

A Side-channel Analysis of Sensor Multiplexing for Covert Channels and Application Fingerprinting on Mobile Devices

Carlton Shepherd, Jan Kalbantner, Benjamin Semal, and Konstantinos Markantonakis

Abstract—Mobile devices often distribute measurements from a single physical sensor to multiple applications using software-based multiplexing. On Android devices, the highest requested sampling frequency is returned to all applications even if other applications request measurements at lower frequencies. In this paper, we comprehensively demonstrate that this design choice exposes practically exploitable side-channels based on frequency-key shifting. By carefully modulating sensor sampling frequencies in software, we show that unprivileged malicious applications can construct reliable spectral covert channels that bypass existing security mechanisms. Moreover, we present a novel variant that allows an unprivileged malicious observer app to fingerprint other victim applications at a coarse-grained level. Both techniques do not impose any special assumptions beyond accessing standard mobile services from unprivileged applications. As such, our work reports side-channel vulnerabilities that exploit subtle yet insecure design choices in mobile sensor stacks.

Index Terms—Side-channel analysis, sensor stacks, mobile systems security.

1 INTRODUCTION

MOBILE devices contain an array of sensors that measure the device’s location, position, and ambient environment. The Android operating system currently supports over a dozen sensors, from traditional accelerometers and gyroscopes to magnetic field, temperature, humidity and air pressure sensors [7]. Devices can host several, sometimes multi-sensor integrated circuits (ICs) in millimeter-sized packages. This has been driven by the proliferation of low-cost micro-electromechanical systems (MEMS), which has miniaturised physical measuring apparatus to the micrometer scale. Today, on-board sensors are used ubiquitously for implementing game controllers, detecting screen orientation changes, and gesture and human activity recognition [7], [20], [46], [22], [16], [17], [18].

At run-time, multiple software applications may attempt to access measurements from one or more physical sensors. For example, a fitness tracker and navigation app may both poll the device’s magnetometer for detecting the user’s directional heading. To resolve this, a widely used approach is to use software-based multiplexing, where measurements are returned to *all* applications at the maximum requested sampling frequency. That is, if apps *A* and *B* request the same sensor at 100Hz and 50Hz respectively, then measurements are returned at 100Hz to both *A* and *B*. Consequently, there is no guarantee that measurements will arrive at the specified rate if another app requests the *same* sensor at a *higher* frequency.

In this work, we show that sensor measurement multiplexing exposes a design flaw that can be leveraged as a software-controlled side-channel. In the first part of this work (§4), we show how unprivileged applications can construct spectral covert channels for unauthorised inter-process communication (IPC). At a high level, this is achieved using a low-frequency carrier signal generated by the receiver application. A transmitting application can then deterministically modulate the receiver’s signal by requesting the same sensor at a higher sampling frequency. This is based on frequency-shift keying (FSK), where high-frequency transmission signals are multiplexed into the lower-frequency receiver signal. For the first time, we present detailed experimental results showing that common mobile device sensors can be exploited on different devices from major manufacturers. Our approach enables arbitrary bit-string transmission between two same-device applications, including low resolution images. The method bypasses existing IPC protection mechanisms, such as application sandboxing, with low error depending on the chosen sensor and device.

In the second part of this paper (§5), we present a novel variant of this technique that enables coarse-grained application fingerprinting. This follows the same approach of creating low-frequency carrier signals. However, this time, a single malicious application establishes carrier signals for *all* device sensors simultaneously. One of these signals is then modulated when a sensor-enabled victim app starts requesting measurements, which can be detected by the malicious app. We evaluate this approach using a two-part study. Firstly, we show how various sensor sampling constants provided by the Android SDK [8] can be detected with high accuracy and near real-time latency. These constants are designed to developers balance utility and battery consumption for various sensor use cases (e.g. games,

• All authors are of the Information Security Group at Royal Holloway University of London, Egham, Surrey, United Kingdom.
E-mail: carlton.shepherd@rhul.ac.uk

This work has been submitted to the IEEE for possible publication. Copyright may be transferred without notice, after which this version may no longer be accessible.

detecting screen orientation changes, and UI interactions). In the following study, the top 250 Android applications according to AndroidRank [10] (February 2022) are tested. We show how 1-in-5 of all tested applications (57/250; 22.8%) used detectable sensor and sampling frequencies, which could be leveraged for user behavioural profiling.

For both attacks, design and implementation information, detailed experimental results, and corrective recommendations are presented. Furthermore, we analyse how recent measures introduced in Android 9, intended to limit long-polling continuous sensors, are ultimately insufficient. Our presented techniques do not impose any special requirements, e.g. rooting or kernel-mode access, beyond the use of sensor APIs available to developers through the Android Sensor SDK.

1.1 Contributions

We extensively examine the extent to which sensor measurement multiplexing can be used for: ① spectral covert channels for app-wise IPC; and ② fingerprinting sensor-enabled victim applications. The methods are evaluated against a large range of standard mobile sensors using devices from separate major OEMs, thus affecting millions of devices globally. Moreover, we show how 22.8% of the top 250 Android applications are vulnerable to our fingerprinting technique to some degree. Proof-of-concept code for the attacks is made open-source to facilitate future research.¹

1.2 Responsible Disclosure

The results in this paper were disclosed to Google’s Android Security Team on 3rd June 2021 under a 90-day disclosure period. They were acknowledged on the same day, a severity assessment was provided on the 21st June 2021. The issues will be addressed in a forthcoming major Android release.

2 RELATED WORK

Current research has focussed on timing side-channels using CPU cache contents (PRIME+PROBE [41], FLUSH+RELOAD [49], FLUSH+FLUSH [19], and their variants [28], [15], [51], [36], [50], [26], [37], [29]), exploiting contention in DRAM memory controllers [44] and GPU access patterns [47], [39], [24]. Additionally, mobile sensors have found wide utility for out-of-band channels using the device’s ambient environment. MEMS gyroscopes have been used as low-frequency microphones for detecting ambient speech [38] and accelerometers for inferring touch interactions [48], [42], [14], [31], [32]. CPU core temperature sensors have been used for same-core and cross-core thermal side-channels, where victim processes deterministically increase the core temperature observed by a malicious application [33], [11], [30]. Novak et al. [40] described how camera flashes, vibrations, and device speakers can be used to transmit data to a (same-device) receiver. Block et al. [12] used a similar approach with ultrasonic frequencies. In 2016, Matyunin et al. [34] showed

¹ Proof-of-concept repository URL: <https://github.com/cgshep/android-multiplexing-security-pocs>

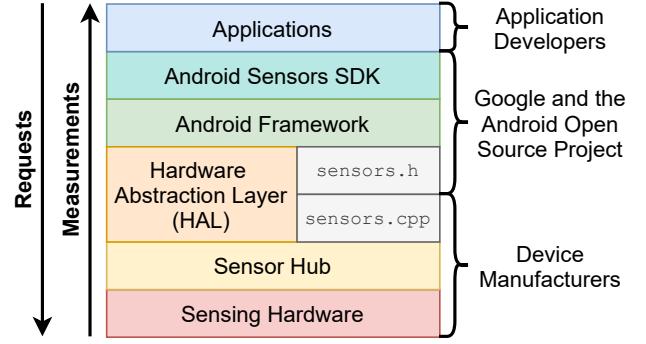


Figure 1: Components, corresponding owners, and information flow of the Android sensor stack (adapted from [8]).

how EM emissions of I/O operations can be detected by mobile magnetic field sensors. In 2019, the MagneticSpy [35] attack showed how a malicious app could fingerprint same-device apps and web pages using the magnetic field sensor with >90% accuracy.

To distinguish this work, we do not use physical transmission media. Instead, we exploit side-effects of the *software interfaces* used for resolving simultaneous accesses to sensing hardware. We take inspiration from Ulz et al. [45], who presented methods for creating covert channels on embedded systems using: ①, unused/reserved registers of sensing microcontroller units (MCUs); ②, reading/writing to MCU register configuration bits; and ③, timing differences between the activation of sensor MCUs. The authors posit that observable effects of different requested sampling periods could be used for frequency-based information encoding. However, detailed experimental evidence was not presented using various sensors and devices, and their error rates. In this work, we comprehensively and empirically show that continuous sensors on Android devices are vulnerable to covert channels using sensor measurement multiplexing. Moreover, we develop a novel attack variant for fingerprinting sensor-enabled victim applications.

3 THE ANDROID SENSOR STACK

The Android framework supports 13 hardware- and software-based sensors.² Hardware sensors take measurements directly from a sensor IC, e.g. a MEMS gyroscope. Software (virtual) sensors derive their measurements using signals from one or more hardware sensors within a sensor hub or the operating system. The exact separation between physical and virtual sensors is OEM-dependent. As a guide, accelerometers, gyroscopes and magnetic field sensors are usually implemented in hardware, while rotation vector and linear acceleration sensors are common virtual sensors [7].

