Power-Related Side-Channel Attacks using the Android Sensor Framework

Mathias Oberhuber, Martin Unterguggenberger, Lukas Maar, Andreas Kogler, Stefan Mangard

Graz University of Technology: {firstname.lastname}@iaik.tugraz.at

Abstract—Software-based power side-channel attacks are a significant security threat to modern computer systems, enabling adversaries to extract confidential information. Existing attacks typically exploit direct power signals from dedicated interfaces, as demonstrated in the PLATYPUS attack, or power-dependent timing variations, as in the case of the Hertzbleed attack. As access to direct power signals is meanwhile restricted on more and more platforms, an important question is whether other exploitable power-related signals exist beyond timing proxies.

In this paper, we show that Android mobile devices expose numerous power-related signals that allow power side-channel attacks. We systematically analyze unprivileged sensors provided by the Android sensor framework on multiple devices and show that these sensors expose parasitic influences of the power consumption. Our results include new insights into Android sensor leakage, particularly a novel leakage primitive: the rotation-dependent power leakage of the geomagnetic rotation vector sensor. We extensively evaluate the exposed sensors for different information leakage types. We compare them with the corresponding ground truth, achieving correlations greater than 0.9 for some of our tested sensors. In extreme cases, we observe not only statistical results but also, e.g., changes in a compass app's needle by approximately $30°$ due to CPU stress. Additionally, we evaluate the capabilities of our identified leakage primitives in two case studies: As a remote attacker via the Google Chrome web browser and as a local attacker running inside an installed app. In particular, we present an end-to-end pixel-stealing attack on different Android devices that effectively circumvents the browser's cross-origin isolation with a leakage rate of 5-$10\,\text{s}$ per pixel. Lastly, we demonstrate a proof-of-concept AES attack, leaking individual key bytes using our newly discovered leakage primitive.

I. INTRODUCTION

One of the growing concerns in system security is the rise of software-based power side-channel attacks [1]–[9]. These attacks exploit information leakage through variations in power consumption due to the underlying CMOS hardware and pose a significant threat to the confidentiality and integrity of sensitive data. Traditional power analysis techniques [10]–[14] typically require external measurement equipment and were mostly practical on embedded devices. However, several traditional power analysis attacks using physical measuring equipment have also targeted mobile devices [15]–[17]. The PLATYPUS attack [3] removes the limitation for external measurement equipment by using a CPU internal interface to measure the power consumption and demonstrates that traditional power analysis attacks are practical using software-based interfaces. However, attacks using such direct power interfaces were easily mitigated by restricting access to the interface [3], [18]. The Hertzbleed attack [6] and Liu et al. [5] show how to circumvent interface restrictions by using power-dependent timing variations originating from the dynamic power management of modern CPUs. We refer to such indirect power signals as *power-related* signals that are strongly correlated to the power consumption of the device.

So far, software-based power analysis attacks exploiting the power consumption directly or via power-related signals have mainly been done on desktop and server CPUs. There have also been initial works on mobile phones. For instance, multiple attacks have used the direct power interface before Android 7 to steal password lengths, user locations, as well as app fingerprinting [19]–[21]. Furthermore, Taneja et al. [22] demonstrate a pixel-stealing attack, leaking pixels from an *inline frame* (iframe), violating the cross-origin isolation of Google Chrome by using a timing-based power-related signal.

Although the direct energy interface on Android mobile systems is meanwhile restricted, the Android sensor framework [23] still exposes sensor signals to unprivileged apps. This framework sparked numerous *non-power-oriented* side-channel attacks [24] focusing, e.g., on motion sensors or the magnetometer to localize devices [25] or infer user inputs [26]–[28]. While initial research has demonstrated a relation between CPU utilization and sensor measurements by the magnetometer [29], it remains unclear whether this relation has a similar potential for exploitation as Hertzbleed [5], [6]. Given that Android exposes this framework to unprivileged apps and web browsers, such an unaddressed threat could pose a significant risk to millions of Android devices.

In this paper, we present a power-related side-channel attack using the Android sensor framework that can be exploited by remote and local attackers. Initially, we systematically analyze 9 commodity Android devices with overall 137 integrated sensors accessible from unprivileged applications through the Android sensor framework, identifying novel power-related signals. We then perform a leakage analysis that categorizes and quantifies sensors based on three distinct leakage types: leaking CPU utilization levels (i.e., high and low CPU load), comparing different instruction sequences, and leaking data operands. These categories identify the potential risk and attack vectors a given sensor exposes, independent of the

actual root cause of the power-related signal. We perform a correlation-based analysis between the sensor readings and a ground truth signal to quantify the leakage strength for each category. Our results indicate that the performance of the sensors varies between these categories. For instance, while the magnetometer of our *Google Pixel 6a* shows a high correlation of 0.973 with CPU utilization, it only shows a medium correlation of 0.395 for instruction-dependent leakage, i.e., correlation of supply voltage and sensor readings. However, we observe the inverse effect for a pressure-based sensor where the instruction-dependent leakage has a high correlation of 0.955 over a medium correlation with CPU utilization of 0.472. In extreme cases, we also observe visual effects besides correlation, e.g., in the deflection of the needle of a compass app by around 30° due to CPU stress. Finally, the *geomagnetic rotation vector* sensor exposes our most prominent power-related signal. We provide a detailed leakage analysis for this sensor, highlighting characteristics such as orientation-dependent leakage and the internal workings of the sensor.

We demonstrate the capabilities of our identified leakage primitives in two case studies: as a remote attacker in the web browser and as a local attacker operating inside an installed app. For the remote setting, we present an end-to-end pixel-stealing attack on different devices that effectively bypasses the cross-origin isolation implemented by a modern web browser, e.g., Google Chrome. In this attack, we leverage sensor-based leakage from the JavaScript sensor framework, accessible as a remote attacker, to leak pixels at leakage rates of 5-$10\,\mathrm{s}$ per pixel, thereby breaking cross-origin isolation. For the local setting, we use the geomagnetic rotation vector sensor to analyze the data-dependent leakage of ARM NEON's side-channel resistant hardware AES implementation. We then provide a proof-of-concept CPA attack to recover AES key bytes within $300\,000$ samples. Our findings conclude that the Android sensor framework exposes multiple sensors that can act as power-related signals due to significant parasitic influences of actual physical properties on the sensor signal on modern mobile devices.

Contributions. In summary, our key contributions are:

1) **Systematic analysis of Android sensors.** We demonstrate through a systematic analysis of 9 recent Android smartphones that general-purpose sensors, accessible without special permissions, capture parasitic physical influences, elevating them to power-related signals.
2) **Analysis of the geomagnetic rotation vector sensor.** We are the first to provide an in-depth analysis of the power-leakage of the geomagnetic rotation vector sensor.
3) **An AES CPA attacks using geomagnetic rotation.** We provide a proof-of-concept leakage analysis of ARM NEON's AES implementation and results of a CPA attack. While full key recovery was not possible, individual keybytes were recovered using $300\,000$ samples.
4) **A remote sensor-based pixel stealing attack.** We provide an end-to-end pixel-stealing attack on Android exploiting the generic sensor framework accessible from JavaScript in the Google Chrome web browser.

Outline. The paper is structured as follows. Section II provides background of this work. Section III provides the systematic sensor leakage analysis. Section IV introduces our novel leakage primitive based on the geomagnetic rotation vector sensor. Section V presents a remote sensor-based pixel-stealing attack. Section VI presents a CPA attack on AES. Section VII discusses related work, mitigations, and limitations. Finally, Section VIII concludes this work.

Responsible Disclosure. We disclosed our findings to Google in February 2024. As a response, Google added additional rounding to the Magnetometer sensor interface of the Google Chrome browser [30]. Additionally, Google is planning to add an additional permission prompt to access the orientation sensor of the Google Chrome browser.

II. BACKGROUND

This section provides the required background on side-channel attacks and the Android sensor framework.

A. Power Side-Channel Attacks

Side-channel attacks are a passive type of attack, exploiting information leakage of a device during data processing.

These attacks can be mounted using software-based interfaces or physical measurement equipment, allowing an adversary to measure the leaked side-channel information. Common sources of side-channel leakages include timing behavior, electromagnetic radiation [31]–[33], and power consumption [10], [11], [34] of the device. The obtained side-channel traces are analyzed to gain insights about the processed secret.

Simple Power Analysis (SPA) [11] exemplifies the concept of side-channel attacks. SPA reconstructs the executed CPU operations based on the processor's power consumption during execution. SPA uses power traces and attempts to match them against the processor's internal computations. This concept of information leakage is extended by using statistical methods. For instance, Differential Power Analysis (DPA) [11] reveals secrets by analyzing differences in power traces while the device is computing on varying data with a constant secret. Furthermore, Correlation Power Analysis (CPA) [10] is a special case of DPA that uses correlation-based approaches to reveal cryptographic secrets.

Recent discoveries highlight that software interfaces reporting power measurements enable power analysis attacks without physical access to the device. Lipp et al. [3] showcase how to extract cryptographic keys from enclaves and how to break Kernel Address Space Layout Randomization (KASLR) using Intel's Running Average Power Limit (RAPL) [35] interface. Consequently, the unprivileged access to the RAPL interface was removed [18]. Wang et al. [6], [7] demonstrate that the adaptive power management of modern CPUs influences the CPU frequency based on the system's power consumption. Thus, variations in power consumption cause frequency throttling to comply with the system's power constraints. Wang et al. [6] show that frequency throttling-induced execution time variations can be used as a power-related signal, stealing cryptographic keys, even as a remote attacker.

TABLE I: The CPU hardware specifications of all the tested Android devices used in our systematic analysis.

Model	CPU	Architecture Perf. Core	Cores	Perf. Cores	#P-States	Frequency [GHz]
Google Pixel 6a	Google Tensor	ARM Cortex-X1	8	2	16	500 - 2802
Samsung Galaxy S20 FE	Samsung Exynos 990	Samsung Exynos M5	8	2	22	442 - 2730
Samsung Galaxy A51	Samsung Exynos 9611	ARM Cortex-A73	8	2	15	403 - 2310
Samasung Galaxy S9	Samsung Exynos 9810	Samsung Exynos M3	8	4	16	455 - 2700
One Plus 5T	Qualcom Snapdragon 835	Kryo 280	8	4	31	300 - 2460
Honor View 20	HiSilicon Kirin 980	ARM Cortex-A55	8	4	13	826 - 2600
Honor P90 Lite	Mediatek Dimensity 6020	Cortex-A76	8	2	16	725 - 2203
Google Pixel 7a	Google Tensor G2	ARM Cortex-X1	8	2	18	500 - 2850
Google Pixel 9	Google Tensor G4	ARM Cortex-X4	8	1	18	700 - 3105

Liu et al. [5] show that frequency and timing variations directly relate to power, extracting cryptographic AES keys. Taneja et al. [22] demonstrate the effect of frequency, power, and temperature variations on ARM mobile devices and GPUs, demonstrating a pixel-stealing attack from the browser and website fingerprinting. Kogler et al. [2] present a generic CPA attack from software on the shared memory subsystem of a CPU, leaking arbitrary data from the kernel and processes.

B. Android Sensor Framework

Android mobile devices come equipped with various sensors [23], providing data on the device's movement, orientation, and environment, usable by Android applications. These sensors are categorized into two main types: hardware- and software-based sensors. Hardware-based sensors directly collect data from dedicated physical hardware to report environmental information, while software-based sensors utilize one or several hardware sensors to emulate real hardware.

Unprivileged applications can access these integrated sensors through the Android sensor framework, providing an interface for configuring and receiving sensor data. Each sensor adheres to specific constraints, such as the resolution for reading values and the minimum and maximum data acquisition rates. An Android application can either set up a callback function that triggers when new sensor values are available or actively poll for the next sensor event. The reporting intervals vary depending on the sensor types. Streaming sensors provide values at regular intervals, while non-streaming sensors only update values when changes in the measured parameter occur.

