Ari Trachtenberg, Jason Hennessey, Alex Ionescu, and Anders Fogh. 2019. Page cache attacks. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19)*. 167–180. https://doi.org/10.1145/3319535.3339809

[14] Berk Gulmezoglu, Thomas Eisenbarth, and Berk Sunar. 2017. Cache-based application detection in the cloud using machine learning. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security (ASIA CCS '17)*. 288–300. https://doi.org/10.1145/3052973.3053036

[15] Berk Gulmezoglu, Andreas Zankl, M Caner Tol, Saad Islam, Thomas Eisenbarth, and Berk Sunar. 2019. Undermining user privacy on mobile devices using ai. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security (Asia CCS '19)*. ACM New York, NY, USA, 214–227. https://doi.org/10.1145/3321705.3329804

[16] Ben Hawkes and Project Zero. 2020. Project Zero: Attacking the Qualcomm Adreno GPU. https://googleprojectzero.blogspot.com/2020/09/attacking-qualcomm-adreno-gpu.html.

[17] Bo-Jhang Ho, Paul Martin, Prashanth Swaminathan, and Mani Srivastava. 2015. From pressure to path: Barometer-based vehicle tracking. In *Proceedings of the 2nd ACM International Conference on Embedded Systems for Energy-Efficient Built Environments*. 65–74. https://doi.org/10.1145/2821650.2821665

[18] Google Inc. 2020. Security-Enhanced Linux in Android. https://source.android.com/security/selinux.

[19] Google Inc. 2021. Google Android fts_driver_test_write Heap-based Buffer Overflow Privilege Escalation Vulnerability. https://www.zerodayinitiative.com/advisories/ZDI-21-279/.

[20] Google Inc. 2021. include/uapi/linux/msm_kgsl.h - kernel/msm - Git at Google. https://android.googlesource.com/kernel/msm/+/android-7.1.0_r0.2/include/uapi/linux/msm_kgsl.h.

[21] Google Inc. 2021. Play Protect | Google Developers. https://developers.google.com/android/play-protect.

[22] Google Inc. 2021. Privilege escalation in Google Android. https://source.android.com/security/bulletin/pixel/2021-01-01.

[23] Google Inc. 2021. Reduce overdraw. https://developer.android.com/topic/performance/rendering/overdraw.

[24] Qualcomm Technologies Inc. 2021. Qualcomm Adreno GPU Overview. https://developer.qualcomm.com/docs/adreno-gpu/developer-guide/gpu/overview.html.

[25] Akanksha Jain and Calvin Lin. 2019. Cache Replacement Policies. *Synthesis Lectures on Computer Architecture* 14, 1 (2019), 1–87. https://doi.org/10.2200/S00922ED1V01Y201905CAC047

[26] Abdul Rehman Javed, Mirza Omer Beg, Muhammad Asim, Thar Baker, and Ali Hilal Al-Bayatti. 2020. AlphaLogger: Detecting motion-based side-channel attack using smartphone keystrokes. *Journal of Ambient Intelligence and Humanized Computing* (2020), 1–14. https://doi.org/10.1007/s12652-020-01770-0

[27] Elmira Karimi, Zhen Hang Jiang, Yunsi Fei, and David Kaeli. 2018. A timing side-channel attack on a mobile gpu. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*. IEEE, 67–74. https://doi.org/10.1109/ICCD.2018.00020

[28] Jari Komppa. 2009. QCOM_performance_monitor_global_mode. https://www.khronos.org/registry/OpenGL/extensions/QCOM/QCOM_performance_monitor_global_mode.txt.

[29] Sangchul Lee and Jae Wook Jeon. 2010. Evaluating performance of Android platform using native C for embedded systems. In *ICCAS 2010*. IEEE, 1160–1163. https://doi.org/10.1109/ICCAS.2010.5669738

[30] Sangho Lee, Youngsok Kim, Jangwoo Kim, and Jong Kim. 2014. Stealing webpages rendered on your browser by exploiting GPU vulnerabilities. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 19–33. https://doi.org/10.1109/SP.2014.9

[31] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. 2016. ARMageddon: Cache Attacks on Mobile Devices. In *Proceedings of the 25th USENIX Conference on Security Symposium (SEC'16)*. 549–564.

[32] Chao Luo, Yunsi Fei, and David Kaeli. 2019. Side-channel Timing Attack of RSA on a GPU. *ACM Transactions on Architecture and Code Optimization (TACO)* 16, 3 (2019), 1–18. https://doi.org/10.1145/3341729

[33] Anindya Maiti, Murtuza Jadliwala, Jibo He, and Igor Bilogrevic. 2015. (Smart) watch your taps: Side-channel keystroke inference attacks using smartwatches. In *Proceedings of the 2015 ACM International Symposium on Wearable Computers*. 27–30. https://doi.org/10.1145/2802083.2808397

[34] Yan Michalevsky, Aaron Schulman, Gunaa Arumugam Veerapandian, Dan Boneh, and Gabi Nakibly. 2015. Powerspy: Location tracking using mobile device power analysis. In *24th USENIX Security Symposium*. 785–800.

[35] Emiliano Miluzzo, Alexander Varshavsky, Suhrid Balakrishnan, and Romit Roy Choudhury. 2012. Tapprints: your finger taps have fingerprints. In *Proceedings of the 10th international conference on Mobile systems, applications, and services (MobiSys '12)*. 323–336. https://doi.org/10.1145/2307636.2307666

[36] Elizabeth Montalbano. 2020. Facebook Messenger Bug Allows Spying on Android Users. https://threatpost.com/facebook-messenger-bug-spying-android/161435/.