The Android sensor stack (Figure 1) comprises several layers for translating raw physical values, which are device- and OEM-dependent, into standard interfaces that can be used by application developers. At an application level,

² We exploit continuous sensors defined in the Android Sensor SDK [7]. GPS location, sound levels, and detecting nearby Bluetooth and Wi-Fi devices have also been treated as so-called ‘sensors’ [21], [23], [40], [12]. However, these use modalities do not use sensor multiplexing and are not vulnerable to the same attacks.

developers use the `SensorManager` class for sensor enumeration and for measurement acquisition. Measurements are requested using the `SensorEventListener` interface and by calling the `registerListener` with the desired sensor and sampling rate. The SDK routes requests to the HAL—a single-client layer that abstracts OEM-specific firmware—which is shared by Android OS and all user applications. The HAL communicates directly with sensing hardware or, optionally, to a sensor hub for pre-processing measurements without waking the main application processor. The HAL returns the resulting data to the Android framework using FIFO fast message queues (FMQs), thus avoiding kernel involvement [2]. The framework returns sensor data to the requesting application as `SensorEvent` objects on an event-driven and first-in first-out (FIFO) basis to its `SensorEventListener` interface. When multiple apps register requests to a single sensor simultaneously, the multiplexing mechanism is used irrespective of whether physical or virtual sensors are requested.

4 BUILDING COVERT CHANNELS FROM SENSOR MULTIPLEXING

We recognised that a lack of appropriate controls at the Sensor API layer enables applications to influence the sampling frequencies of other applications. This section develops an FSK-based covert channel based on this observation.

4.1 Threat Model

The covert channel involves two colluding applications—a transmitter, Trn , and receiver, $Recv$ —who wish to establish a uni-directional communication channel that bypasses Android’s IPC security mechanisms. The attack scenario assumes that Trn has access to security- or privacy-sensitive data; for example, permission to access SMS data or GPS co-ordinates. Trn wishes to send this data to $Recv$, which does not possess these permissions. However, $Recv$ does have the ability to extract data from the device, such as permissions for accessing the internet to leak data to a remote server. Crucially, the user may not wish to download $Recv$ if it requests permissions to sensitive data and a data transmission medium. Our proposed channel enables Trn to leak data to $Recv$ using only standard methods for receiving sensor measurements. These are implemented as separate services launched independently by Trn and $Recv$, which may be disguised as two legitimate applications.³

4.2 Channel Design

The channel relies on Trn and $Recv$ targeting a shared sensor, S . Initially, $Recv$ registers a sensor listener for S using a long (slow) sampling period, T_c . After this, Trn repeatedly registers and unregisters listeners for S using a faster sampling period, T_{tr} . The repeated registering/unregistering corresponds to the bits of information that Trn it wishes to transmit. Due to the multiplexing phenomenon, $Recv$ ’s observed sampling period will modulate between T_c

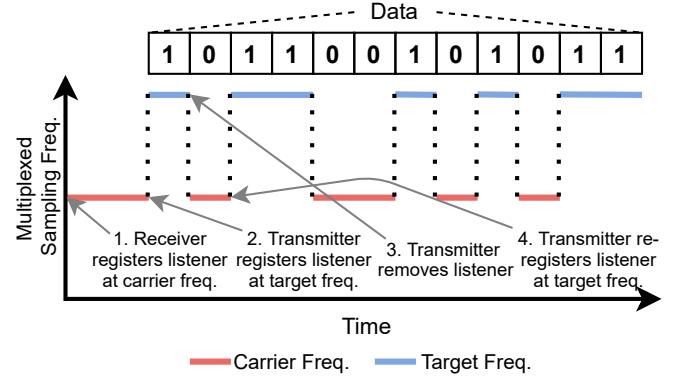


Figure 2: Simplified bit transmission procedure; the multiplexed frequency is observed by both applications.

and T_{tr} . After each listener (un-)registration, Trn must wait for a short time period (pulse width), w , to allow the new sampling period to be multiplexed into $Recv$ ’s signal. Note that the channel is not limited to two frequencies, T_c and T_{tr} . It is possible to register multiple sampling periods at higher frequencies than T_{tr} , which are modulated at their own rates. Using this observation, we constructed a transmission protocol using four sampling periods: ① to define the start of a message transmission, i.e. a syncword, T_{sync} ; ② to indicate transmission endings, T_{end} ; ③ $Recv$ ’s carrier period, T_c ; and ④ Trn ’s modulating period, T_{tr} . By setting appropriate sampling periods and pulse widths—explored in §4.3—an FSK-based binary transmission channel is established.

The channel involves three stages:

1. Channel initialisation: $Recv$ registers measurements using a target sensor, S , with period T_c . Trn registers a listener for the *same* sensor, S , but at a significantly higher frequency, T_{sync} , to signal the start of a transmission.

2. Data transmission (Figure 2): Trn unregisters the listener with sampling period T_{sync} . Trn transmits a bit-encoded secret, $v = [b_0, b_1, \dots, b_n]$, by registering a new listener at T_{tr} when $b_i = 1$ for time w . Conversely, Trn unregisters the listener and waits for time w to transmit $b_i = 0$. This causes $Recv$ ’s sampling frequency to return to T_c . In the background, $Recv$ observes these frequency changes and appends the bit values to an internal buffer.

3. Channel termination: Trn ends the transmission by registering a listener with period T_{end} . After detecting this, $Recv$ conducts any post-transmission operations, such as sending the buffer to an external server.

4.3 Implementation

State machines were implemented for managing our protocol using two separate Android applications for Trn and $Recv$. The full implementations are released as open-source software (see §1). The applications targeted *continuous* sensors on the devices under test, which deliver measurements as `SensorEvent` objects as soon as they are available from the HAL. Continuous reporting sensors comprise the majority of sensors available in Android [7]. In this work, the accelerometer (AC), gyroscope (GY), gravity (GR), linear acceleration (LA), magnetic field (MF), and rotation vector (RV) sensors were successfully utilised. While theoretically

3. One-to-one communication is described for simplicity; however, our approach can be used without alteration as a covert broadcast channel to multiple receivers. This is possible because sensor multiplexing modulates the sampling frequency of *all* applications.

vulnerable, sensors with other reporting modes, e.g. one-shot and on-change reporting, cannot be leveraged as effectively. These modalities require constant user interaction for consistent event generation.

4.3.1 Test Devices and Hardware Information

Three test devices from separate OEMs were evaluated:

- 1) *Xiaomi Poco F1* with a Qualcomm Snapdragon 845 (octo-core at 2.8GHz, 6GB RAM) and Android 10 (build QKQ1.190828.002). Cost: £309/~/\$370 USD (2019).
- 2) *Google Pixel 4A* with a Qualcomm SDM730 Snapdragon 730G (octo-core at 2x2.2GHz and 6x1.8GHz, 6GB RAM) and Android 11 (build RQ2A.210405.005). Cost: £349/~/\$450 USD (2020).
- 3) *Motorola Moto G5* with a Qualcomm MSM8937 Snapdragon 430 (octo-core at 1.2GHz, 2GB RAM) and Android 8.1 (build OPP28.82-19-4-2). Cost: £120/~/\$170 USD (2017).

The covert channel leverages changes in sampling frequencies, thus the fastest supported sampling rate acts as a hard throughput limit for a given sensor. Sensing hardware information can be gathered using the `List<Sensor> getSensorList (int type)` function from the `SensorManager` class, where `int type` is `Sensor.TYPE_ALL`. Next, `getMinDelay()` can be used to find the minimum latency allowed between two `SensorEvent` objects in microseconds for continuous sensors. We used this approach to find the minimum supported sampling periods for each device and sensor, including their hardware models (where known). This is shown in Table 1.

4.3.2 Challenges

During preliminary experiments, it became evident that the protocol implementation must overcome two obstacles:

[O1]: Measurements from mobile device sensors are subject to significant temporal jitter.

Sampling jitter complicated the ability to precisely infer the current sampling period. According to the sensor batching documentation: “Physical sensors sometimes have limitations on the rates at which they can run and the accuracy of their clocks. To account for this, the actual sampling frequency can differ from the requested frequency as long as it satisfies the [following] requirements.” [1]. If the requested frequency is below the min. frequency, then between 90%–110% of the min. frequency must be returned. If the request is between the min. and max. frequency, then 90%–220% of the requested frequency must be returned. Lastly, if the request is above the max. supported frequency, then 90%–110% of the max. frequency must be returned, but below 1100Hz [1]. Measurement jitter was studied by sampling each sensor at its maximum supported rate on each device (Table 1). 10,000 measurements were collected per sensor and per device, before calculating the inferred sampling period using the microsecond-level time between consecutive `SensorEvent` objects. (Sensor measurements are managed internally using FIFO FMQs, thus accurate period inferencing is possible using the timestamps of sequential `SensorEvents` [6]).