In addition, the maximum refresh rate of the sensors depends on their initial configuration. Starting from Android 12, an application needs to declare the `HIGH_SAMPLING_RATE_SENSORS` permission in the applications' manifest to leverage arbitrary small sensor delays. Otherwise, the maximum sensor refresh is capped at 200 Hz. Requesting refresh rates larger than 200 Hz is regarded as a warning and may lead to rejection from a listing in the Google Play Store [36]. Notably, the specified update rates serve only as a recommendation and are not obligatory for the system to adhere to, as each sensor imposes a maximal update rate. Additionally, the system will not disable sensors when the screen is turned off [23]. Furthermore, the application must either be in the foreground or implement
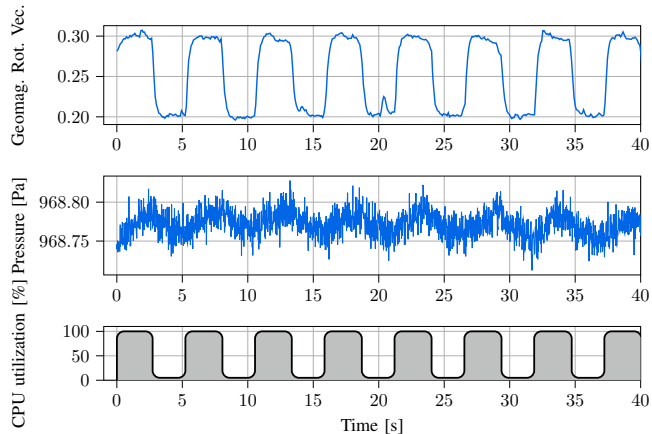

Fig. 1: Sensor values of the *geomagnetic rotation vector* sensor and the *pressure* sensor of the Google Pixel 6a, captured over a time period of 40 s while concurrently varying the CPU utilization by creating CPU stress on all but one CPU core alternated with sleep periods of roughly 2.5 s each.

a foreground service to capture sensor values. A subset of the sensors from the native Android sensor framework is accessible from JavaScript in the web browser [37]. The web browser adds additional abstraction, like filtering and reducing the refresh rates [38].

III. SYSTEMATIC ANALYSIS OF SENSOR LEAKAGE

This section provides a systematic analysis of sensor leakage on Android mobile devices. We demonstrate that Android sensors, accessible by unprivileged applications [23], can be exploited to obtain side-channel leakage of concurrent system operations. Our systematic analysis includes all available streaming sensors of various evaluated Android devices accessible to unprivileged applications through the Android sensor framework. We quantify the leakage of each sensor for CPU utilization (Section III-A), instruction-based leakage (Section III-B), and data-dependent leakage (Section III-C). This analysis identifies potential sensors to mount more powerful side-channel attacks.

Experimental Setup. We analyze 9 Android mobile devices featuring recent ARM CPUs. Table I highlights the CPU

TABLE II: Results of the systematic leakage analysis of all streaming sensors of the Android sensor framework on the Google Pixel 6a. The table includes the details for each evaluated sensor, whether it contains electromagnetic (EM) readings, its resolution (indicating the smallest reported sensor increment) and update rate (fastest time interval between consecutive sensor values). We provide an absolute correlation coefficient r in three categories: Varying CPU Utilization, Varying Instruction Sequences and Varying Data. We also include a Significance Factor ($|SF|$) for varying data (defined in Section III-C).

Sensor	EM	Resolution	Update Rate [ms]	Varying CPU Utilization r	Varying Instruction Sequences r	Varying Data r	Varying Data $|SF|$
LSM6DSR Accelerometer	✗	$4.78 \cdot 10^{-3}$	5	0.476	0.132	-	-
MMC56X3X Magnetometer	✓	$9.76 \cdot 10^{-2}$	10	**0.972**	0.298	0.008	1.16
Orientation Sensor	✓	$1.00 \cdot 10^{-5}$	5	0.362	0.614	0.018	5.55
LSM6DSR Gyroscope	✗	$1.22 \cdot 10^{-3}$	5	0.399	-	0.016	2.18
ICP10101 Pressure Sensor	✗	$1.00 \cdot 10^{-4}$	20	0.472	**0.955**	-	-
Gravity Sensor	✗	$1.00 \cdot 10^{-5}$	5	0.509	0.656	0.014	4.55
Linear Acceleration Sensor	✗	$1.00 \cdot 10^{-5}$	20	0.258	0.032	-	-
Rotation Vector Sensor	✓	$1.00 \cdot 10^{-5}$	5	0.460	0.145	0.006	1.80
MMC56X3X Magnetometer-Uncalibrated	✓	$9.76 \cdot 10^{-2}$	10	**0.973**	0.395	0.018	3.58
Game Rotation Vector Sensor	✗	$1.00 \cdot 10^{-5}$	5	0.426	**0.809**	0.009	2.64
LSM6DSR Gyroscope-Uncalibrated	✗	$1.22 \cdot 10^{-3}$	5	0.397	0.248	0.011	2.06
Geomagnetic Rotation Vector Sensor	✓	$1.00 \cdot 10^{-5}$	5	**0.932**	**0.913**	0.040	**12.77**
LSM6DSR Accelerometer-Uncalibrated	✗	$4.79 \cdot 10^{-3}$	5	0.327	0.152	0.010	1.53
LSM6DSR Temperature	✗	$3.91 \cdot 10^{-3}$	9.23	-	-	-	-
ICP10101 Temperature	✗	$1.00 \cdot 10^{-2}$	20	-	-	-	-
Auto Brightness	✗	$1.00 \cdot 10^{-3}$	1000	-	-	-	-

hardware specifications of the tested Android devices, including high-end devices from Samsung, Google, OnePlus, and Honor. All devices run the latest available Android version at the time of writing, ranging from Android 10 (Honor View 20, Samsung Galaxy S9), Android 13, to Android 14 (Google Pixel 9). Unless stated otherwise, all evaluated devices maintain their default configurations without any modifications. Complementary information, like the ground truth value of the system voltage and CPU frequency, is measured on rooted Android devices. All experiments run as ordinary Android applications without requiring additional permissions. The experimental results are captured while the devices are connected to a power source. However, we did not observe a significant difference between the results of the experiments on battery power. Additionally, we do not pin the CPU frequency and keep Dynamic Voltage and Frequency Scaling (DVFS) enabled. This allows dynamic adjustments to the system's voltage and operating frequency to adhere to thermal and power limits.

A. Distinguishing CPU Utilization

As an initial step for our analysis, we demonstrate that concurrent CPU utilization influences the sensor values of the Android sensor framework. We alternate between heavy CPU workload and idle periods to vary CPU utilization, with each phase lasting $2.5\,\mathrm{s}$. We use these time intervals as they provide good results while maintaining a reasonable evaluation time. We conduct multiple evaluations to capture measurements (i.e., sensor values) from all available sensors at their maximum refresh rate. To stress the CPU, we utilize a benchmark from the `stress-ng` [39] suite, specifically designed to generate heavy CPU loads by executing multiple arithmetic instructions. Additionally, we record the CPU temperature, supply voltage, and current. Since access to voltage, current, and CPU temper-

ature requires root privileges, we further validate that rooting the device does not affect the sensor interface by repeating the experiments on unrooted devices.

Figure 1 shows the results of our experiment conducted on the Google Pixel 6a Android device. The graphs illustrate varying sensor values throughout the experiment, alongside the CPU utilization during phases of CPU stress and CPU idle, respectively. The sensor values demonstrate a correlation with CPU utilization, indicating a recognizable trend that aligns with the pattern of CPU utilization. Notably, the geomagnetic rotation vector in Figure 1 shows fast transitions from low to high and vice versa. Similarly, the pressure sensor of Figure 1 changes based on the system state but at a slower rate. Additionally, the pressure sensor signal is more noisy than the geomagnetic rotation vector sensor.

To systematically quantify the extent of information leakage, we establish a ground truth model $\mathcal{M} = \{1, 0\}$, representing high and low CPU utilization, respectively. Consequently, we calculate the Pearson correlation coefficient r between the captured sensor values and the ground truth model $\mathcal{M}$, quantifying the strength of the linear relationship. For sensors with notable noise, like the pressure sensor, we remove high frequent noise using a low pass filter. Additionally, many sensors collect measurements via multiple channels, e.g., the orientation sensor captures the orientation along the x, y, and z axes. In our evaluation, we consider all channels from each sensor and detail the result of the best channel per sensor. Table II lists the results of this experiment for each sensor of the Google Pixel 6a. The table includes the details for each evaluated sensor, such as its value resolution (indicating the smallest reported sensor increment) and update rate (the fastest time interval between consecutive sensor values). We list the gathered correlation coefficients of this experiment, quantifying the relation between the sensor values and the
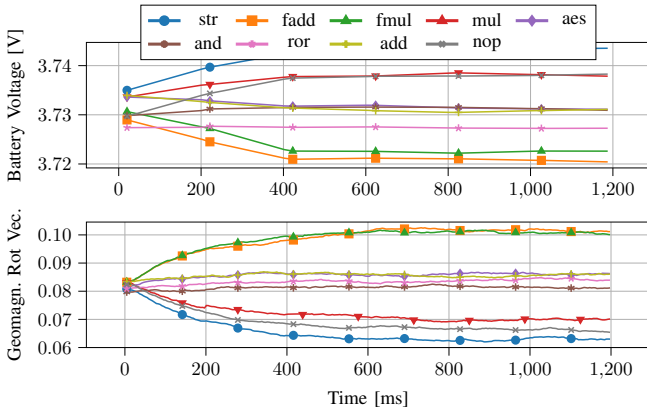
Fig. 2: Averaged battery voltage readings and sensor readings of the *geomagnetic rotation vector* sensor over $1.2\,$s, while executing different instructions on the Google Pixel 6a.

varying CPU utilization in the column *Varying CPU Utilization*. Appendix A provides detailed experiment results, including experimental results from every evaluated smartphone. We only list statistically significant correlations. Our results show that 3 sensors of the Google Pixel 6a and 26 sensors overall are susceptible to variations in CPU utilization (i.e., show $r > 0.7$). Sensors that include magnetometer data, like the magnetometer or geomagnetic rotation vector, demonstrate the best results. Sensors that do not include electromagnetic measurements, such as the pressure sensor, accelerometer, gravity sensor, and game rotation vector, are also susceptible to side-channel leakage of concurrent CPU utilization.

> Our analysis shows that $18.9\,\%$ of all sensors expose a significant ($r > 0.7$) influence of the CPU utilization in the actual sensor measurement.

B. Distinguishing Instructions

This section presents a detailed analysis of side-channel leakage beyond CPU utilization. Specifically, we demonstrate that individual instructions executed on ARM mobile CPUs affect the sensor readings of the Android sensor framework. Additionally, we showcase that the unprivileged sensor readings correlate to the phone's battery voltage, which corresponds to the internal supply voltage, thus indicating that unprivileged sensor readings serve as a proxy for measuring battery voltage.

In this experiment, we collect sensor readings of each sensor of the Android sensor framework at the highest data acquisition rate while concurrently executing selected instructions of the ARM hardware platform in a loop. As a basis for the instruction selection, we use, similar to Taneja et al. [22], the survey from Oscar Lab [40], which quantifies and categorizes the most frequently used x86 instructions. We select ARM equivalents from these most commonly used instruction categories for our analysis. The selected instructions include data store instructions (`str`) from a register to a specified memory address, bit operations performed on registers (`ror`, `and`), and arithmetic addition and multiplication (`add`, `mul`). Besides

integer arithmetics, we execute floating point addition and multiplication (`fadd`, `fmul`). We repeatedly choose random instructions from this set, executing them in a loop for $4\,$s each. We chose this longer interval to account for the voltage interface's slower update rate of $175\,$ms. In a second thread, running in parallel, we query sensor readings at the highest data acquisition rate from the Android sensor framework [23]. Additionally, we query the battery voltage at regular intervals of $200\,$ms, providing a ground truth for evaluation. We gather the battery voltage from the `sysfs` interface[1], which provides an updated voltage reading roughly every $175\,$ms. Note that this voltage interface is only accessible by privileged users, thus requiring root access for this experiment. We execute the experiment for each sensor for $2\,$h, running at battery power.

Figure 2 illustrates the sensor measurements from the geomagnetic rotation vector sensor and the battery voltage readings on the Google Pixel 6a. The geomagnetic rotation vector's measurements per instruction reveal three groups ranging from 0.06 to 0.10. The `str`, `mul`, and `nop` instructions form the lowest value group, while the `and`, `ror`, `add` and `aes` instructions occupy the middle-value range. The floating point instructions `fadd` and `fmul` occupy the highest sensor values. Notably, the voltage readings in Figure 2 show similar but inversely proportional distributions for the instructions.

To quantify the observed relationship between the voltage and sensor readings, we calculate the absolute correlation coefficient r between the sensor measurements and the recorded battery voltage. Since the `sysfs` voltage interface reports measurements at a much lower rate than the sensor framework, we linearly interpolate the voltage readings before calculating the correlation coefficients. The results in Table II, demonstrate that 2 of the sensors on the Google Pixel 6a correlate with the battery voltage. Especially the geomagnetic rotation vector and the pressure sensor demonstrate very large correlations ($r > 0.90$). Furthermore, the game rotation vector, the orientation sensor, and the gravity sensor also show a large correlation, with the remaining sensors demonstrating lower correlation coefficients while still showing statistical significance. Note that we observe significant correlations between the voltage readings and the sensor values independent of whether the sensor contains magnetometer data.

> Overall, $12.5\,\%$ of sensors of the Google Pixel 6a correlate significantly ($r > 0.90$) to the battery voltage independent if they include magnetometer readings.