[37] Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael Abu-Ghazaleh. 2018. Rendered insecure: GPU side channel attacks are practical. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. 2139–2153. https://doi.org/10.1145/3243734.3243831

[38] Lucky Onwuzurike and Emiliano De Cristofaro. 2015. Danger is my middle name: experimenting with SSL vulnerabilities in Android apps. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*. 1–6. https://doi.org/10.1145/2766498.2766522

[39] Emmanuel Owusu, Jun Han, Sauvik Das, Adrian Perrig, and Joy Zhang. 2012. Accessory: password inference using accelerometers on smartphones. In *Proceedings of the twelfth workshop on mobile computing systems & applications (HotMobile '12)*. 1–6. https://doi.org/10.1145/2162081.2162095

[40] Sarah Pearman, Shikun Aerin Zhang, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. 2019. Why people (don't) use password managers effectively. In *The 15th Symposium on Usable Privacy and Security*. 319–338.

[41] Bahman Rashidi and Carol J Fung. 2015. A Survey of Android Security Threats and Defenses. *J. Wirel. Mob. Networks Ubiquitous Comput. Dependable Appl.* 6, 3 (2015), 3–35. https://doi.org/10.22667/JOWUA.2015.09.31.003

[42] Jose Ribeiro, Firooz B Saghezchi, Georgios Mantas, Jonathan Rodriguez, and Raed A Abd-Alhameed. 2020. Hidroid: prototyping a behavioral host-based intrusion detection and prevention system for android. *IEEE Access* 8 (2020), 23154–23168. https://doi.org/10.1109/ACCESS.2020.2969626

[43] Jong-hyuk Roh, Sung-Hun Lee, and Soohyung Kim. 2016. Keystroke dynamics for authentication in smartphone. In *2016 International Conference on Information and Communication Technology Convergence (ICTC)*. IEEE, 1155–1159. https://doi.org/10.1109/ICTC.2016.7763394

[44] Martin Peres Samuel Pitoiset. 2014. Expose NVIDIA's performance counters to the userspace for NV50/Tesla. https://www.x.org/wiki/Events/XDC2014/XDC2014PitoisetNouveau/talk-perf.pdf.

[45] Andrea Saracino, Daniele Sgandurra, Gianluca Dini, and Fabio Martinelli. 2016. Madam: Effective and efficient behavior-based android malware detection and prevention. *IEEE Transactions on Dependable and Secure Computing* 15, 1 (2016), 83–97. https://doi.org/10.1109/TDSC.2016.2536605

[46] Asaf Shabtai, Uri Kanonov, Yuval Elovici, Chanan Glezer, and Yael Weiss. 2012. "Andromaly": a behavioral malware detection framework for android devices. *Journal of Intelligent Information Systems* 38, 1 (2012), 161–190. https://doi.org/10.1007/s10844-010-0148-x

[47] Yuru Shao, Xiapu Luo, and Chenxiong Qian. 2014. Rootguard: Protecting rooted android phones. *Computer* 47, 6 (2014), 32–40. https://doi.org/10.1109/MC.2014.163

[48] Ki-Cheol Son and Jong-Yeol Lee. 2011. The method of android application speed up by using NDK. In *2011 3rd International Conference on Awareness Science and Technology (iCAST)*. IEEE, 382–385. https://doi.org/10.1109/ICAwST.2011.6163104

[49] Raphael Spreitzer, Felix Kirchengast, Daniel Gruss, and Stefan Mangard. 2018. Procharvester: Fully automated analysis of procfs side-channel leaks on android. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security (ASIACCS '18)*. 749–763. https://doi.org/10.1145/3196494.3196510

[50] Raphael Spreitzer, Gerald Palfinger, and Stefan Mangard. 2018. Scandroid: Automated side-channel analysis of android apis. In *Proceedings of the 11th ACM Conference on Security & Privacy in Wireless and Mobile Networks (WiSec '18)*. 224–235. https://doi.org/10.1145/3212480.3212506

[51] Elizabeth Stobert and Robert Biddle. 2015. Expert password management. In *International Conference on Passwords*. Springer, 3–20. https://doi.org/10.1007/978-3-319-29938-9_1

[52] Jeff Vander Stoep. 2015. ioctl command whitelisting in SELinux. http://kernsec.org/files/lss2015/vanderstoep.pdf.

[53] Daimeng Wang, Ajaya Neupane, Zhiyun Qian, Nael B Abu-Ghazaleh, Srikanth V Krishnamurthy, Edward JM Colbert, and Paul Yu. 2019. Unveiling your keystrokes: A Cache-based Side-channel Attack on Graphics Libraries. In *NDSS*.

[54] Davey Winder. 2020. Qualcomm Snapdragon Bugs Leave 40% Of World's Smartphones Exposed To Spying Threat. https://www.forbes.com/sites/daveywinder/2020/08/06/hundreds-of-millions-of-android-phones-can-spy-on-users-as-400-snapdragon-security-flaws-confirmed-qualcomm-google-lg-samsung-oneplus/.

[55] Dmitrijs Zaparanuks, Milan Jovic, and Matthias Hauswirth. 2009. Accuracy of performance counter measurements. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE, 23–32. https://doi.org/10.1109/ISPASS.2009.4919635

[56] Hang Zhang, Dongdong She, and Zhiyun Qian. 2015. Android root and its providers: A double-edged sword. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 1093–1104. https://doi.org/10.1145/2810103.2813714

[57] Xiaoyong Zhou, Soteris Demetriou, Dongjing He, Muhammad Naveed, Xiaorui Pan, XiaoFeng Wang, Carl A Gunter, and Klara Nahrstedt. 2013. Identity, location, disease and more: Inferring your secrets from android public resources. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 1017–1028. https://doi.org/10.1145/2508859.2516661