Figure 3 shows the distributions of the inferred sampling periods. The disparity in device- and sensor-wise jitter likely arises from varying sensor manufacturing tolerances. This

is compounded by implementation differences in proprietary, OEM-specific sensor HALs. To overcome this, an error threshold, $\epsilon = 0.1$ (10%), was used as a detection tolerance for each protocol frequency band. In general, the MF sensor exhibited the lowest jitter (<0.5% error from the mean, Xiaomi Poco F1 and Google Pixel 4A; no MF sensor on the Moto G5). The AC, GR and LA sensors were the next best performing (<1–2%, Xiaomi Poco F1; <2%, Pixel 4A; <2.5%, Moto G5). The GY and RV exhibited the largest jitter (<2–4%, Xiaomi Poco F1; <2%, Pixel 4A; <2.5–5%, Moto G5). It is noteworthy that jitter tends to increase when the minimum supported sampling period decreases.

[O2]: The *requested* sampling period can significantly differ from the *actual* period at which measurements are returned.

On the Xiaomi Poco F1, the GR, LA and RV sensors exhibited step function-like behaviour, and regularly over-sampled the requested period (Figure 4). In the worst case, the GR and LA sensors sampled at a ~40% faster rate than requested. This differed between IC manufacturers on the same device: the least accurate (GR and LA) belonged to an undisclosed Qualcomm IC, while the most accurate—the AC, GY and MF sensors—were provided by a Bosch BM160 and AKM AK0991x. The RV sensor also consistently over-sampled, but without the step-like behaviour of the GR and LA sensors. Generally, the Google Pixel 4A exhibited the best response rates: all actual periods were within >95% of the requested period, exceeding 99% in most cases. Only two Moto G5 sensors responded well: the AC and GY sensors (from a Bosch IC). The remaining Moto G5 sensors returned only a single frequency irrespective of the requested frequency. This prevented the use of multiple frequencies for our channel, so these sensors were removed from further consideration.

4.4 Evaluation

This section evaluates the developed covert channel using the six continuous sensors from §4.3.

4.4.1 Methodology

The average error rate and throughput (bit rate) of the proposed covert channel was evaluated for each device. The pulse width—the length of time between transmitting individual bits—was the dominating factor for the channel’s throughput. Intuitively, the shorter the pulse width, then the greater the bit rate. Protocol sampling bands were based on the maximum supported frequency as the base signal. Other frequencies were derived from multiples of this period according to the relationship in Equation 1. Where this method could not be used, the next distinct sampling rate at a longer period was selected. The full set of sampling bands used on each device is provided in Appendix A.

$$Trn(T_{end}) \ll Trn(T_{sync}) \ll Trn(T_{tr}) \ll Recv(T_c) \quad (1)$$

We measured 100 transmissions of 64–256 bit random sequences on a per-sensor and per-device basis. For each configuration, pulse widths were evaluated in the following range: the minimum value was the shortest period before total channel failure was observed (no bits detected by

Table 1: Sensing hardware and their minimum supported sampling periods (μs).

Sensor	Xiaomi Poco F1 Vendor	Xiaomi Poco F1 Min. Period	Motorola Moto G5 Vendor	Motorola Moto G5 Min. Period	Google Pixel 4A Vendor	Google Pixel 4A Min. Period
AC	Bosch BM160	2500	Bosch*	10000	STM LSM6DSR	2404
GR	Qualcomm*	5000	Motorola*	10000	Google*	5000
GY	Bosch BM160	5000	Bosch*	5000	STM LSM6DSR	2404
LA	Qualcomm*	5000	Motorola	10000	Google*	20000
MF	AKM AK0991x	10000	—	—	STM LIS2MDL	10000
RV	Xiaomi*	5000	Bosch*	5000	Google*	5000

*: Specific model not known, —: Not supported.

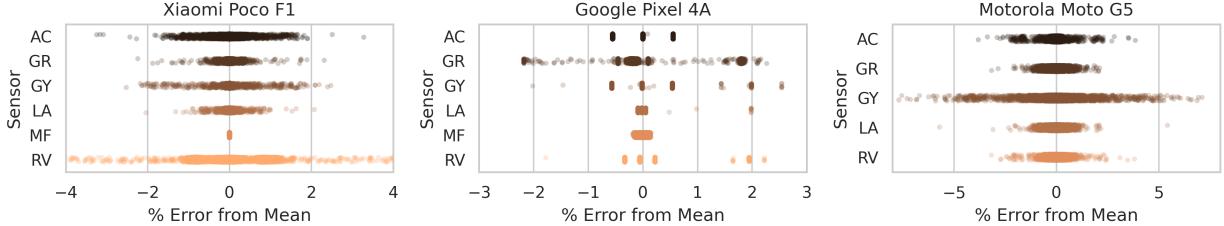


Figure 3: Inferred sampling period distributions using the lowest supported periods for each device and sensor.

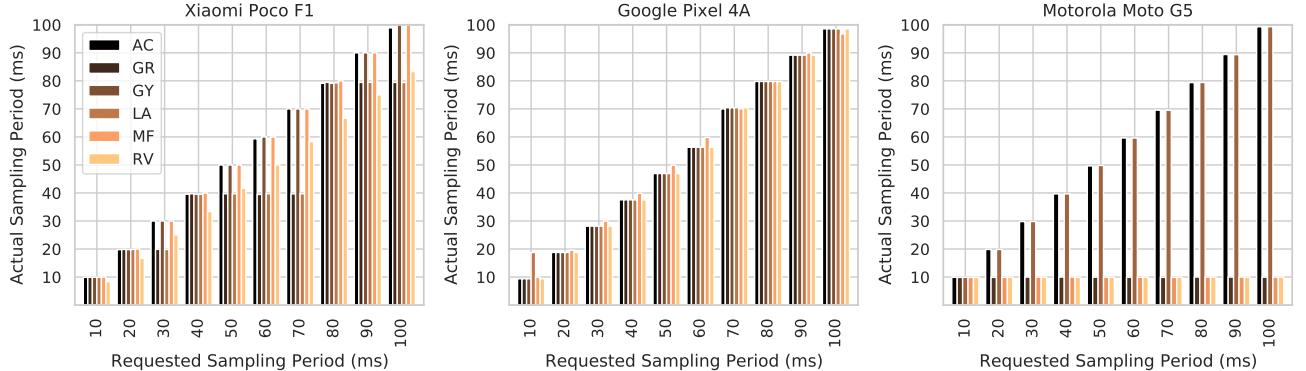


Figure 4: Sensor measurement sampling responses.

Recv). This value was increased until error-free transmission was achieved (across 100 bit-strings) or the error rate was unchanged. In total, 74 configurations were evaluated across the devices and sensors.

Data files were created on *Trn* and *Recv* at each channel instance, containing the transmitted and received bit-strings, and timestamps for each sensor and pulse width. The timestamps corresponded to immediately before the syncword transmission (*Trn*-side) and after the post-amble signal (*Recv*-side). The data files from the experiments—completed over two days by three volunteers—were retrieved from the devices for analysis. The average (median) bit rate was then calculated for each pulse width, device, and sensor. The transmission error rate was measured using the Levenshtein (edit) distance between the transmitted and received bit-strings. This metric, also used in [36], [13], [25], [27], [29], counts the insertions, deletions and substitutions for reverting the received bit-string to the transmitted one. Bit deletions/omissions were the primary source of transmission errors. This is due to Android’s lack of real-time measurement delivery guarantees, which became increasingly pronounced at higher sampling frequencies.

4.4.2 Results

The experimental results are presented in Figure 5. For each sensor, very low transmission error rates were attained on at least one evaluation device. The best-case rates are shown in Table 2. The AC sensor was able to achieve near-zero error rate using a pulse width of 75ms (Pixel 4A and Poco F1), corresponding to a \sim 10bps bit-rate. Both devices produced zero error using a 150ms width with a bit-rate at 5.1bps. The Moto G5 AC sensor, however, produced significant error (\sim 12 edit distance), even at long widths (e.g. 300ms). MF was the next best performing sensor, achieving error-free transmission for the Poco F1 (100ms, 9.62bps). However, the Pixel 4A required a longer width to achieve the same results (175ms, 5.08bps). The Pixel 4A’s GR sensor achieved near error-free transmission (0.215 average edit distance) with a 7.28bps bit rate using a 150ms width. The Poco F1 did not achieve this until a 350ms pulse width (2.81bps). In the best case, the RV sensor on the Pixel 4A achieved the lowest error rate of a 0.636 edit distance at 15ms (6.70bps). The LA sensor fared significantly worse, error-free transmission at 2.89bps was still achieved with the Pixel 4A.