C. Distinguishing Data-Operands

As a next step of our analysis, we demonstrate that sensor readings gathered via the Android sensor framework show data-dependent leakage. We demonstrate this effect by repeatedly executing a constant-cycle instruction [41], processing different data operands in a loop. In parallel, we collect sensor readings at each sensor's highest data acquisition rate. To quantify the observed interference of the processed data

---

[1]Sysfs: `/sys/class/power_supply/battery/voltage_now`
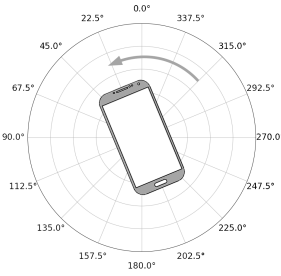
(a) Schematic experimental setup rotating the phone on a horizontal plane in steps of $22.5°$ while creating CPU pressure concurrent to reading sensor values.
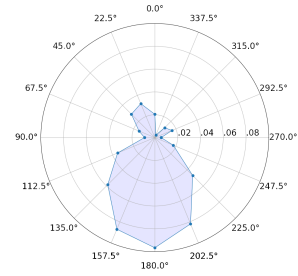
(b) The amplitude of sensor value deviations of the *geomagnetic rotation vector sensor* between CPU stress and sleep periods rotating the phone $360°$ in steps of $22.5°$.

Fig. 3: Schematic experimental setup and analysis, evaluating which phone orientation preserves the best leakage.

operands with the measurements, we correlate the hamming distance of the processed operands as a power model with the gathered measurements.

The experiment involves an application executing a constant-cycle instruction, i.e., `xor` (`eor`) repeatedly in a loop. We utilize a constant-cycle instruction, removing data-dependent control flow variations. The used `xor` instruction operates exclusively on registers, taking two 64-bit input registers as input, writing the result back to an output register. We use registers instead of reading operands directly from memory to minimize side effects, e.g., power influences of caches [2]. Before starting the experiment, we pre-define random, pairwise data operands $\mathcal{G}$ and $\mathcal{V}$, which are used as input to the experiment. We pick a random number between 0 and 7 for each data operand, and encode it in each hex digit of that 64-bit input operand. This concatenates 16 hex digits of the same value in the input register. We use this technique, similar to Kogler et al. [2], to amplify the leakage by a factor of 16, speeding up the trace collection process of the experiment. In the experiment, we use these value pairs $\mathcal{G}$ and $\mathcal{V}$, as input to the repeated `xor` instructions for a duration of approximately $0.5\,\mathrm{s}$. We collect measurements for each sensor for two hours while performing this experiment.

In the subsequent evaluation, we quantify the observed leakage of the measurements by determining the Pearson correlation coefficient r between the gathered measurements of each sensor and the power model of the processed data operands. We use the Hamming distance between the two input operands $\mathcal{G}$ and $\mathcal{V}$ of the `xor` instruction as a power model, i.e., the number of differing bits between the 64-bit registers. We use this model as it usually reflects the power consumption of `xor` instruction. We list the correlation coefficients of each sensor of the Google Pixel 6a in Table II, only considering statistically significant [34] entries.

Due to the varying data acquisition rates of each sensor, which lie between $5\,\mathrm{ms}$ and $1\,\mathrm{s}$, the number of gathered sensor values N of each sensor in the experiment varies. To evaluate the statistical significance of the collected traces, we calculate the *noise level* for a given number of traces N, which is defined as $r_{noise} = \frac{4}{\sqrt{N}}$ [34]. We only consider correlations $r \geq r_{noise}$ as statistically significant and report the *significance factor*

$|SF| = r/r_{noise}$ in our evaluation. We list the results of the experiment of the Google Pixel 6a in Table II, additionally providing results of all evaluated smartphones in Appendix A.

Our evaluations demonstrate that 60 of 137 (43.79%) evaluated sensors pick up parasitic coupling of concurrently processed data operands, as multiple sensors show statistically significant correlations with the power model. The observed leakage varies across different sensors, with sensors containing magnetometer readings like the *geomagnetic rotation vector*, magnetometer, and orientation sensor showing the best leakage properties. Notably, the *geomagnetic rotation vector* sensor demonstrates the highest leakage while running at the highest sampling rate of $5\,\mathrm{ms}$. Additionally, sensors that do not contain magnetometer readings, like the *gyroscope* and *game rotation vector*, also show data-dependent leakage but at a lower magnitude.

> We find that $43.79\,\%$ of the sensors leak data-dependent information about concurrent CPU operations. We identify that the geomagnetic rotation vector shows the strongest leakage across our analysis.

D. Leakage Across Different Devices

Our systematic analysis demonstrates that various built-in sensors from different smartphones of multiple vendors leak power-related CPU information. We provide a comprehensive evaluation of multiple phones from different vendors and smartphone generations, demonstrating the applicability of the side channel, also on recent phones (see Table I).

While all evaluated phones of our systematic analysis show power-related leakage (see Appendix A), our analysis also demonstrates that phones across different generations are susceptible to our power-related side channel. For instance, we analyze the Google Pixel 6a (released in July 2022), the Google Pixel 7a (released in May 2023), and the most recent model, the Pixel 9 (released in August 2024). The integrated sensors of these generations differ in their versions and properties. While the Pixel 6a and 7a use sensors with similar (or equal) properties concerning the sensor types and resolutions, specific sensors of the Pixel 9 include increased resolution. Despite the varying CPU chipsets and hardware

sensors, the leakage properties of all three Google Pixel phones are comparable. For instance, sensors containing magnetometers show high correlations regarding CPU utilization, and orientation sensors show a medium correlation of around 0.5. Some results, e.g., of the accelerometer, differ between the devices; while the Pixel 6a and 9 show a medium correlation with CPU utilization, the Pixel 7a shows little correlation concerning this type of sensor. To summarize, while newer phones such as the Google Pixel 9 use different sensors, our side channel still applies to these newer phones.

> Our analysis covers 9 phones across different vendors and generations released between 2017 and 2024, demonstrating comparable trends of sensor-based power-related leakage.
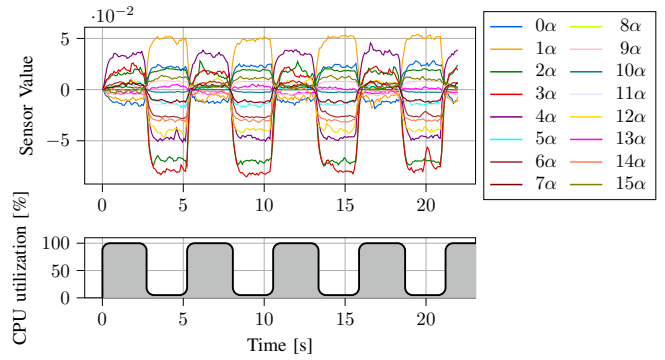

Fig. 4: Sensor readings of the geomagnetic rotation vector sensor, and the varying CPU utilization while rotating the phone in steps of $\alpha = 22.5°$ on a flat surface aligned at a shared starting point at the y-axis of 0.

IV. GEOMAGNETIC ROTATION VECTOR LEAKAGE

In this section, we analyze the properties of the *geomagnetic rotation vector* sensor. This sensor is the best-performing sensor of our leakage analysis (see Section III) for the Google Pixel 6a. Our focus is to obtain a deeper understanding of the leakage properties of the sensor and how these properties influence potential attacks. First, we demonstrate that the phone's spatial orientation influences the magnitude of interference (Section IV-A). Second, we present a novel, generic method to determine if the sensor exposes an integrating signal, i.e., if the sensor accumulates a signal over time. Additionally, we determine the actual start and length of the integration period relative to the reported timestamp of the Android sensor framework (Section IV-B). Our results show that the *geomagnetic rotation vector* is a rotation-dependent and integrating sensor. Subsequently, we use these insights for our AES analysis (see Section VI).

A. Orientation-Dependent Leakage

Our first analysis quantifies the intensity of leakage corresponding to different rotational placements to identify the optimal orientation for potential attacks.

Experimental Setup. To systematically evaluate the effect of rotational orientation on the sensor readings, we initially place the device under test on a flat surface with the display facing up (see Figure 3). This resembles a typical scenario in which the phone is placed, e.g., for charging. We start our experiment by rotating the phone in steps of $22.5°$ for an entire $360°$ rotation, as depicted in the schematic setup in Figure 3a. We define a global reference point at the magnetic North Pole at an angle of $0°$. In each evaluated orientation of the phone across the full $360°$ rotation, we conduct the identical run: We either create computational CPU stress in all but one concurrent CPU threads or leave them idle. The CPU stress phases alternate with sleep periods, each lasting 2.5 s. We collect sensor values in the remaining thread at the maximum data acquisition rate in parallel to this CPU workload. We run this measurement for around 45 s while keeping the phone in a fixed position, ultimately resulting in 16 measurement traces, one for each

orientation. The phone remains connected to a power source via a cable throughout the experiment.

Evaluation. We align the recorded traces in the time and value domain since the *geomagnetic orientation vector* sensor, by design, represents the orientation in the value domain. Therefore, we align each of the 16 measured traces at the start of the experiment by subtracting the first value from the trace. Furthermore, we use the ground truth to align the traces in the time domain. This approach eliminates variations in the absolute sensor values, allowing a straightforward comparison of the relative sensor value deviations. Figure 4 shows the aligned traces. We observe two distinct properties in the traces: First, the traces differ in amplitude, which is indicated by the difference between the plot's extremes. Second, the direction of the interference differs between the orientations.

We observe that these leakage characteristics depend only on the absolute value of the amplitude of the traces. Orientations that expose larger amplitudes are less susceptible to noise and other side effects for our attacks. To quantify the leakage magnitude of each orientation, we compute the average distance between the sensor values of the stress and the idle phase, excluding the transition phases. Depending on the phone's orientation, the resulting averaged amplitudes range between a maximum of 0.097 (± 0.005) and a minimum of 0.002 (± 0.0002). This magnitude analysis signifies a difference in factor 50 between the two extremes. To put this into perspective, in extreme cases, this is equivalent to a deviation of a compass app's needle by roughly $30°$ due to CPU stress.

We visualize the amplitudes for each orientation in Figure 3b to determine which orientation shows the highest magnitude. The plot shows the maximal magnitude around $180°$, corresponding to the magnetic South Pole. The observed amplitude decreases symmetrically as the phone is rotated in both directions, reaching its minimum at a $\pm 90°$ angle to the plane between the North and South Poles. Consequently, the best leakage of the *geomagnetic rotation vector* sensor is achieved while oriented towards the geomagnetic South Pole.
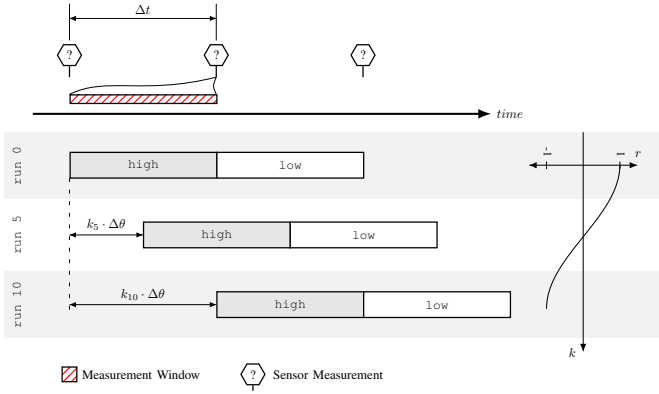
Fig. 5: Experimental setup characterizing a fine granular, integrating behavior of the sensor measurements.

> The leakage of the *geomagnetic rotation vector* sensor is rotation-dependent and has the maximum leakage magnitude when pointing to the geomagnetic South Pole. CPU load can deviate a compass app's needle up to approximately 30°.

B. Analysis of Integrating Behavior

This section presents a generic method characterizing the time window during which CPU operations affect sensor measurements. We determine if and how operations at specific time points affect sensor measurements done in parallel and after executing the CPU operations. In addition, we determine the relative start and end timestamps of the time window affecting a single sensor measurement point.

The Android sensor framework [23] uses Analog-to-Digital Converters (ADCs) to acquire analog signals and provide them as digital representations to software. The datasheets for these converters provide information on the hardware characteristics and configuration options. However, they do not specify how the hardware measurements relate to the reported software sensor readings. For our detailed leakage assessments regarding power-related leakage of CPU operations, the information provided is insufficient and requires further evaluation.

Approach. Analog-to-Digital Converters (ADCs) sample analog signals by observing a specific measurement time window. Changes in an analog signal during that time window are integrated and reported as a sensor value. CPU operations executed concurrently to this measurement time window distort that corresponding sensor value (cf. Section III). Our experimental approach aims to optimize the alignment between the sensor time window and concurrently executed workloads.

Our experiment involves multiple runs, each executing two workloads denoted as high and low. The workload high corresponds to a high power draw, while low represents a low power draw. Each run alternately executes the high and low workloads while simultaneously collecting sensor measurements at the highest data acquisition rate. As depicted in the schematic setup in Figure 5, each workload is executed

for a predetermined time interval Δt. This time interval Δt equals the time between two consecutive sensor measurements.

The runs of the experiment differ in the temporal alignment of the workloads in relation to the sensor measurement. For this alignment, we introduce a varying relative offset $k_i \cdot \Delta\theta$ between the start of each workload and the corresponding sensor measurement. We determine the time interval $\Delta\theta$ by dividing the $2\Delta t$ time frame into 20 equally-sized time intervals, providing fine-grained temporal granularity for the experiment. The experiment consists of 20 runs, each aligning the start of each workload at different relative offsets $k_i \cdot \Delta\theta$ with $k_i \in [0, 19]$ to the sensor measurements.

To evaluate the alignment and overlap of the executed workload with the measurement window in each run, we compute a correlation coefficient r between a power model representing the workloads and the corresponding sensor measurements. Our power model represents $\mathcal{M} = \{0, 1\}$ for the ground truth of the drawn power (low, high). The calculated correlation coefficient reflects the relation between the model and the measured values in an interval of $[1, -1]$, resembling the relationship's magnitude and direction. A high correlation between the power model and a run's measurements thus indicates strong alignment of the measurement window with the workloads at that relative offset of $k_i \cdot \Delta\theta$.

Figure 5 depicts an exemplary sequence of correlations of different runs with varying relative offsets, visualizing the concept of the experiment. Run 0 shows strong alignment of the measurement window and the workloads, thus resulting in a large positive correlation r. Conversely, run 10, at a relative offset of $k_{10} \cdot \theta = \Delta t$, the time interval between two sensor measurements, results in a negative correlation of similar absolute value. At offsets where the measurement window captures both workloads (i.e., $k_x \cdot \theta$), the absolute correlation at that offset $k_i \cdot \theta$ is decreased depending on the contribution of each alternating workload to the time window. **Detailed Experimental Setup.** Our experiment involves executing identical constant-cycle workloads for both high and low phases of Figure 5. While the workload's control flow is identical, they draw varying amounts of power due to the varying processed data operands. The workloads execute the xor (eor) instruction consecutively, processing two 64-bit operands $\mathcal{G}$ and $\mathcal{V}$. Furthermore, the workloads only operate on registers, reducing side effects of memory loads, e.g., caching.

We set both input operands $\mathcal{G}$ and $\mathcal{V}$ to a pre-defined 64-bit value before each invocation of the workload. The first operand $\mathcal{G}$ is always fixed to an initial value of 0, while the second operand $\mathcal{V}$ is either fixed to 0 (no bits set) or 1..1 (all bits set). At each invocation, the workload repeatedly computes the xor-instruction with the pre-defined input operands $\mathcal{G}$ and $\mathcal{V}$, writing the result back to operand $\mathcal{G}$. This results in the source register $\mathcal{G}$ being toggled (high power consumption) or not (low power consumption), depending on the operand $\mathcal{V}$.

Additionally, we determine the exact interval Δt between two consecutive sensor values. Although the Android sensor framework software interface reports a minimum configurable time between two sensor events of $5\,\text{ms}$, we observe a precise
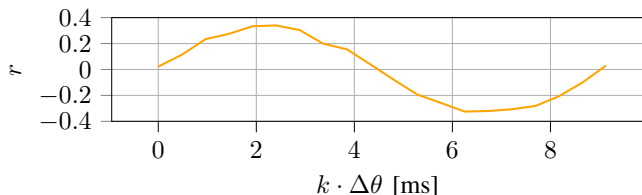
Fig. 6: Correlation r of the sensor values of the geomagnetic rotation vector sensor on the Google Pixel 6a with a model of the shifted workloads at each shift offset throughout $9\,\text{ms}$.
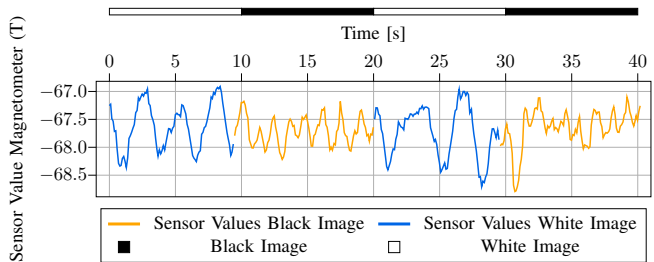

Fig. 7: The measurements of sensor values from the magnetometer using the generic sensor framework. Here, the magnetometer oscillates at different frequencies based on the rendered image color when applying an SVG filter stack.

time interval of $4.571\,\text{ms}$ ($\pm\ 2.86\,\mu\text{s}$) between consecutive sensor events. We determine this interval by collecting $50\,000$ sensor readings, measuring the time difference between them while the phone is idle.

Evaluation. In the evaluation, we calculate the correlation coefficient r between the power model and the collected measurements of each run. Each run includes $200\,000$ sensor measurements used for evaluation. We use the operand $\mathcal{V}$ of the `xor` operation, corresponding to `high` and `low` power draw, as the power model. We display the resulting correlations at each relative offset $k_i \cdot \Delta\theta$ of each run k_i in Figure 6. The plot exhibits a sine curve with a period of $2\Delta t$, starting at an offset $k_0 \cdot \Delta\theta = 0$ with a correlation coefficient of approximately 0.

The plot highlights a maximum correlation at a relative offset of $2.40\,\text{ms}$ to the sensor values, indicating the best alignment between the measurement window and the start of the workloads. Thus, we infer that the sensor's measurement window starts around $2.17\,\text{ms}$ before a corresponding sensor value. Furthermore, Figure 6 displays a gradual transition between the maximum and minimum correlation values, forming a sine curve. Therefore, we infer that the sensor integrates over a substantial measurement window rather than sampling a small time window. The duration between the maximum and minimum correlation values directly indicates the sensor's measurement window size. A shorter measurement window would show faster transitions from the maximum to the minimum with plateaus at the maxima and minima. Figure 6 shows this peak-to-peak duration of Δt, indicating that the sensor measures and integrates over the entire duration of Δt.

V. PIXEL-STEALING ATTACK

This section presents a remote attack scenario where a target Android device visits an attacker-controlled website within a web browser, e.g., Google Chrome. The goal of this scenario is to perform a pixel-stealing attack [42] that violates the isolation enforcement of the web browser, i.e., cross-origin isolation, by using our novel sensor-based side-channel attack. In this context, we refer to a pixel-stealing attack as the capability to leak individual pixel colors rendered in the web browser. In the following, we evaluate different sensor-based *leakage primitives* (cf. Section V-A) accessible from the JavaScript environment, demonstrating the leakage properties of our approach. Subsequently, we present our *end-*

to-end exploit (cf. Section V-B), which leaks cross-origin image content on different Android devices.

Threat Model. We assume a remote attack scenario where a victim visits a website hosted or controlled by a remote attacker via their browser. Potential methods for targeted attacks could include, for instance, malicious web link distribution via targeted phishing attacks [43] or malicious advertising, as demonstrated in prior work [44]. The attacker's website embeds cross-origin content, like images or webpages, inside an `iframe`, which serves as an element to display content directly on the attacker's web page. Due to the strict isolation enforcement of the browser, the malicious website can not read the iframe's contents directly. Thus, the attacker's website utilizes sensor measurements gathered via the JavaScript sensor interface to leak the secret image content rendered inside the target iframe. Note that JavaScript grants access to the sensor API [37] per default to unprivileged JavaScript. For the general attack setup, we assume the victim runs Android 13 with an installed version of Google Chrome that does not include Google's patch applied in response to our disclosure, e.g., version 120. Our general threat model aligns with threat models presented in prior work [7], [22], [45], involving a remote attacker running JavaScript inside a malicious website on the victim's device.

A. Leakage Primitive of Rendered Images

First, we need a leakage primitive in order to perform our pixel-stealing attack. Therefore, we analyze the potential leakage of the Google Chrome browser's rendering process through measured sensor values provided by the generic sensor framework through JavaScript. Note that these sensor values are accessible to an attacker hosting a malicious website.

Methodology. Initially, we analyze the leakage properties of our sensor-based approach by distinguishing between two monochrome images consisting of 256 x 256, either all-white or all-black pixels. Here, we intentionally choose complementary colors since their representation in all RGB channels (i.e., 0 for black and 255 for white) results in sizable Hamming weight differences, resulting in substantial variations in power consumption during the rendering process between both images. We render both images in an alternating pattern on
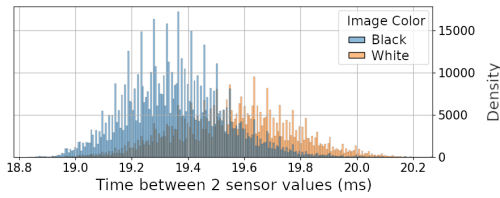
Fig. 8: Captured time intervals between consecutive sensor values from the absolute orientation sensor. The histogram shows distinguishable distributions for black and white images, i.e., timing differences between consecutive sensor values.

a canvas of the Google Chrome browser, with each image displayed for approximately $10\,$s. Similar to existing work [7], [22], we apply an SVG filter before rendering the image in each render pass of the browser window. The SVG filter performs GPU-intensive computations on the pixel values, leading to GPU power consumption variations due to the pixels' different Hamming weights. Specifically, the SVG filter applies a Gaussian blur filter with a standard deviation of 1 to each pixel of the image.

When a filter is applied multiple times to the same image, the web browser will not recompute the filter on each render pass and instead reuses cached results. Therefore, we add an image containing random pixel noise to the input image on every render pass before applying the Gaussian blur filter. This methodology preserves significant Hamming weight differences between the two images while only slightly altering the image, forcing a recomputation of the Gaussian blur filter. The images with random components used in this process are dynamically generated at runtime before initiating the measurement. Each time the SVG filter is invoked, a random image is chosen and leveraged for composition with the input image. Simultaneously, we capture sensor readings from the JavaScript generic sensor framework.

Analysis Results. We identified two different leakage primitives: The *magnetometer* and the *absolute orientation sensor*, which show variations when rendering different colored images. Specifically, we can report different sensor-based leakage: the measurements with the magnetometer lead to different signal frequencies for the two images, while the absolute orientation sensor leads to time differences between constitutive sensor values depending on the pixel color. Both sensors are accessible from JavaScript in the Google Chrome browser using the generic sensor framework. Access to the magnetometer requires one-time enabling of a browser flag [37] (which is cached by the web browser), while the absolute orientation sensor is always accessible without permission.

Figure 7 shows measurements gathered from our evaluation of the magnetometer. The graph illustrates the sensor values collected during the rendering process of alternating black and white images throughout $40\,$s, with a low pass filter applied to remove high-frequency noise. Moreover, the graph shows sensor values for an alternating pattern, i.e., $10\,$s of black and white images, where the blue signal (white image) and the

orange signal (black image) are concatenated for their respective time frame. While the absolute sensor values for both images are similar in the time domain, we observe differences in the signal's frequency domain. Specifically, when rendering white images, the signal oscillates at a lower frequency than during the rendering of black images. We hypothesize that this effect originates from changes in the GPU's operating voltage and frequency (i.e., P-states) while processing different colors. This is in line with our systematic analysis (see Section III), which shows that sensor measurements correlate with the system's battery voltage. Recent work [7], [22] demonstrates a similar effect by measuring execution time differences due to frequency changes.

Besides the magnetometer, we evaluate the *absolute orientation sensor*, and demonstrate distinguishable differences in the sensor measurements when rendering monochrome black-and-white images. While the raw absolute sensor values demonstrate little variations across different image colors, we observe variations in the time intervals between consecutive sensor values. Figure 8 shows a histogram of the captured time intervals between consecutive sensor values while rendering both black and white images. Here, the histogram shows distinguishable distributions for both classes of rendering. Precisely, time intervals between consecutive sensor values collected while rendering black images are, on average, lower than when rendering white images, thus allowing the image color of the rendered images to be distinguished. As a root cause behind this behavior, we suspect a security mechanism implemented in the JavaScript sensor interface. The JavaScript sensor interface implements an algorithm that performs threshold checks before reporting sensor values for specific sensors [38]. The interface discards sensor readings that do not significantly differ from the latest readings of the underlying Android sensor. The varying power consumption of the rendered images of different color results in variations of the sensor values by a varying magnitude. Consequently, the threshold check discards different amounts of sensor values per image color, leading to varying intervals between the sensor values per image color.

> We demonstrate that the magnetometer introduces leakage in the form of the oscillation frequency of the sensor signal, and the absolute orientation sensor introduces timing differences for consecutive sensor readings.