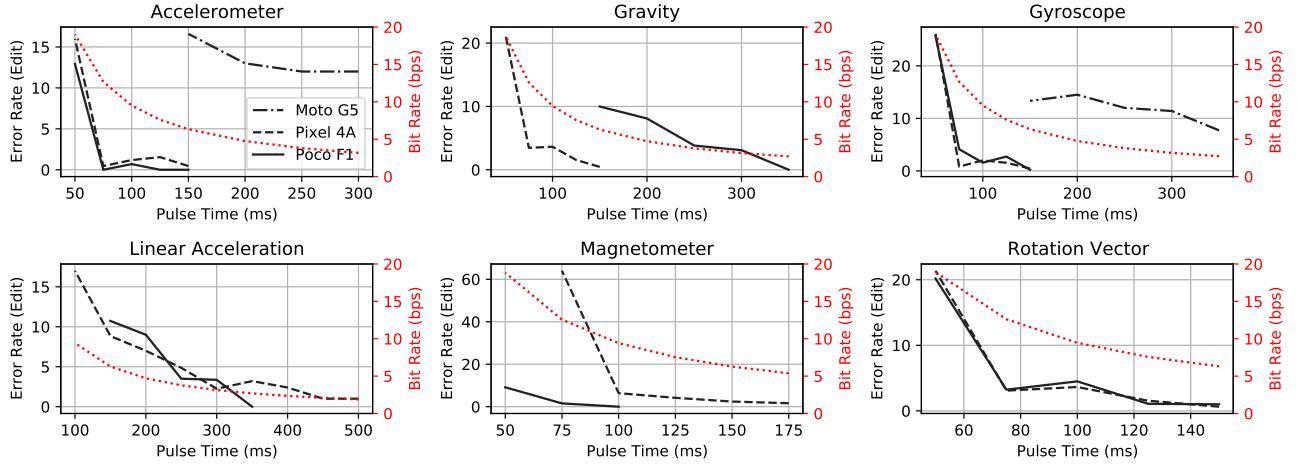


Figure 5: Average channel error rates and bit rates for each device and sensor.

Table 2: Best case error rates (average of 100 iterations).

Sensor	Pulse (ms)	Error (Edit)	Bit Rate (bps)	Device
AC	150	0	5.10	Poco F1 + Pixel 4A
GR	150	0	7.37	Pixel 4A
GY	150	0.215	7.28	Poco F1
LA	350	0	2.89	Poco F1
MF	100	0	9.62	Poco F1
RV	150	0.636	6.70	Pixel 4A

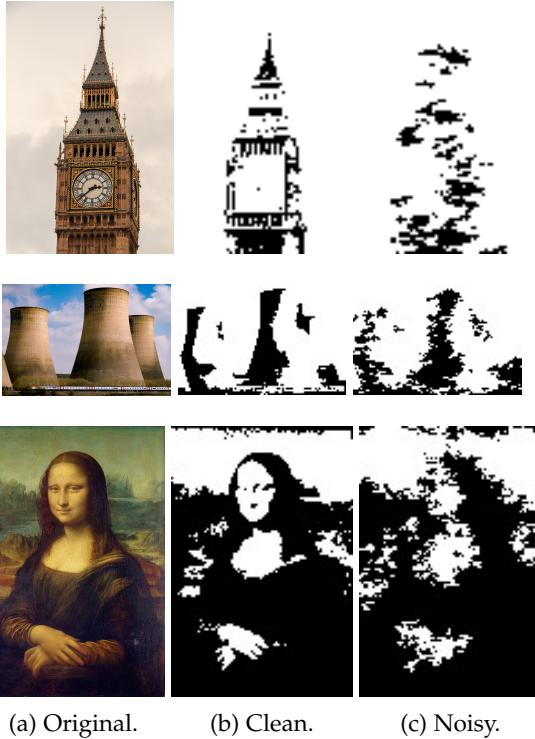


Figure 6: Example image transmissions. (Device: Xiaomi Poco F1; clean channel: MF at 150ms; noisy: GR at 250ms.)

Interestingly, the channel performed most effectively on the highest cost device (Pixel 4A). Conversely, the oldest, lowest cost handset (Moto G5) exhibited high error and had only two exploitable sensors. Error-free channels were not observed on this device. This leads to the conjecture that

handsets manufactured with higher quality component, are particularly susceptible to low error, high bit-rate covert channel transmissions. Beyond random bit-strings, we also investigated transferring images and numeric values. This better represents a realistic scenario where a disguised legitimate app leaks sensitive data to a receiver. Accordingly, we experimented with transferring photographs using a three-stage process: ① encoding the desired image using a one-bit colour palette; ② transmitting the encoded bit-string; and ③ exporting the bit-string in a suitable image format, e.g. PNG. Stages ① and ③ utilised a Python script using NumPy, SciPy, and the Python Image Library (PIL). Example image transfers are shown in Figure 6. We were also successful in transmitting GPS coordinates by encoding and segmenting decimal values as 4-bit strings. Communicating natural language is more difficult, e.g. leaking SMS records and instant messages. This requires a character encoding scheme that retains reasonable throughput. Given the low bit rate— $<10\text{bps}$ for near error-free transfers—transmitting individual characters take $\sim 1000\text{ms}$ using UTF-8, which limits its feasibility to short strings.

5 USING MULTIPLEXING FOR COARSE-GRAINED APPLICATION FINGERPRINTING

The last section showed how an application’s observed sensor sampling rate can be modulated by a second application. During the course of this work, a second attack vector was discovered that would enable a malicious application to learn information about a sensor-enabled victim app.

We identified that applications were likely to access certain sensors at particular sampling rates. For example, accelerometers are used ubiquitously as a stabilisation mechanism in games for aiming weapons, directing first-person viewpoints, and detecting screen orientation changes [4]. Based on this, we developed a two-part study to evaluate the extent to which a malicious application can detect sampling periods used by sensor-enabled victim apps. This consisted of, firstly, detecting Android Sensor SDK sampling period constants under different sensors and devices. The Android Sensors SDK provides developers with pre-defined

sampling periods in order to minimise battery consumption for several use cases [9]. These correspond to: gaming controllers (`SENSOR_DELAY_GAME`, 20ms); UI interactions, e.g. gesture recognition (`SENSOR_DELAY_UI`, 60ms); and detecting screen orientation (`SENSOR_DELAY_NORMAL`, 200ms). The fourth, `SENSOR_DELAY_FASTEST`, polls at the maximum supported frequency. In the second part of the study, we evaluated the extent to which sensor-based interactions can be detected using the top 250 Android applications [10].

5.1 Attack Design

The threat model assumes a malicious app, M , that wishes to detect the use of sensor-enabled games, UI interactions, or to directly identify a victim app, V . The goal is for M to perform this without requiring additional permissions that may reveal its intentions upon installation. The high-level attack approach follows three steps:

- 1) *Malicious Set-up*: M registers listeners for one or more sensors using their lowest supported sampling frequency simultaneously.
- 2) *Victim Execution*: Unwittingly, V registers another listener at another frequency for the same sensor, whether that is a pre-defined period, e.g. `SENSOR_DELAY_GAME`, or a custom frequency. Due to multiplexing, one of M 's signals is modulated to the higher frequency registered by V . V may use the received measurements for its intended purpose before de-registering the listener.
- 3) *Malicious Execution*: Concurrently, M detects V 's sampling period and the sensor used, and logs this information to file.

This process is illustrated in Figure 7, with an optional fourth state for recording the interaction. (Note: ① and ② are interchangeable. If ② is performed first, then the faster frequency used by V will already be returned to M when it requests values at an extremely slow frequency.)

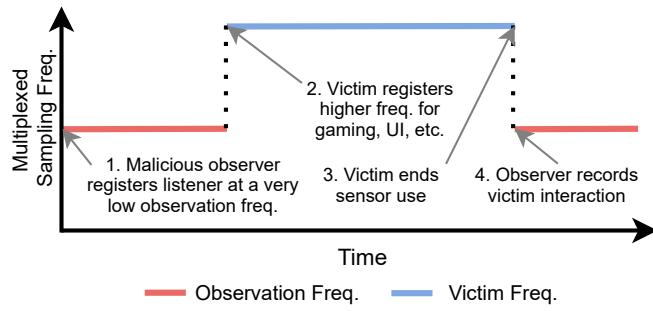


Figure 7: Application fingerprinting overview.