B. End-To-End Pixel Stealing Attack

We present an end-to-end pixel-stealing attack targeting Android devices, leveraging the generic sensor framework accessible through JavaScript in the Google Chrome web browser. Thereby, our attack successfully circumvents the cross-origin isolation enforced by the web browser, which typically isolates content from different origins. Precisely, cross-origin content, e.g., content rendered in containers like an `iframe`, is intended to be inaccessible to JavaScript outside that container. We demonstrate the effectiveness of our approach using the JavaScript sensor framework to break
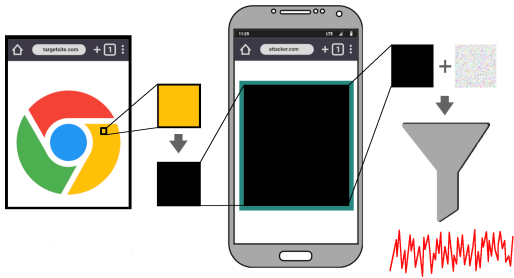
Fig. 9: High-level overview of the end-to-end pixel stealing attack on Android smartphones. The attacker runs a malicious website embedding an `iframe` containing cross-origin content, e.g., an image. We crop the rendered image down to the individual selected pixel and then apply the SVG filter stack individually to each pixel of the image to leak the color.

this isolation, recovering images from different origins. We evaluate our attack on different devices that demonstrated leakage in our initial systematic leakage analysis (cf. Section III), precisely the Google Pixel 6a and the Samsung Galaxy S20.
Attack Methodology. In Figure 9, we provide a schematic overview of the pixel-stealing attack. We follow a similar approach to recent work [7], [22], where the attacker runs a malicious website that embeds an `iframe` containing cross-origin content, e.g., an image. The size of the `iframe` is 256 x 256 pixels and the `iframe`'s content is rendered to the screen, visible to the user of the Android device. Protected by the cross-origin isolation of the web browser, the JavaScript running on the attacker's website is unable to access the content within the `iframe`. To extract individual pixel values from the rendered cross-origin content, the attacker applies an SVG filter stack on top of the rendering process of the `iframe`, as previously described in Section V-A.

To distinguish different pixels colors, the SVG filter stack is applied individually to each pixel of the image. Therefore, a pixel of interest is selected before applying the filter stack. Thus, the attacker crops the rendered image to the individual selected pixel, discarding the rest of the image. In JavaScript this is accomplished using the CSS `clip-path` property, which selects a visible image region while hiding the rest of the image outside that region. This clip-path region of size 1 x 1 is shifted over the image pixel by pixel, enabling the selective rendering of one pixel at a time. Subsequently, the cropped pixel is scaled up to 256 x 256 pixels, filling the entire `iframe` with the color of the selected pixel. The displayed image is converted to black and white to enhance the Hamming weight differences of the pixels. Following the selection and scaling of the target pixel, the SVG filter stack is applied to the image. We use the SVG filter described in Section V-A for our end-to-end pixel-stealing attack. In parallel to applying the filter stack, we collect sensor measurements from the JavaScript sensor framework at the highest data acquisition rates. We provide a detailed evaluation of the attack using different sensors, namely the magnetometer and the absolute orientation sensor on the Google Pixel 6a. Additionally, we

evaluate the attack on different devices of our initial leakage analysis (cf. Section III).

When reconstructing the individual pixel colors of the magnetometer traces, we calculate a Fast Fourier Transform (FFT) of the collected measurements per pixel, revealing each pixel's signal frequency spectrum. As the singal oscillates with varying frequencies at different pixel colors (see Section V-A), the FFT thus reveals peaks at different frequencies for the varying pixel colors. Using a threshold at a specific frequency, we can classify the pixel color of the image. Similarly, when evaluating the Absolute Orientation sensor, we use a thresholding approach on the average times between consecutive sensor values (see Section V-A), thus recovering the original image.
Evaluation. In our evaluation, we reconstruct a rendered image with dimensions of 48 x 48 pixels. We display the Google Chrome logo inside the iframe as an original image. We demonstrate image reconstruction using the different sensors and evaluate our attack on different target devices.

Figure 10 illustrates the reconstructed images using measurements of different sensors. Additionally, it depicts the corresponding reconstruction speed per pixel [s] and the reconstruction accuracy [%], comparing the leakage of the different sensors on a victim device such as the Google Pixel 6a. Here, the magnetometer provides the fastest reconstruction time of $5\,\mathrm{s}$/pixel, providing a reconstruction accuracy of $90\,\%$. The reconstructed image exhibits clear outlines, thus providing a distinct representation of the original content. The reconstructed image highlights sporadic, randomly distributed bit errors. Nevertheless, these bit errors do not hinder the recognition of the original image's content, as the shape of the image elements is clearly recognizable. Additionally, Figure 10 shows the reconstructed image from the absolute orientation sensor, demonstrating a reconstruction accuracy of $70\,\%$ at $10\,\mathrm{s}$ per pixel. While the reconstructed image quality is lower than that of the magnetometer trace, the image content is still recognizable.

> We demonstrate an end-to-end pixel-stealing attack in JavaScript in Google Chrome, bypassing the browser's cross-origin isolation by exploiting power-related Magnetometer and Absolute Orientation sensor readings.

Figure 11 depicts the results of the pixel-stealing attack on different mobile devices, identified in our initial leakage analysis (cf. Section III). The results of these evaluations using the magnetometer sensor demonstrate similar reconstruction accuracies around $90\,\%$ at reconstruction speeds around 5 to $10\,\mathrm{s}$/pixel. The resulting reconstructed images show distinct outlines with randomly distributed bit errors. In summary, our attack achieves results comparable to prior pixel-stealing attacks [7], [22], leveraging power-related signals such as the Hertzbleed effect (see Section VII-A).

> We demonstrate that smartphones identified in our initial leakage analysis are susceptible to sensor-based pixel-stealing attacks.

Image:	Original	Magnetometer	Abs. Orientation
Time/Pixel [s]:		5	10
Accuracy [%]:		90.2	70

Fig. 10: The results of our attack evaluation include the reconstructed images of the Google Chrome logo using the magnetometer and the absolute orientation sensor on the Google Pixel 6a, together with the leakage rates and accuracy.

Original	Samsung Galaxy S20 FE	Google Pixel 6a

Phone	Time/Pixel [s]	Accuracy [%]
Samsung Galaxy S20 FE	10	89.2
Google Pixel 6a	5	90.2

Fig. 11: Results of the pixel-stealing attack on the Samsung Galaxy S20 FE and the Google Pixel 6a with the corresponding leakage rates and reconstruction accuracies observing magnetometer traces.

VI. CPA ATTACK ON THE ARM AES INSTRUCTION

This section considers a local attack scenario where an attacker runs inside an app. This app has been installed by the user, e.g., via the Google Play store [46], and obeys the strong isolation enforcement of Android's app sandboxing. In this attack scenario, we demonstrate how a local attacker can exploit the default permitted sensor readings to leak secret information, i.e., encryption keys. In the following, we perform a case study that attacks a hardware-accelerated AES implementation by performing Correlation Power Analysis (CPA) [10]. In our CPA attack, we exploit the profiled properties of the previously analyzed geomagnetic rotation vector sensor (see Section IV). Specifically, we demonstrate a rank reduction attack by leaking individual AES key bytes.

Threat Model. We assume a local attacker operating within an Android app installed by the device user, e.g., via Android's Google Play store [46]. The victim's device runs Android 13 with the default configuration and no additional system modifications. The malicious application requires access to the Android sensor framework, which Android grants by default and does not require user confirmation. There is no additional requirement for user-granted system permissions to access any other functionality. Thus, listing the application on an official App Store, like the Google Play Store [46], would be possible without violating security policies. To be able to capture sensor readings, the application has to either run in the foreground or as a foreground service. Examples of such foreground services are a navigation app, fitness or sleep tracker initiated by the user. This service can collect sensor measurements as a part of its functionality, providing ongoing notifications without the need for the app to be in the foreground [47]. Finally, we assume that the malicious app can trigger arbitrary AES encryptions with known plaintext blocks. This assumption of an interface capable of initiating encryptions aligns with prior state-of-the-art research publications like the PLATY-PUS [3], Frequency-throttling [5] or the Plundervolt [48] attacks. The attacked AES implementation uses constant-cycle ARM NEON AES vector instructions [49] for each encryption round, as commonly implemented in application-class processors, e.g., smartphones. The used implementation

is comparable to recent AES implementations of real-world cryptographic libraries like Mbed TLS [50] or OpenSSL [51], without side-channel countermeasures.

Attack Setup and Methodology. The attacker operates within an unprivileged Android application evaluated on the Google Pixel 6a, with default privileges. The attacker has access to an interface capable of initiating AES encryptions with arbitrarily known plaintexts. The AES implementation makes use of dedicated ARM hardware instructions from the ARM NEON [49] cryptographic extension, specifically the `aese` and `aesmc` instructions. These vector instructions operate on 16-byte vector registers and execute the sub-steps of an AES round, such as *add roundkey*, *sub bytes*, and *shift rows* with the `aese` instruction, and *mix columns* with the `aesmc` instruction. Interleaving these instructions for each of the 10 AES rounds results in a full `AES128` implementation. Notably, the evaluated AES implementation is constant-cycle, always requiring the same number of cycles on each invocation, independent of the input data.

The attacker repeatedly invokes the AES implementation with known plaintext blocks. Simultaneously, the attacker collects measurements of the geomagnetic rotation vector sensor on the Google Pixel 6a, utilizing the highest available data acquisition rate of $4.571\,\mathrm{ms}$ ($\pm\,2.86\,\mu s$). The attacker aligns the invocations of the AES implementation to the profiled relative measurement window of each sensor event, as detailed in Section IV-B. Consequently, the attacker chooses a new plaintext for each sensor measurement window and repeatedly triggers the encryption with that plaintext.

Evaluation. First, we analyze the observable Hamming distance and Hamming weight leakage of all architectural intermediates, i.e., all the observable register values $\texttt{aesse}_i$ and $\texttt{aesmc}_i$ of the internal encryption state changing due to specific AES instructions. Second, we perform a CPA-style attack and report how the key candidates of this analysis evolve over the number of recorded samples.

Table III shows the results of the Hamming distance and Hamming weight analysis. We recorded approximately 35.6

12

TABLE III: Correlation analysis of the measurements and the architectural observable intermediates. We report the Pearson correlation coefficient r and the significance factor $|SF|$.

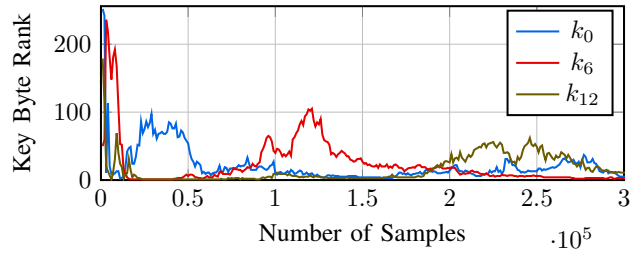| Hamming Weight			Hamming Distance		
Model	r in 10^{-3}	$|SF|$	Model	r in 10^{-3}	$|SF|$
hw(pt)	1.340	1.41	-		
hw(aesse$_0$)	-0.085	0.09	hd(pt, aesse$_0$)	-3.595	3.79
hw(aesmc$_0$)	2.572	2.71	hd(aesse$_0$, aesmc$_0$)	0.744	0.78
hw(aesse$_1$)	-0.671	0.71	hd(aesmc$_0$, aesse$_1$)	-0.878	0.92
hw(aesmc$_1$)	0.042	0.04	hd(aesse$_1$, aesmc$_1$)	-1.688	1.78
hw(aesse$_2$)	0.380	0.40	hd(aesmc$_1$, aesse$_2$)	2.870	3.02
hw(aesmc$_2$)	0.385	0.41	hd(aesse$_2$, aesmc$_2$)	0.295	0.31
hw(aesse$_3$)	-0.965	1.02	hd(aesmc$_2$, aesse$_3$)	0.196	0.21
hw(aesmc$_3$)	-0.847	0.89	hd(aesse$_3$, aesmc$_3$)	-1.678	1.77
hw(aesse$_4$)	-1.772	1.87	hd(aesmc$_3$, aesse$_4$)	1.238	1.30
hw(aesmc$_4$)	-1.465	1.54	hd(aesse$_4$, aesmc$_4$)	-1.065	1.12
hw(aesse$_5$)	-0.407	0.43	hd(aesmc$_4$, aesse$_5$)	1.113	1.17
hw(aesmc$_5$)	-0.050	0.05	hd(aesse$_5$, aesmc$_5$)	0.560	0.59
hw(aesse$_6$)	-0.175	0.18	hd(aesmc$_5$, aesse$_6$)	-0.211	0.22
hw(aesmc$_6$)	0.761	0.80	hd(aesse$_6$, aesmc$_6$)	-2.037	2.15
hw(aesse$_7$)	1.469	1.55	hd(aesmc$_6$, aesse$_7$)	-1.038	1.09
hw(aesmc$_7$)	1.121	1.18	hd(aesse$_7$, aesmc$_7$)	0.723	0.76
hw(aesse$_8$)	-1.521	1.60	hd(aesmc$_7$, aesse$_8$)	-1.535	1.62
hw(aesmc$_8$)	6.535	6.89	hd(aesse$_8$, aesmc$_8$)	1.878	1.98
hw(aesse$_9$)	1.138	1.20	hd(aesmc$_8$, aesse$_9$)	1.220	1.29
hw(aesmc$_9$)	-0.707	0.74	-		

Fig. 12: Rank complexity analysis of the collected traces of the AES implementation for different key byte guesses.

signal usable for a CPA attack, not all key bytes converge toward a stable result.

> The *geomagnetic rotation vector* sensor exposes a significant leakage signal of the AES implementation. Although the leakage exposes rather low correlation coefficients, an adversary can still reduce the key complexity.

VII. RELATED WORK & DISCUSSION

This section gives an overview of related work, discussing possible mitigations, limitations, and future work.

A. Related Work

Android Software-based Power Side Channels. Android versions prior to Android 7 (released in 2018) offered unrestricted access to power data via the sysfs [52]. This enabled unprivileged Android apps to exploit power readings directly without requiring power-related signals for estimation. Yan et al. [19] and Michalevsky et al. [20] demonstrated that software power interfaces allow to leak user locations and password lengths to unprivileged users [19]. Subsequent work [21] performed application fingerprinting using this power interface. Android 7 removed unprivileged access to this interface, removing the attack surface using direct power signals. Thus, Qin et al. [52] demonstrated a power estimation method to perform website fingerprinting by, e.g., reading Android 7's unprivileged CPU load and frequency information. Subsequently, Android 8 removed unprivileged access to these power-related interfaces [53], reducing the attack surface.

Sensor-based Attacks on Mobile Phones. Side-channel attacks on smartphones, especially attacks using sensors [54], mostly use motion data inferring, e.g., user interactions [24]. Demonstrated attacks perform user localization using motion sensors [25] or infer user input due to device motion [26]–[28]. In addition to attacks that exploit sensors' intended motion properties, there are attacks exploiting the parasitic signal captured with high-precision magnetic field sensors. Specifically, Matyunin et al. [29] demonstrated that the Android's magnetic field sensor can be used to build covert channels or website and application fingerprinting natively [29], and from JavaScript [55]. Another study [56] leaked application usage data through Magnetometer readings using deep neural networks. Even though these studies provide a great preliminary insight into the capabilities of an attacker, they

million samples over a time span of $44\,\mathrm{h}$. However, we only use 17.8 million samples in the following analysis due to filtering. During the experiment, we varied all possible input bytes of the plaintext to include all potential leakage influences. We report the Pearson correlation coefficient and the significance factor (see Section III-C) in the table for both the Hamming weight and the Hamming distance of the individual models, similar to the CPA analysis from Lipp et al. [3]. The overall Pearson correlation coefficients are relatively low in the range of 10^{-3}. Nevertheless, we observe that, for instance, the correlation of the Hamming weight of the output of the eighth *mix columns* output shows a high significance factor of $|SF| = 6.89$. Furthermore, we observe a significant leakage ($|SF| = 3.79$) of the Hamming distance between the plaintext and the output of the first aese instruction. Finally, we observe that the *direction* of the Pearson correlation coefficient is, i.e., changing the sign depending on the targeted round. For the CPA attack, we reuse the samples from the previous experiment. We use these recorded samples and perform a CPA analysis using the following model: $\mathcal{M} = \mathrm{hw}(\mathrm{sbox}[p_n \oplus k_n])$. Where p_n represents the nth plaintext byte and k_n the nth targeted key byte of the first round. For each key byte k_n, we enumerate all possible key byte candidates ranging from 0 to 255. We use the Pearson correlation coefficient to correlate the defined model $\mathcal{M}$ of each key byte candidate with the measured samples. Figure 12 shows the rank of the correct key candidate over the number of samples used in the CPA attack. We observe that the rank of the correct key candidate shows a downward trend and fluctuates after $150\,000$ samples between 0 and 50. Overall, we observe key bytes for which the CPA converges to a stable result (cf. Figure 12) like key byte 0, 6, and 12. However, some of the bytes did not show a meaningful trend. Therefore, we conclude that although the *geomagnetic rotation vector* sensor exposes a power-related

only leak information based on coarse-grained influences, e.g., CPU utilization, leaving the full potential of power-related sensor-based side channels to be explored. Thus, in this work, we show that various sensors, beyond magnetic field sensors, expose power-related signals capable of also leaking fine-grained differences in power, e.g., processed data operands. Besides power-related signals, Mohamed et al. [57] demonstrated that magnetometers in Apple iPhones can leak touch events. Shepherd et al. [58] illustrated how to leverage sensor multiplexing to construct a covert channel and conduct application profiling.

Pixel Stealing. Barth et al. [42] initially observed that SVG filters [59] leak rendered pixel colors in browsers. This led to the discovery of pixel-stealing attacks, exploiting data-dependent timing variations in the filters [60], [61]. Subsequently, browser vendors mitigated these attacks by removing these data-dependent execution time dependencies [62]–[64]. Despite removing data dependencies, Andrysco et al. [65] demonstrated that pixel stealing remained feasible using microarchitectural side channels. Kohlbrenner et al. [66] later demonstrated that timing side channels in floating point operations are still present, prompting additional mitigations from major browser vendors [67]–[69].

The introduction of software-based power side channels opened up a new attack vector concerning pixel-stealing attacks. As JavaScript does not provide a power interface, recent discoveries [7], [22] exploited power-related signals using the Hertzbleed effect, leveraging time as a proxy for power to perform pixel stealing. While Wang et al. [7] focused on laptop CPUs, achieving great leakage rates, Taneja et al. [22] performed evaluations on multiple hardware platforms, including mobile phones. On Android, they recover pixel values at rates between 10 and $19\,\mathrm{s/pixel}$ with an accuracy of 70 to 82%. Our work demonstrates additional power-related sensor signals, beyond timing, that enable pixel-stealing attacks. Our findings show recovery rates of 5 to $10\,\mathrm{s/pixel}$ and an accuracy of 70 to 90%, comparable to the results of Taneja et al. [22] (see Section V-B). Besides power-related signals, Olejnik et al. [70] demonstrated a pixel-stealing attack on mobile devices using the ambient light sensor, exploiting physical light changes due to rendered images. O'Connell et al. [45] demonstrated cache attacks leaking cross-origin content in the browser. Wang et al. [71] demonstrate pixel-stealing exploiting GPU data-compression, and Pustelnik et al. [72] leak pixels exploiting uninitialized GPU registers, showing the active research effort towards performing pixel-stealing attacks in the browser.

B. Discussion

Limitations and Mitigations. Similar to prior side-channel attacks [3], [6], mitigating our novel sensor-based attack must be considered with user experience in mind. A successful attack requires access to the Android sensor framework. Thus, a straightforward mitigation would be restricting or limiting sensor framework access. However, since many apps require sensor readings to provide basic functionality, removing access to the sensor framework would significantly impact user ex-

perience. Hiding specific sensors behind a permission prompt could reduce the attack surface, making attacks easier to detect while maintaining app functionality. However, prior work on permission prompts on Android has shown [73] that users may still grant permission, partially mitigating our attacks. Reducing the accuracy of the sensor readings could decrease leakage. This approach, similar as implemented in the Google Chrome browser [38], could make attacks harder while providing enough information for coarse-grained tasks required in general-purpose apps. However, reducing the interface's precision only partially prevents future attacks, as seen in the PLATYPUS attack [3] that exploited the coarse-grained RAPL interface, which seemed too imprecise for attacks at the time.

On a hardware level, one approach is implementing more sophisticated algorithmic countermeasures like masking [74], with cryptographic hardware instructions. This significantly increases the difficulty of attacks. Likewise, implementing algorithmic countermeasures from software can increase the security of existing unprotected systems. Software countermeasures typically increase latency, hindering widespread adoption. Grégoire et al. [75] demonstrated how to implement fast, higher-order masking using ARM NEON's vectorized AES instructions. This approach could be used to increase the security of a system from software. A more generic defensive approach is to add shielding to the sensors or to separate the sensor circuit from the CPU's power supply domain.

VIII. CONCLUSION

In this paper, we presented power-related side-channel attacks targeting Android phones through sensor-based leakage. We provided results of our systematic analysis of potential leakage channels misusing the Android sensor framework. We detailed novel insights into the characteristics of the geomagnetic rotation vector sensor, serving as a leakage primitive for our proof-of-concept exploit. We evaluated the power-related leakage in two proof-of-concept attacks: in a remote setting in the web browser and locally inside an app. As a remote JavaScript attacker, we present an end-to-end pixel-stealing attack, bypassing the browser's cross-origin isolation. In the local setting, we performed an AES CPA attack, exploiting energy variations measured through the geomagnetic rotation vector sensor, accessible to unprivileged Android applications.

REFERENCES

[1] D. R. Gnad, J. Krautter, and M. B. Tahoori, "Leaky noise: New side-channel attack vectors in mixed-signal IoT devices," *TCHES*, pp. 305–339, 2019.

[2] A. Kogler, J. Juffinger, L. Giner, L. Gerlach, M. Schwarzl, M. Schwarz, D. Gruss, and S. Mangard, "Collide+Power: Leaking Inaccessible Data with Software-based Power Side Channels," in *USENIX Security*, 2023.

[3] M. Lipp, A. Kogler, D. Oswald, M. Schwarz, C. Easdon, C. Canella, and D. Gruss, "PLATYPUS: Software-based Power Side-Channel Attacks on x86," in *S&P*, 2021.

[4] D. Genkin, N. Nissan, R. Schuster, and E. Tromer, "Lend Me Your Ear: Passive Remote Physical Side Channels on PCs," in *USENIX Security*, 2022.

[5] C. Liu, A. Chakraborty, N. Chawla, and N. Roggel, "Frequency throttling side-channel attack," in *CCS*, 2022.

[6] Y. Wang, R. Paccagnella, E. He, H. Shacham, C. W. Fletcher, and D. Kohlbrenner, "Hertzbleed: Turning Power Side-Channel Attacks Into Remote Timing Attacks on x86," in *USENIX Security*, 2022.

[7] Y. Wang, R. Paccagnella, A. Wandke, Z. Gang, G. Garrett-Grossman, C. W. Fletcher, D. Kohlbrenner, and H. Shacham, "DVFS frequently leaks secrets: Hertzbleed attacks beyond SIKE, cryptography, and CPU-only data," in *S&P*, 2023.

[8] H. Islam, Z. Zhang, and F. Yao, "PowSpectre: Powering Up Speculation Attacks with TSX-based Replay," in *AsiaCCS*, 2024.

[9] W. Wang, M. Li, Y. Zhang, and Z. Lin, "PwrLeak: Exploiting Power Reporting Interface for Side-Channel Attacks on AMD SEV," in *DIMVA*, 2023.

[10] E. Brier, C. Clavier, and F. Olivier, "Correlation Power Analysis with a Leakage Model," in *CHES*, 2004.

[11] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *CRYPTO*, 1999.

[12] J. D. Golić and C. Tymen, "Multiplicative Masking and Power Analysis of AES," in *CHES*, 2002.

[13] L. Goubin and J. Patarin, "DES and Differential Power Analysis: The "Duplication" Method," in *CHES*, 1999.

[14] J. Balasch, B. Gierlichs, O. Reparaz, and I. Verbauwhede, "DPA, bitslicing and masking at 1 GHz," in *CHES*, 2015.

[15] G. Goller and G. Sigl, "Side channel attacks on smartphones and embedded devices using standard radio equipment," in *COSCADE*, 2015.

[16] Z. Liu, N. Samwel, L. Weissbart, Z. Zhao, D. Lauret, L. Batina, and M. Larson, "Screen gleaning: A screen reading TEMPEST attack on mobile devices exploiting an electromagnetic side channel," in *NDSS*, 2021.

[17] P. Cronin, X. Gao, C. Yang, and H. Wang, "Charger-Surfing: Exploiting a Power Line Side-Channel for Smartphone Information Leakage," in *USENIX Security*, 2021.

[18] Intel, "Intel-SA-00389," 2020. [Online]. Available: https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00389.html

[19] L. Yan, Y. Guo, X. Chen, and H. Mei, "A Study on Power Side Channels on Mobile Devices," in *Symposium on Internetware*, 2015.

[20] Y. Michalevsky, A. Schulman, G. A. Veerapandian, D. Boneh, and G. Nakibly, "PowerSpy: Location Tracking Using Mobile Device Power Analysis," in *USENIX*, 2015.

[21] Y. Chen, X. Jin, J. Sun, R. Zhang, and Y. Zhang, "POWERFUL: Mobile app fingerprinting via power analysis," in *INFOCOM*, 2017.

[22] H. Taneja, J. Kim, J. J. Xu, S. van Schaik, D. Genkin, and Y. Yarom, "Hot Pixels: Frequency, Power, and Temperature Attacks on GPUs and ARM SoCs," in *USENIX Security*, 2023.

[23] Android, "Sensors Overview," 2023. [Online]. Available: https://developer.android.com/guide/topics/sensors/sensors_overview

[24] R. Spreitzer, V. Moonsamy, T. Korak, and S. Mangard, "Systematic classification of side-channel attacks: a case study for mobile devices," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 1, pp. 465–488, 2017.