5.2 Implementation

The two apps from §4 were modified for inferring the victim sampling frequencies from a malicious observation app. Unlike §4, however, it cannot be assumed that a particular sensor will be used by a victim app *a priori*. This raises the question: how can M detect multiplexed sampling period changes of an arbitrary sensor used by V ? To address this, we instrumented M to register and receive measurements from all six continuous sensors concurrently (Listing 1). This

is valid behaviour under the Android framework. Note that the inferred periods of each sensor must be maintained independently.

In our test-bed, we implemented V to register a random sensor at one of the aforementioned pre-defined sampling constants from the Android Sensor SDK. This was triggered by user input. The time that V activated the sensor and the sampling period were stored for later analysis. Upon detection, M stored the detected sensor, inferred sampling period, and the associated timestamp.

```

1 final int[] sensors = {
2     Sensor.TYPE_ACCELEROMETER,
3     Sensor.TYPE_GRAVITY,
4     Sensor.TYPE_GYROSCOPE,
5     Sensor.TYPE_LINEAR_ACCELERATION,
6     Sensor.TYPE_MAGNETIC_FIELD,
7     Sensor.TYPE_ROTATION_VECTOR
8 };
9 for (int s : sensors) {
10     mSensorManager.registerListener(this,
11         mSensorManager.getDefaultSensor(s),
12         OBSERVATION_SAMPLING_PERIOD);
13 }
```

Listing 1: Java snippet for registering multiple sensors.

5.3 Evaluation

This section evaluates the accuracy and latency of our proposed fingerprinting method.

5.3.1 Methodology

Initial trials were conducted where M was set to run persistently in the background. This created signals for all six target sensors in parallel using their maximum supported frequencies. Next, V was opened for 25 trials per sensor, device, and sampling period. Like our covert channel, many of the actual inferred frequencies did not strictly reflect those specified in the Android SDK [7]. For two devices (Poco F1 and Pixel 4A), the observed periods still broadly reflected those specified Android SDK, albeit within a $\pm 10\%$ error band. Despite this, the observed rates were still individually distinguishable from a long-period carrier signal in most cases.

It also became apparent that M could not always discriminate between the signals transmitted by V on certain devices. That is, M could not distinguish between `SensorManager` sampling period constants used by V when multiplexed into M 's carrier signal. For instance, the `SENSOR_DELAY_NORMAL` period on the Xiaomi F1's AC and GY sensors is the maximum supported period. Consequently, these periods used by V cannot be distinguished from M 's carrier frequency. Moreover, the Moto G5's single-frequency reporting caused issues once more: the GR, LA, and RV sensors could not be distinguished from the maximum period. As a consequence, the sensors for this device were disregarded from further study. In total, 20/24 (83.3%) cases could be successfully detected on the Xiaomi Poco F1 and Google Pixel 4A. Only 6/20 (30%) of cases were detectable with the Motorola Moto G5. A breakdown of distinguishable cases is shown in Tables 3–5.

Table 3: Actual observed sampling periods (ms) for each SensorManager constant (*Xiaomi Poco F1*).

Sensor	Fastest	Game	UI	Normal	Max.
AC	2.484	19.83	66.67	198.6	198.6
GR	4.961	19.89	79.41	198.7	198.6
GY	4.968	19.87	66.67	198.6	198.6
LA	4.961	19.89	79.48	198.6	198.6
MF	10.0	20.00	66.67	200.0	1000
RV	4.166	16.64	55.59	166.6	833.3

Legend — Green: distinguishable using the max. period as the reference period; Red: indistinguishable.

Table 4: Actual observed sampling periods (ms) for each SensorManager constant (*Google Pixel 4A*).

Sensor	Fastest	Game	UI	Normal	Max.
AC	2.346	18.77	65.70	197.1	976.2
GR	4.693	18.77	65.70	197.1	197.1
GY	2.346	18.77	65.70	197.1	976.2
LA	14.08*	18.77	65.70	197.1	197.1
MF	9.956	19.58	66.67	195.9	979.5
RV	4.693	18.77	65.70	197.1	197.1

*: Erroneously returned the `SENSOR_DELAY_GAME` period when `SENSOR_DELAY_FASTEST` was used.

5.3.2 Identifying Android Sampling Constants

Following the initial results, we conducted further experiments of 100 additional trials per sensor, per sampling period, and per device. This produced 4,600 interactions in total between V and M . In total, M was able to correctly detect V 's chosen sensor and sampling period in all observed cases. The average detection latencies for each sensor, device, and constant are given in Figure 8. On average, a given sensor and sampling constant was detected by M in <500ms. Notably, the detection latency was reduced as V 's chosen sampling period was lower (faster). This is a natural result of our sampling period inferencing method, which uses sequential sensor events. When V requests measurements at a higher frequency, then measurements are multiplexed and inferred by M at a faster rate.

In the best cases, the detection latency was reduced to <80ms when the victim used the sensor's fastest sampling rate. In a small number of cases, detection latency reached under half (40ms), e.g. the Pixel 4A's AC and RV sensors. At the opposite end, V 's sampling period was inferred in ~400ms where `SENSOR_DELAY_NORMAL` was used. The `SENSOR_DELAY_UI` constant could be detected in <250ms for all devices and sensors, and <150ms for the Pixel 4A and Moto G5. Lastly, `SENSOR_DELAY_GAME` was detected in <100ms, and <60ms for the Pixel 4A and Moto G5 on average. In general, the detection latency is device- and sensor-dependent. The detection latency was largely consistent for the Pixel 4A. In contrast to the covert channel, the Moto G5 performed relatively well for the remaining sensors. The Poco F1 performed with reduced consistency; in particular, the GR and GY sensors required more time to detect V 's sampling period. Yet, in all cases, this could be achieved in under half a second on our test devices.

Table 5: Actual observed sampling periods (ms) for each SensorManager constant (*Motorola Moto G5*).

Sensor	Fastest	Game	UI	Normal	Max.
AC	9.934	19.98	59.68	198.6	198.6
GR	5.026*	9.893	9.941	9.890	9.915
GY	4.944	19.87	59.59	198.6	198.6
LA	4.958*	9.893	9.863	9.879	9.901
RV	4.950◊	9.906	10.03	9.926	9.918

*: Failed to return `SENSOR_DELAY_FASTEST` when multiplexed into the maximum period.

◊: GY was modulated when RV was activated. (Note: RV is a virtual sensor that uses GY as input.)

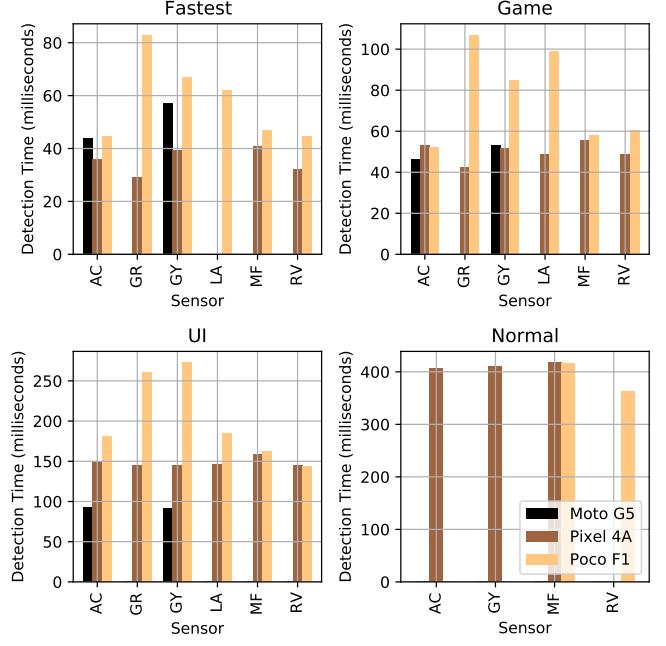


Figure 8: Average fingerprinting detection latency for each SensorManager constant, sensor, and device.

5.3.3 Evaluating Real-World Applications

The previous section showed how particular sampling constants used by a victim application can be identified by a malicious observer. To extend these results, we investigated the top 250 most popular Android applications (as of February 2022, according to AndroidRank [10]). In this study, the malicious application, M , was launched that registered listeners for all supported device sensors at the slowest possible frequency. Next, each application from the top 250 was launched. We proceeded beyond main menus, login dialogs, and other intermediate interfaces to reach the main activity where sensors were utilised (if any). If no sensors were used, then the application was discarded from further consideration. In cases where sensors were used, the interaction was logged to file by M , which was subsequently retrieved for off-line analysis.