[25] H.-H. Hsu, J.-K. Chang, W.-J. Peng, T. K. Shih, T.-W. Pai, and K. L. Man, "Indoor localization and navigation using smartphone sensory data," *Annals of Operations Research*, vol. 265, pp. 187–204, 2018.

[26] Spreitzer, Raphael, "Pin skimming: Exploiting the ambient-light sensor in mobile devices," in *SPSM*, 2014.

[27] L. Cai and H. Chen, "TouchLogger: Inferring Keystrokes on Touch Screen from Smartphone Motion," in *USENIX HotSec*, 2011.

[28] M. Mehrnezhad, E. Toreini, S. F. Shahandashti, and F. Hao, "Touchsignatures: identification of user touch actions and pins based on mobile sensor data via javascript," *JISA*, 2016.

[29] N. Matyunin, Y. Wang, T. Arul, K. Kullmann, J. Szefer, and S. Katzenbeisser, "Magneticspy: Exploiting magnetometer in mobile devices for website and application fingerprinting," in *WPES*, 2019.

[30] Google, "Bug: 326383778: Sensors: Round magnetometer sensor readings," 2024. [Online]. Available: https://github.com/chromium/chromium/commit/615c26084e5437d6db046ed22460a3653c078c4f

[31] D. Agrawal, B. Archambeault, J. R. Rao, and P. Rohatgi, "The EM side—channel (s)," in *CHES*, 2002.

[32] J.-J. Quisquater and D. Samyde, "ElectroMagnetic Analysis (EMA): Measures and Counter-Measures for Smart Cards," in *E-smart*, 2001.

[33] K. Gandolfi, C. Mourtel, and F. Olivier, "Electromagnetic Analysis: Concrete Results," in *CHES*, 2001.

[34] S. Mangard, E. Oswald, and T. Popp, *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer Science & Business Media, 2008.

[35] Intel, "Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide," 2023.

[36] Google, "High sensor sampling rate," 2021. [Online]. Available: https://googlesamples.github.io/android-custom-lint-rules/checks/HighSamplingRate.md.html

[37] ——, "Sensors for the web," 2017. [Online]. Available: https://developer.chrome.com/docs/capabilities/web-apis/generic-sensor

[38] W3C, "Generic Sensor API," 2017. [Online]. Available: https://www.w3.org/TR/2017/WD-generic-sensor-20170530/

[39] Kolin Ian King, "Stress-ng," 2024. [Online]. Available: https://github.com/ColinIanKing/stress-ng

[40] Oscar Lab, "Occurrence of instructions among C/C++ binaries in Ubuntu 16.04," 2018. [Online]. Available: https://x86instructionpop.com/

[41] Intel, "Guidelines for Mitigating Timing Side Channels Against Cryptographic Implementations," 2019. [Online]. Available: https://software.intel.com/security-software-guidance/secure-coding/guidelines-mitigating-timing-side-channels-against-cryptographic-implementations

[42] Barth, Adam, "Timing Attacks on CSS Shaders," 2011. [Online]. Available: https://www.schemehostport.com/2011/12/timing-attacks-on-css-shaders.html

[43] A. Burns, M. E. Johnson, and D. D. Caputo, "Spear phishing in a barrel: Insights from a targeted phishing campaign," *JOCE*, vol. 29, pp. 24–39, 2019.

[44] A. Zarras, A. Kapravelos, G. Stringhini, T. Holz, C. Kruegel, and G. Vigna, "The dark alleys of madison avenue: Understanding malicious advertisements," in *IMC*, 2014.

[45] S. O'Connell, L. A. Sour, R. Magen, D. Genkin, Y. Oren, H. Shacham, and Y. Yarom, "Pixel Thief: Exploiting SVG Filter Leakage in Firefox and Chrome," in *USENIX Security*, 2024.

[46] Google, "Google Play Store," 2024. [Online]. Available: https://play.google.com/

[47] Android, "Foreground Services," 2024. [Online]. Available: https://developer.android.com/develop/background-work/services/foreground-services

[48] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens, "Plundervolt: Software-based Fault Injection Attacks against Intel SGX," in *S&P*, 2020.

[49] ARM, "Arm Architecture Reference Manual for A-profile architecture," 2 2022.

[50] Linaro, "MBed TLS," 2024. [Online]. Available: https://www.trustedfirmware.org/projects/mbed-tls/

[51] OpenSSL, "OpenSSL: The Open Source toolkit for SSL/TLS," 2024. [Online]. Available: http://www.openssl.org

[52] Y. Qin and C. Yue, "Website Fingerprinting by Power Estimation Based Side-Channel Attacks on Android 7," in *TrustCom/BigDataSE*, 2018.

[53] Google, "Android O prevents access to /proc/stat," 2017. [Online]. Available: https://issuetracker.google.com/issues/37140047?pli=1

[54] A. I. Champa, M. F. Rabbi, F. Z. Eishita, and M. F. Zibran, "Are We Aware? An Empirical Study on the Privacy and Security Awareness of Smartphone Sensors," in *SERA*, 2023.

[55] N. Matyunin, N. A. Anagnostopoulos, S. Boukoros, M. Heinrich, A. Schaller, M. Kolinichenko, and S. Katzenbeisser, "Tracking private browsing sessions using cpu-based covert channels," in *WISEC*, 2018.

[56] H. Pan, L. Yang, H. Li, C.-W. You, X. Ji, Y.-C. Chen, Z. Hu, and G. Xue, "Magthief: Stealing private app usage data on mobile devices via built-in magnetometer," in *SECON*, 2021.

[57] R. Mohamed, H. Farrukh, Y. Lu, H. Wang, and Z. B. Celik, "iStelan: Disclosing Sensitive User Information by Mobile Magnetometer from Finger Touches," *PETs*, vol. 2, pp. 79–96, 2023.

[58] C. Shepherd, J. Kalbantner, B. Semal, and K. Markantonakis, "A Side-Channel Analysis of Sensor Multiplexing for Covert Channels and Application Profiling on Mobile Devices," *IEEE TDSC*, 2023.

[59] W3C, "Filter Effects Module Level 1," 2019. [Online]. Available: https://drafts.fxtf.org/filter-effects/

[60] R. Kotcher, Y. Pei, P. Jumde, and C. Jackson, "Cross-origin pixel stealing: timing attacks using CSS filters," in *CCS*, 2013.

[61] Stone, Paul, "Pixel-Perfect Timing Attacks with HTML5," in *Black Hat*, 2013.

[62] S. Abou-Hallawa, "Timing Attack on SVG feComposite Filter Circumvents Same-Origin Policy," 2016. [Online]. Available: https://github.com/WebKit/WebKit/commit/43863de

[63] Mozilla, "SVG Filter Timing Attack," 2011. [Online]. Available: https://bugzilla.mozilla.org/show_bug.cgi?id=711043

[64] crbug, "Security: Timing attack on SVG feComposite filter circumvents same-origin policy," 2016. [Online]. Available: https://issues.chromium.org/issues/40083699

[65] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham, "On subnormal floating point and abnormal timing," in *S&P*, 2015.

[66] D. Kohlbrenner and H. Shacham, "On the effectiveness of mitigations against floating-point timing channels," in *USENIX Security*, 2017.

[67] S. Abou-Hallawa, "Time channel attack on SVG Filters," 2017. [Online]. Available: https://github.com/WebKit/WebKit/commit/a65a5b8

[68] Mozilla, "Pixelstealing and history-stealing through floating-point timing side channel with SVG filters." 2017. [Online]. Available: https://bugzilla.mozilla.org/show_bug.cgi?id=1336622

[69] crbug, "Security: Cross-origin pixel reading and history sniffing via SVG filter timing attack," 2017. [Online]. Available: https://issues.chromium.org/issues/40086661

[70] Olejnik, Lukasz, "Stealing sensitive browser data with the W3C Ambient Light Sensor API," 2017. [Online]. Available: https://blog.lukaszolejnik.com/stealing-sensitive-browser-data-with-the-w3c-ambient-light-sensor-api/

[71] Y. Wang, R. Paccagnella, Z. Gang, W. R. Vasquez, D. Kohlbrenner, H. Shacham, and C. W. Fletcher, "GPU. zip: On the Side-Channel Implications of Hardware-Based Graphical Data Compression," *S&P*, 2024.

[72] F. D. Pustelnik, X. M. Sass, and J.-P. Seifert, "Whispering Pixels: Exploiting Uninitialized Register Accesses in Modern GPUs," in *EuroS&P*, 2024.

[73] W. Cao, C. Xia, S. T. Peddinti, D. Lie, N. Taft, and L. M. Austin, "A Large Scale Study of User Behavior, Expectations and Engagement with Android Permissions," in *USENIX Security*, 2021.

[74] M.-L. Akkar and C. Giraud, "An implementation of DES and AES, secure against some attacks," in *CHES*, 2001.

[75] B. Grégoire, K. Papagiannopoulos, P. Schwabe, and K. Stoffelen, "Vectorizing Higher-Order Masking," in *COSCADE*, 2018.

APPENDIX

A. Leakage Analysis Phones

In this section, we show the results of the systematic analysis of all the remaining devices of Section III. The systematic analysis in Tables IV, V, and VI, contains evaluations of devices released over the past years up to recently released phones, e.g., the Google Pixel 9 (released August 2024).

Besides the Google Pixel 6a in Section III, we also demonstrate the leakage properties of different generations of the Google Pixel series. Thus, we additionally evaluate the Pixel 7a (see Table Vc) and the Pixel 9 (see Table Vd). Despite the different CPU generations and sensors, all three Google Pixel devices show similar leakage properties. Each phone generation shows significant leakage ($r > 0.7$) regarding varying CPU utilization in three sensors, also demonstrating a lower, observable correlation in other sensors, e.g., orientation sensors. While the general trend is similar, there are slight differences with specific sensors. For example, the accelerometer shows little correlation concerning CPU utilization on Pixel 7a while showing medium correlation on the other Pixel devices. Regarding data-dependent leakage, the trend between the varying Pixel devices is comparable, with the geomagnetic rotation vector sensor showing the best results. Overall, 26 out of 47 sensors (55%) of three pixel devices demonstrate significant ($|SF| > 1$) data-dependent leakage.

Table Va shows the results of the systematic leakage analysis of the Honor View 20. It shows that 6 sensors (50%) demonstrate CPU utilization-dependent leakage ($r > 0.7$), and 7 sensors (58%) show a statistically significant ($|SF| > 1$) data-dependent leakage. The evaluation results of Honor P90 Lite, released in 2023, given in Table IV demonstrate lower correlations regarding CPU utilization-dependent leakage. Our analysis shows that 7 of 10 sensors show to data-dependent leakage on the Honor P90 Lite. We additionally evaluated 11 sensors of the One Plus 5T, in Table Vb. We observe 6 sensors (54.5%) demonstrating considerable CPU utilization-dependent leakage and 5 sensors (45.5%) showing statistically significant data-dependent leakage. Finally, we evaluated 3 Samsung devices, namely the Samsung Galaxy S20 FE (see Table VIa), the Samsung Galaxy A51 (see Table VIb), and the Samsung Galaxy S9 (see Table VIc). Out of the 57 evaluated sensors of the Samsung devices, 5 sensors (8.7%) show significant leakage ($r > 0.7$) concerning CPU utilization. Additionally, 15 sensors (26.3%) demonstrate data-dependent leakage ($|SF| > 1$). Notably, the Samsung Galaxy S9 does not show large correlations in CPU utilization while still showing statistically significant data-dependent correlations.

TABLE IV: Systematic leakage analysis of the all sensos on the Honor P90 Lite

Sensor	Resolution	Update Rate [ms]	Varying CPU Utilization r	Varying Data r	$|SF|$
ACCELEROMETER	$1.20 \cdot 10^{-3}$	5	0.028	-	-
MAGNETOMETER	$1.50 \cdot 10^{-1}$	20	0.645	0.009	1.253
ORIENTATION	$3.91 \cdot 10^{-3}$	5	0.325	0.004	1.225
GRAVITY	$1.20 \cdot 10^{-3}$	5	0.194	0.005	1.480
LINEARACCEL	$1.20 \cdot 10^{-3}$	5	0.049	-	-
ROTATION_VECTOR	$5.96 \cdot 10^{-8}$	5	0.612	0.008	1.821
UNCALI_MAG	$1.50 \cdot 10^{-1}$	20	0.665	0.007	1.062
GAME_ROTATION_VECTOR	$5.96 \cdot 10^{-8}$	5	0.309	0.008	2.217
GEOMAGNETIC_ROTATION_VECTOR	$5.96 \cdot 10^{-8}$	5	0.275	0.008	2.411
UNCALI_ACC	$1.20 \cdot 10^{-3}$	5	0.027	-	-

TABLE V: Systematic leakage analysis of the Honor View 20, the One Plus 5T, the Google Pixel 7a, and Google Pixel 9.

(a) Systematic leakage analysis of the all sensos on the Honor View 20.