In total, our malicious observer detected 57/250 apps (22.8%) where sensors were used. The majority of these were games (38; 66.7%), followed by food and drink (3; 5.2%), maps and navigation (3), photography (3), music and audio (2; 3.5%), travel and local (2), art and design (1; 1.7%),

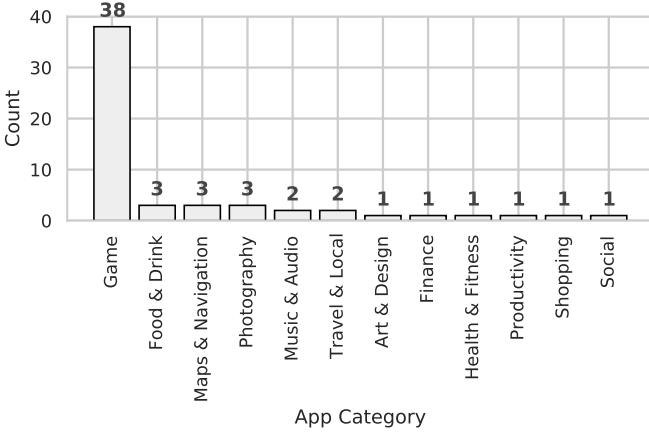


Figure 9: Category breakdown of detected applications.

finance (1), health and fitness (1), productivity (1), shopping (1), and social (1). This is illustrated in Figure 9. Comprehensive results are given in Appendix B in Table 9. Of the gaming applications, the most common sub-categories were action (18), racing (7), adventure (2), arcade (2), casual (2), trivia (2), sports (2), board (1), puzzle (1), and simulation (1).

Following our analysis, we found that the vast majority of apps used the AC sensor (53/57; 92.9%), primarily at the `SENSOR_DELAY_GAME` frequency (33/53; 62.3%). This is not surprising: accelerometers are supported by “*most*” Android devices for implementing tilting, shaking, rotation and swinging-based interactions, particularly in games [4]. What was surprising was the large number of sensors and non-standard sampling rates used by sophisticated apps, e.g. Pokémon Go—an augmented reality (AR) game by Niantic, Inc. This particular application used six different sensors—AC, GR, MF, LA, and RV—with a non-standard 10ms sampling period for the AC sensor. Indeed, this application used a large and unique combination of sensors and sampling rates. Interestingly, we found unique sensor-sampling combinations were used by 13/57 (22.8%) of detected apps. This information may enable a malicious actor to precisely fingerprint individual applications. Comparatively, the remaining applications used conflicting parameters (44/57; 77.2%). That is, the sensor(s) and sampling period(s) were used by at least one other application. While individual identification is challenging here, an attacker may still be able to deduce the victim’s high-level nature. For instance, we observed that all but one game (37/38; 97.4%) used the AC sensor at the `SENSOR_DELAY_GAME` frequency or greater. This increased to 100% of apps within some sub-categories (racing and simulation). In addition, we recognised that apps which relied on real-world navigation, e.g. location and heading, typically utilised both the AC and MF sensors at a high frequency (Gojek, Google Fit, Google Maps, Grab, Pokémon Go, Uber, Waze, and Yandex Go).

6 SECURITY EVALUATION

To counter security issues from mobile sensors (see §2), Android 9 introduced updates for visibly indicating when high-frequency, long-polling sensors were in use. In this section, these changes are examined and how they are

unsatisfactory towards addressing our attacks. We discuss UI-, API-, and system-level countermeasures, and offer recommendations.

6.1 Android 9 Changes

From Android 9 (API level 28), applications cannot use background services to access sensors in continuous reporting mode. If this is attempted, then sensor events are not returned to the requesting application. Instead, apps must use foreground services that are visible to the user using a taskbar notification. This is to raise the user’s awareness of on-going background processing. The notification is removed only when the service is stopped. Therefore, Android application manifest files must contain the `FOREGROUND_SERVICE` permission in order to access sensor services. Unfortunately, this leads to the first issue: `FOREGROUND_SERVICE` is a ‘normal’ permission. That is, according to the Android documentation, the associated “*data and actions present very little risk to the user’s privacy*” [5]. Android OS automatically grants normal permissions to requesting applications without user input. This compares with ‘dangerous’ permissions—e.g. accessing SMS messages, call logs, and location data—that requires user confirmation through a dialog prompt [3]. Put otherwise, a malicious application can access sensor services to mount our attacks without requiring explicit user approval.

Secondly, foreground service notifications are heavily customisable (Figure 10). The notification heading, text body, and icon can all be changed with minimal restrictions by a malicious developer. It is conceivable that an attacker could manipulate these properties in order to masquerade as a seemingly legitimate application. A potential UI countermeasure is to require applications to display the full set of active registered sensor listeners in their foreground service notification. This would indicate any unusual sensor usage that strays beyond the application’s expected remit. Yet, using the technical names of sensors is likely to prompt user confusion. Referring to the *class* of sensors in use, e.g. ‘motion’ or ‘position’, may be more appropriate. However, a usability investigation for indicating potentially dangerous sensors to users is warranted.

6.2 Recommendations

From the design of the Sensors SDK, a robust countermeasure is to remove the ability for application developers to control the sampling rate through the `registerListener` method of the Sensors SDK. Instead, the appropriate sampling rate could be inferred and returned to the application at an operating system level.

Recommendation 1: Consider removing the ability to directly set sensor sampling periods.

By adopting this measure, malicious apps are unable to establish carrier signals at a desired frequency, thereby preventing both presented attacks.

The root cause of both attacks is the design choice of using the fastest sampling rate when multiple apps request the same sensor at different frequencies. The ultimate solution is to enforce that the requested sampling period is the one that is received by an application. This is the case used by

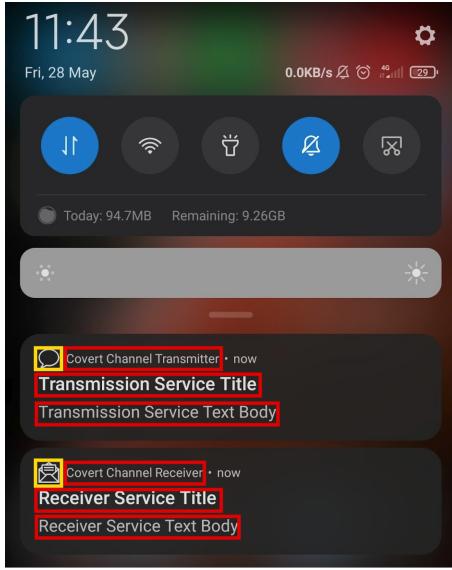


Figure 10: Covert channel foreground notifications. Yellow regions: customisable images; red: customisable text areas.

Apple iOS devices, which we were not able to successfully exploit in the same way. However, it requires changes to how the Android framework multiplexes sensor events to applications and services.

Recommendation 2: Enforce requested sampling periods on a per-app basis.

This prevents both attacks by stopping malicious apps from exploiting software multiplexing using higher frequency signals registered by other apps.

6.3 Limitations

The presented methods work optimally where malicious apps leverage unused shared sensors. The attacks cannot be mounted where a third application is already using a sensor at the highest frequency. This prevents the creation of appropriate frequency bands by the malicious application. In these cases, it makes sense for attackers to target rarely used sensors, such as gravity and linear acceleration sensors where available. It is possible for malicious apps to detect cases where a sensor(s) are already being used at its fastest rate. This can be achieved by registering multiple sensors, i.e. in Listing 1, and comparing the inferred rate with the highest supported frequency from the `SensorManager` class. It is also worth noting that our channel does not have the same throughput as lower-level channels, such as those using L3 cache activity (500 kbps [29] and 751 bps [36]), memory bus controllers (333 bps [44]), and DRAM channels (411 kbps [43]). However, our approach imposes very few assumptions, is CPU/SoC-agnostic, and uses pre-existing interfaces made available to application developers.

7 CONCLUSION

In this paper, we presented two software-controlled side-channel methods that arise from insecure design choices in mobile sensor stacks. Specifically, we show how software-based multiplexing can result in the creation of reliable

FSK-based covert channels and the ability to fingerprint sensor-enabled applications. Neither approach imposes special requirements—for example, a rooted handset or kernel-mode access—beyond the standard Sensor SDK made available to application developers.

In §4, a spectral covert channel was developed for transmitting arbitrary bit-strings between same-device applications in a way that bypassed IPC security mechanisms. This was achieved using a carrier frequency established by a receiver app, which was modulated by a transmitted app that requested a higher sensor sampling frequency. It was shown how many physical and virtual continuous sensors could be used for this purpose, including accelerometers, gyroscopes, and magnetic field sensors, with low error rates. Indeed, the transmission of low-resolution yet legible single-bit images was possible at up to 20 bps.

In §5, a variant of this technique was developed for fingerprinting sensor-enabled applications at a coarse-grained level. This involved a malicious app that generated multiple, simultaneous sensor signals using their lowest supported frequency. Victim apps which used the same sensor(s) at a higher frequency thus triggered modulations in this signal. It was shown how this could be used for detecting particular interactions (e.g. UI interactions, screen orientation changes, and gaming activity) with low latency (30–400ms). Moreover, an analysis of the top 250 Android applications showed the activity of 57 (22.8%) were detected using our approach. It was also shown how some highly downloaded applications exhibited unique fingerprints. This was due to the use of particular sensor and sampling frequency combinations, which could be passively detected by a malicious application.