Sensor	Resolution	Update Rate [ms]	Varying CPU Utilization r	Varying Data r	$\|SF\|$
accelerometer-lsm6ds3-c	$1.00 \cdot 10^{-5}$	4	0.610	-	-
akm-akm09918	$6.25 \cdot 10^{-2}$	10	0.915	0.008	1.529
orientation	$1.00 \cdot 10^{-1}$	10	0.633	0.006	1.251
st-lsm6ds3-c	$1.70 \cdot 10^{-5}$	4	0.547	-	-
gravity	$1.53 \cdot 10^{-1}$	10	0.713	0.008	1.738
linear Acceleration	$9.58 \cdot 10^{-3}$	10	0.484	-	-
rotation Vector	$5.96 \cdot 10^{-8}$	10	0.710	0.019	4.251
uncalibrated Magnetic Field	$6.25 \cdot 10^{-2}$	16.667	0.900	0.007	1.444
game Rotation Vector	$5.96 \cdot 10^{-8}$	10	0.793	0.011	2.479
uncalibrated Gyroscope	$1.70 \cdot 10^{-5}$	4	0.563	-	-
geomagnetic Rotation Vector	$5.96 \cdot 10^{-8}$	10	0.895	0.006	1.28
RPC sensor	$1.00 \cdot 10^{-0}$	500	-	-	-

(b) Systematic leakage analysis of the all sensos on the One Plus 5T.

Sensor	Resolution	Update Rate [ms]	Varying CPU Utilization r	Varying Data r	$\|SF\|$
BMI160 Accelerometer Uncalibrated	$2.39 \cdot 10^{-3}$	5	0.587	-	-
AK09911 Magnetometer	$5.99 \cdot 10^{-1}$	20	0.280	0.007	1.047
AK09911 Magnetometer Uncalibrated	$5.99 \cdot 10^{-1}$	20	0.311	-	-
BMI160 Gyroscope	$1.07 \cdot 10^{-3}$	5	-	-	-
BMI160 Gyroscope Uncalibrated	$1.07 \cdot 10^{-3}$	5	0.179	-	-
Gravity	$2.39 \cdot 10^{-3}$	5	0.778	0.006	1.766
Linear Acceleration	$2.39 \cdot 10^{-3}$	5	0.766	-	-
Rotation Vector	$5.96 \cdot 10^{-8}$	5	0.745	0.012	3.655
Game Rotation Vector	$5.96 \cdot 10^{-8}$	5	0.718	0.005	1.602
GeoMagnetic Rotation Vector	$5.96 \cdot 10^{-8}$	20	0.763	-	-
Orientation	$1.00 \cdot 10^{-1}$	5	0.778	0.008	2.455

(c) Systematic leakage analysis of the all sensos on the Google Pixel 7a.

Sensor	Resolution	Update Rate [ms]	Varying CPU Utilization r	Varying Data r	$\|SF\|$
LSM6DSV Accelerometer	$4.79 \cdot 10^{-3}$	5	0.039	-	-
MMC56X3X Magnetometer	$9.76 \cdot 10^{-2}$	10	0.940	0.008	1.388
Orientation Sensor	$1.00 \cdot 10^{-5}$	5	0.521	0.009	1.890
LSM6DSV Gyroscope	$1.22 \cdot 10^{-3}$	5	0.106	-	-
ICP20100 Pressure Sensor	$1.00 \cdot 10^{-4}$	25	0.185	-	-
Gravity Sensor	$1.00 \cdot 10^{-5}$	5	0.492	0.005	1.041
Linear Acceleration Sensor	$1.00 \cdot 10^{-5}$	20	0.158	-	-
Rotation Vector Sensor	$1.00 \cdot 10^{-5}$	5	0.559	0.017	3.769
MMC56X3X Magnetometer-Uncalibrated	$9.76 \cdot 10^{-2}$	10	0.923	0.005	1.080
Game Rotation Vector Sensor	$1.00 \cdot 10^{-5}$	5	0.424	0.010	2.220
LSM6DSV Gyroscope-Uncalibrated	$1.22 \cdot 10^{-3}$	5	0.449	0.007	1.635
Geomagnetic Rotation Vector Sensor	$1.00 \cdot 10^{-5}$	5	0.934	0.022	4.827
LSM6DSV Accelerometer-Uncalibrated	$4.79 \cdot 10^{-3}$	5	-	-	-
LSM6DSV Temperature	$3.91 \cdot 10^{-3}$	16.667	0.110	-	-
ICP20100 Temperature	$1.00 \cdot 10^{-2}$	25	0.107	-	-

(d) Systematic leakage analysis of the all sensos on the Google Pixel 9.

Sensor	Resolution	Update Rate [ms]	Varying CPU Utilization r	Varying Data r	$\|SF\|$
ICM45631 Accelerometer	$5.99 \cdot 10^{-4}$	5	0.439	-	-
MMC5616 Magnetometer	$9.76 \cdot 10^{-2}$	10	0.969	0.018	3.352
Orientation Sensor	$1.00 \cdot 10^{-5}$	5	0.417	0.017	4.652
ICM45631 Gyroscope	$1.33 \cdot 10^{-4}$	5	0.055	-	-
ICP20100 Pressure Sensor	$1.00 \cdot 10^{-4}$	25	0.085	-	-
Gravity Sensor	$1.00 \cdot 10^{-5}$	5	0.202	0.004	1.174
Linear Acceleration Sensor	$1.00 \cdot 10^{-5}$	20	0.511	0.009	1.320
Rotation Vector Sensor	$1.00 \cdot 10^{-5}$	5	0.433	0.008	2.233
MMC5616 Magnetometer-Uncalibrated	$9.76 \cdot 10^{-2}$	10	0.971	0.015	2.785
Game Rotation Vector Sensor	$1.00 \cdot 10^{-5}$	5	0.394	0.009	2.463
ICM45631 Gyroscope-Uncalibrated	$1.33 \cdot 10^{-4}$	5	0.187	-	-
Geomagnetic Rotation Vector Sensor	$1.00 \cdot 10^{-5}$	5	0.892	0.035	7.742
ICM45631 Accelerometer-Uncalibrated	$5.99 \cdot 10^{-4}$	5	0.461	-	-
ICM45631 Temperature	$7.81 \cdot 10^{-3}$	20	0.241	-	-
ICP20100 Temperature	$1.00 \cdot 10^{-2}$	25	0.350	-	-
VD6282 Rear Light Sensor	$0.00 \cdot 10^{-1}$	16	-	-	-

TABLE VI: Systematic leakage analysis of the Samsung S20 FE, the Samsung Galaxy A51, and the Samsung Galaxy S9.

(a) Systematic leakage analysis of the all sensos on the Samsung Galaxy S20 FE.

Sensor	Resolution	Update Rate [ms]	Varying CPU Utilization r	Varying Data r	$|SF|$
ICM42632M Accelerometer	$2.39 \cdot 10^{-3}$	5	0.137	0.008	2.457
ICM42632M Gyroscope	$5.33 \cdot 10^{-3}$	5	-	-	-
AK09918C Uncalibrated Magnetometer	$6.00 \cdot 10^{-2}$	10	0.748	0.006	1.141
AK09918C Magnetometer	$6.00 \cdot 10^{-2}$	10	0.737	-	-
ICM42632M Uncalibrated Gyroscope	$5.33 \cdot 10^{-4}$	5	0.131	-	-
Game Rotation Vector	$5.96 \cdot 10^{-8}$	10	0.217	0.014	3.001
Samsung Rotation Vector	$5.96 \cdot 10^{-8}$	10	0.645	0.007	1.677
STK31610 Light CCT	$1.00 \cdot 10^{-0}$	20	-	-	-
ICM42632M Uncalirated Accelerometer	$2.39 \cdot 10^{-3}$	5	-	-	-
STK31610 Uncalibrated Light	$1.00 \cdot 10^{-0}$	20	-	-	-
Gravity Sensor	$5.96 \cdot 10^{-8}$	10	0.631	0.005	1.162
Linear Acceleration Sensor	$2.39 \cdot 10^{-3}$	10	0.652	-	-
Orientation Sensor	$3.91 \cdot 10^{-3}$	10	0.661	0.006	1.369
STK31610 Light	$1.00 \cdot 10^{-0}$	20	-	-	-
ICM42632M Interrupt Gyroscope	$1.07 \cdot 10^{-3}$	20	-	-	-
TCS3407 Rear ALS	$1.00 \cdot 10^{-0}$	10	-	-	-
Calibrated Lux Sensor	$1.00 \cdot 10^{-0}$	20	-	-	-
STK31610 Light IR	$1.00 \cdot 10^{-0}$	20	-	-	-
Camera Light Sensor	$1.00 \cdot 10^{-0}$	20	-	-	-

(b) Systematic leakage analysis of the all sensos on the Samsung Galaxy A51.

Sensor	Resolution	Update Rate [ms]	Varying CPU Utilization r	Varying Data r	$|SF|$
AK09918C Magnetometer	$6.00 \cdot 10^{-2}$	8	0.800	-	-
LSM6DSL Gyroscope	$6.11 \cdot 10^{-4}$	8	0.114	-	-
TCS3701 Light	$1.00 \cdot 10^{-0}$	200	0.794	-	-
AK09918C Magnetometer Uncalibrated	$6.00 \cdot 10^{-2}$	8	0.798	-	-
LSM6DSL Gyroscope Uncalibrated	$6.11 \cdot 10^{-4}$	8	0.104	-	-
TCS3701 Light CCT	$1.00 \cdot 10^{-0}$	200	0.318	-	-
Samsung Game Rotation Vector Sensor	$5.96 \cdot 10^{-8}$	10	0.258	0.005	1.297
Samsung Gravity Sensor	$2.39 \cdot 10^{-3}$	10	0.204	0.015	3.487
Samsung Rotation Vector Sensor	$5.96 \cdot 10^{-8}$	10	0.179	0.005	1.339
Samsung Orientation Sensor	$3.91 \cdot 10^{-3}$	10	0.256	0.005	1.330
LSM6DSL Accelerometer	$2.39 \cdot 10^{-3}$	8	-	-	-
Samsung Linear Acceleration Sensor	$2.39 \cdot 10^{-3}$	10	-	-	-
Interrupt Gyroscope	$6.11 \cdot 10^{-4}$	20	-	-	-
VDIS Gyroscope	$6.11 \cdot 10^{-5}$	10	-	-	-
Calibrated Lux Sensor	$1.00 \cdot 10^{-0}$	200	-	-	-

(c) Systematic leakage analysis of the all sensos on the Samsung Galaxy S9.

Sensor	Resolution	Update Rate [ms]	Varying CPU Utilization r	Varying Data r	$|SF|$
LSM6DSL Acceleration Sensor	$2.39 \cdot 10^{-3}$	2	-	-	-
LSM6DSL Gyroscope Sensor	$6.11 \cdot 10^{-4}$	2	-	-	-
Acceleration Sensor UnCalibrated	$2.39 \cdot 10^{-3}$	2	-	-	-
Gyroscope sensor UnCalibrated	$6.11 \cdot 10^{-4}$	2	-	-	-
AK09916C Magnetic field Sensor	$6.00 \cdot 10^{-2}$	10	0.157	-	-
Uncalibrated Magnetic Sensor	$6.00 \cdot 10^{-2}$	10	0.135	0.012	1.750
LPS22H Barometer Sensor	$2.44 \cdot 10^{-4}$	100	0.102	-	-
Samsung Rotation Vector	$5.96 \cdot 10^{-8}$	10	0.140	0.007	1.081
Samsung Game Rotation Vector	$5.96 \cdot 10^{-8}$	10	0.140	0.011	1.668
Gravity Sensor	$5.96 \cdot 10^{-8}$	10	0.145	0.011	1.679
Linear Acceleration Sensor	$2.39 \cdot 10^{-3}$	10	0.124	-	-
Orientation Sensor	$3.90 \cdot 10^{-3}$	10	0.151	0.013	2.001
TMD4906 lux Sensor	$1.00 \cdot 10^{-0}$	200	-	-	-
TMD4906 RGB IR Sensor	$1.00 \cdot 10^{-0}$	200	-	-	-
TMD4906 RGB Sensor	$1.00 \cdot 10^{-0}$	200	-	-	-
Light Flicker Sensor	$0.00 \cdot 10^{-1}$	200	-	-	-
MAX86915 Rear ALS	$1.00 \cdot 10^{-0}$	10	-	-	-
MAX86915 IR	$1.00 \cdot 10^{-0}$	10	-	-	-
MAX86915 RED	$1.00 \cdot 10^{-0}$	10	-	-	-
MAX86915 GREEN	$1.00 \cdot 10^{-0}$	10	-	-	-
MAX86915 BLUE	$1.00 \cdot 10^{-0}$	10	-	-	-
MAX86915 HRM RAW	$1.00 \cdot 10^{-0}$	10	-	-	-
HeartRate Sensor	$1.00 \cdot 10^{-0}$	1000	-	-	-