For both approaches, all evaluation devices from different OEMs were vulnerable, potentially affecting millions of devices worldwide. In §6, we suggested two recommendations: ① removing the ability for developers to set precise sensor sampling periods; and ② enforcing the requested sampling periods by removing the multiplexing functionality. Unfortunately, both solutions require design changes to the Android sensor stack.

ACKNOWLEDGMENTS

This work received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No. 883156 (EXFILES).

REFERENCES

- [1] Android Open Source Project. Batching, 2021. <https://source.android.com/devices/sensors/batching>.
- [2] Android Open Source Project. Fast message queues (FMQs), 2021. <https://source.android.com/devices/architecture/hidl/fmq>.
- [3] Android Open Source Project. Manifest.permission, 2021. <https://developer.android.com/reference/android/Manifest.permission>.
- [4] Android Open Source Project. Motion sensors, 2021. https://developer.android.com/guide/topics/sensors/sensors_motion.
- [5] Android Open Source Project. Permissions on Android, 2021. <https://developer.android.com/guide/topics/permissions/overview#normal-dangerous>.
- [6] Android Open Source Project. Sensor HAL2, 2021. <https://source.android.com/devices/sensors/sensors-hal2>.
- [7] Android Open Source Project. Sensor overview, 2021. https://developer.android.com/guide/topics/sensors/sensors_overview.

- [8] Android Open Source Project. Sensor stack, 2021. <https://source.android.com/devices/sensors/sensor-stack>.
- [9] Android Open Source Project. SensorManager, 2021. <https://developer.android.com/reference/android/hardware/SensorManager>.
- [10] AndroidRank. Open Android market data, 2022. <https://www.androidrank.org/>.
- [11] Davide B Bartolini, Philipp Miedl, and Lothar Thiele. On the capacity of thermal covert channels in multicores. In *Proceedings of the 11th European Conference on Computer Systems*, pages 1–16, 2016.
- [12] Kenneth Block, Sashank Narain, and Guevara Noubir. An autonomic and permissionless android covert channel. In *Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pages 184–194, 2017.
- [13] Serdar Cabuk, Carla E Brodley, and Clay Shields. IP covert timing channels: Design and detection. In *11th ACM Conference on Computer and Communications Security*, pages 178–187, 2004.
- [14] Liang Cai and Hao Chen. Touchlogger: Inferring keystrokes on touch screen from smartphone motion. *HotSec*, 11(2011):9, 2011.
- [15] Haehyun Cho, Penghui Zhang, Donguk Kim, Jinbum Park, Choong-Hoon Lee, Ziming Zhao, Adam Doupé, and Gail-Joon Ahn. Prime+Count: Novel cross-world covert channels on ARM TrustZone. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 441–452, 2018.
- [16] John P. Doran. *Unity 2017 Mobile Game Development*. Packt Publishing, 2017.
- [17] Mayank Goel, Leah Findlater, and Jacob Wobbrock. WalkType: using accelerometer data to accomodate situational impairments in mobile touch screen text entry. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2687–2696, 2012.
- [18] Mayank Goel, Jacob Wobbrock, and Shwetak Patel. GripSense: using built-in sensors to detect hand posture and pressure on commodity mobile phones. In *25th ACM Symposium on User Interface Software and Technology*, pages 545–554, 2012.
- [19] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: a fast and stealthy cache attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 279–299. Springer, 2016.
- [20] Hari Prabhat Gupta, Haresh S Chudgar, Siddhartha Mukherjee, Tania Dutta, and Kulwant Sharma. A continuous hand gestures recognition technique for human-machine interaction using accelerometer and gyroscope sensors. *IEEE Sensors Journal*, 16(16):6425–6432, 2016.
- [21] Iakovos Gurulian, Carlton Shepherd, Eibe Frank, Konstantinos Markantonakis, Raja Naeem Akram, and Keith Mayes. On the effectiveness of ambient sensing for detecting NFC relay attacks. In *16th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, pages 41–49. IEEE, 2017.
- [22] Sojeong Ha and Seungjin Choi. Convolutional neural networks for human activity recognition using multiple accelerometer and gyroscope sensors. In *International Joint Conference on Neural Networks*, pages 381–388. IEEE, 2016.
- [23] Tzipora Halevi, Di Ma, Nitesh Saxena, and Tuo Xiang. Secure proximity detection for NFC devices based on ambient sensor data. In *European Symposium on Research in Computer Security*, pages 379–396. Springer, 2012.
- [24] Zhen Hang Jiang, Yunsi Fei, and David Kaeli. A complete key recovery timing attack on a GPU. In *IEEE International Symposium on High Performance Computer Architecture*, pages 394–405. IEEE, 2016.
- [25] Robert Krösche, Kashyap Thimmaraju, Liron Schiff, and Stefan Schmid. I DPID it my way! A covert timing channel in software-defined networks. In *IFIP Networking Conference and Workshops*, pages 217–225. IEEE, 2018.
- [26] Ben Lapid and Avishai Wool. Cache-attacks on the ARM TrustZone implementations of AES-256 and AES-256-GCM via GPU-based analysis. In *International Conference on Selected Areas in Cryptography*, pages 235–256. Springer, 2018.
- [27] Yuqi Lin, Saif UR Malik, Kashif Bilal, Qiusong Yang, Yongji Wang, and Samee U Khan. Designing and modeling of covert channels in operating systems. *IEEE Transactions on Computers*, 65(6):1706–1719, 2015.
- [28] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. Armageddon: Cache attacks on mobile devices. In *25th USENIX Security Symposium (USENIX Security)*, pages 549–564, 2016.
- [29] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *IEEE Symposium on Security and Privacy*, pages 605–622. IEEE, 2015.
- [30] Zijun Long, Xiaohang Wang, Yingtao Jiang, Guofeng Cui, Li Zhang, and Terrence Mak. Improving the efficiency of thermal covert channels in multi-/many-core systems. In *Design, Automation and Test in Europe*, pages 1459–1464. IEEE, 2018.
- [31] Anindya Maiti, Oscar Armbruster, Murtuza Jadliwala, and Jibo He. Smartwatch-based keystroke inference attacks and context-aware protection mechanisms. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 795–806, 2016.
- [32] Philip Marquardt, Arunabh Verma, Henry Carter, and Patrick Traynor. (sp)iPhone: Decoding vibrations from nearby keyboards using mobile phone accelerometers. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 551–562, 2011.
- [33] Ramya Jayaram Masti, Devendra Rai, Aanjhan Ranganathan, Christian Müller, Lothar Thiele, and Srdjan Capkun. Thermal covert channels on multi-core platforms. In *24th USENIX Security Symposium (USENIX Security)*, pages 865–880, 2015.
- [34] Nikolay Matyunin, Jakub Szefer, Sebastian Biedermann, and Stefan Katzenbeisser. Covert channels using mobile device’s magnetic field sensors. In *21st Asia and South Pacific Design Automation Conference*, pages 525–532. IEEE, 2016.
- [35] Nikolay Matyunin, Yujue Wang, Tolga Arul, Kristian Kullmann, Jakub Szefer, and Stefan Katzenbeisser. MagneticSpy: Exploiting magnetometer in mobile devices for website and application fingerprinting. In *Proceedings of the 18th ACM Workshop on Privacy in the Electronic Society*, pages 135–149, 2019.
- [36] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. C5: Cross-cores cache covert channel. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 46–64. Springer, 2015.
- [37] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. Hello from the other side: SSH over robust cache covert channels in the cloud. In *Network and Distributed System Security Symposium*, volume 17, pages 8–11, 2017.
- [38] Yan Michalevsky, Dan Boneh, and Gabi Nakibly. Gyrophone: Recognizing speech from gyroscope signals. In *23rd USENIX Security Symposium (USENIX Security)*, pages 1053–1067, 2014.
- [39] Hoda NaghibiJouybari, Ajaya Neupane, Zhiyun Qian, and Nael Abu-Ghazaleh. Rendered insecure: GPU side channel attacks are practical. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 2139–2153, 2018.
- [40] Ed Novak, Yutao Tang, Zijiang Hao, Qun Li, and Yifan Zhang. Physical media covert channels on smart mobile devices. In *Proceedings of the ACM International Joint Conference on Pervasive and Ubiquitous Computing*, pages 367–378, 2015.
- [41] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of aes. In *Cryptographers’ Track at the RSA Conference*, pages 1–20. Springer, 2006.
- [42] Emmanuel Owusu, Jun Han, Sauvik Das, Adrian Perrig, and Joy Zhang. Accessory: Password inference using accelerometers on smartphones. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, pages 1–6, 2012.
- [43] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM addressing for cross-CPU attacks. In *25th USENIX security symposium (USENIX security 16)*, pages 565–581, 2016.
- [44] Benjamin Semal, Konstantinos Markantonakis, Raja Naeem Akram, and Jan Kalbantner. Leaky controller: Cross-VM memory controller covert channel on multi-core systems. In *IFIP International Conference on ICT Systems Security and Privacy Protection, IFIP SEC*, pages 3–16. Springer, 2020.
- [45] Thomas Ulz, Markus Feldbacher, Thomas W Pieber, and Christian Steger. Sensing danger: Exploiting sensors to build covert channels. In *International Conference on Information Systems Security and Privacy, ICISPP*, pages 100–113, 2019.
- [46] Xian Wang, Paula Tarrio, Eduardo Metola, Ana M Bernardos, and José R Casar. Gesture recognition using mobile phone’s inertial sensors. In *Distributed Computing and Artificial Intelligence*, pages 173–184. Springer, 2012.
- [47] Qiumin Xu, Hoda NaghibiJouybari, Shibo Wang, Nael Abu-Ghazaleh, and Murali Annavaram. GPUGuard: mitigating contention based side and covert channel attacks on GPUs. In

- Proceedings of the ACM International Conference on Supercomputing*, pages 497–509, 2019.
- [48] Zhi Xu, Kun Bai, and Sencun Zhu. Taplogger: Inferring user inputs on smartphone touchscreens using on-board motion sensors. In *Proceedings of the 5th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pages 113–124, 2012.
- [49] Yuval Yarom and Katrina Falkner. Flush+Reload: A high resolution, low noise, L3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security)*, pages 719–732, 2014.
- [50] Ning Zhang, Kun Sun, Deborah Shands, Wenjing Lou, and Y Thomas Hou. Truspy: Cache side-channel information leakage from the secure world on ARM devices. *IACR Cryptol. ePrint Arch.*, 2016:980, 2016.
- [51] Xiaokuan Zhang, Yuan Xiao, and Yingqian Zhang. Return-oriented flush-reload side channels on ARM and their implications for Android devices. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 858–870, 2016.



Carlton Shepherd received his Ph.D. in Information Security from the Information Security Group at Royal Holloway, University of London, U.K., and his B.S. in Computer Science from Newcastle University, U.K. He is currently a Senior Research Fellow at the Information Security Group at Royal Holloway, University of London. His research interests centre around the security of trusted execution environments (TEEs) and their applications, secure CPU design, and embedded systems.



Jan Kalbantner received his B.S. in Computer Science from Cooperative State University BW, Mosbach, Germany and his M.S. in Information Security from Royal Holloway, University of London, U.K., in 2016 and 2019 respectively. Since 2019, he is working towards his Ph.D. degree at Royal Holloway, University of London. His research interests include cyber-physical systems, DLT, network security, and secure architectures.



Benjamin Semal received his M.Eng. in Electrical Engineering from Ecole Polytechnique Universitaire of Montpellier, France and M.S. in robotics from Cranfield University, U.K. He then worked as a hardware security analyst at UL Transaction Security. He later joined Royal Holloway, University of London to pursue a Ph.D. in Information Security. His research focusses on side-channel attacks for information leakage in multi-tenant environments. Benjamin now works as a security engineer at SERMA Security & Safety evaluating point-of-sale devices and cryptographic modules.



Konstantinos Markantonakis received his B.S. in Computer Science from Lancaster University, U.K.; and his M.S. and Ph.D. in Information Security, and M.B.A. in International Management from Royal Holloway, University of London, London, UK. He is currently the Director of the Smart Card and IoT Security Centre. He has co-authored over 190 papers in international conferences and journals. His research interests include smart card security, trusted execution environments, and the Internet of Things.

APPENDIX A COVERT CHANNEL SAMPLING PARAMETERS

This appendix provides the precise parameters used for each device and sensor for establishing FSK-based covert channels. These are shown in Tables 6–8.

Table 6: Xiaomi Poco F1 channel sampling periods (μ s).

Sensor	T_c	T_{tr}	T_{sync}	T_{end}
AC	10000	7500	5000	2500
GR	40000	20000	10000	5000
GY	20000	15000	10000	5000
LA	40000	20000	10000	5000
MF	25000	20000	15000	10000
RV	20000	15000	10000	5000

Table 7: Google Pixel 4A channel sampling periods (μ s).

Sensor	T_c	T_{tr}	T_{sync}	T_{end}
AC	10000	7500	5000	2500
GR	20000	15000	10000	5000
GY	10000	7500	5000	2500
LA	50000	40000	30000	20000
MF	25000	20000	15000	10000
RV	20000	15000	10000	5000

Table 8: Motorola Moto G5 channel sampling periods (μ s).

Sensor	T_c	T_{tr}	T_{sync}	T_{end}
AC	40000	30000	20000	10000
GY	30000	20000	10000	5000

APPENDIX B FINGERPRINTED APPLICATIONS

Successfully fingerprinted applications, taken from the top 250 Android applications [10] (as of February 2022), are shown in Table 9.

Table 9: Detected apps, category, triggered sensor(s) and sampling period(s).

Application	Category	Detected Sensors and Sampling Periods
<i>AliExpress</i>	Shopping	AC (SENSOR_DELAY_UI)
<i>Among Us</i>	Game (Action)	AC (SENSOR_DELAY_GAME)
<i>Ashphalt 8</i>	Game (Racing)	AC (SENSOR_DELAY_GAME)
<i>Banana Kong</i>	Game (Action)	AC (SENSOR_DELAY_GAME)
<i>B612 Photo/Video Editor</i>	Photography	AC (SENSOR_DELAY_UI)
<i>Brain Out</i>	Game (Puzzle)	AC, GR, GY, MF, RV (SENSOR_DELAY_GAME)
<i>Brain Test</i>	Game (Trivia)	AC (SENSOR_DELAY_GAME)
<i>Beauty Plus</i>	Photography	RV (SENSOR_DELAY_UI)
<i>CamScanner</i>	Productivity	AC (SENSOR_DELAY_UI)
<i>Canva</i>	Art & Design	AC, GY, MF (SENSOR_DELAY_UI)
<i>Call of Duty Mobile</i>	Game (Action)	AC (SENSOR_DELAY_GAME), GY (SENSOR_DELAY_NORMAL)
<i>Coin Master</i>	Game (Casual)	AC (SENSOR_DELAY_GAME)
<i>Crossy Road</i>	Game (Action)	AC (SENSOR_DELAY_GAME)
<i>CSR Racing 2</i>	Game (Racing)	AC (SENSOR_DELAY_GAME)
<i>Dr. Driving</i>	Game (Racing)	AC (SENSOR_DELAY_GAME)
<i>eFootball PES 2021</i>	Game (Sports)	AC, GY, MF (SENSOR_DELAY_GAME)
<i>Extreme Car Driving</i>	Game (Racing)	AC (SENSOR_DELAY_GAME)
<i>Gaana Music App</i>	Music & Audio	AC (SENSOR_DELAY_UI)
<i>Garena Free Fire</i>	Game (Action)	AC (SENSOR_DELAY_GAME)
<i>Gojek</i>	Travel & Local	AC (10ms), MF (SENSOR_DELAY_GAME)
<i>Google Fit</i>	Health & Fitness	AC (10ms), MF (SENSOR_DELAY_GAME)
<i>Google Maps</i>	Travel & Local	AC (10ms), GR, LA, MF (SENSOR_DELAY_GAME)
<i>Grab</i>	Food & Drink	AC (10ms), MF (SENSOR_DELAY_GAME)
<i>Granny</i>	Game (Arcade)	AC (SENSOR_DELAY_GAME)
<i>Helix Jump</i>	Game (Action)	AC (SENSOR_DELAY_GAME)
<i>Hungry Shark Evolution</i>	Game (Action)	AC (SENSOR_DELAY_GAME)
<i>Idle Minor Tycoon</i>	Game (Simulation)	AC (SENSOR_DELAY_GAME)
<i>iFood Delivery de Comida</i>	Food & Drink	AC (SENSOR_NORMAL_UI)
<i>Last Day on Earth: Survival</i>	Game (Action)	AC (SENSOR_DELAY_GAME)
<i>Ludo King</i>	Game (Board)	AC (SENSOR_DELAY_GAME)
<i>Mobile Legends</i>	Game (Action)	AC (15ms)
<i>Modern Combat 5</i>	Game (Action)	GY (15ms)
<i>Mortal Kombat</i>	Game (Action)	AC, GY, MF (SENSOR_DELAY_GAME)
<i>My Talking Tom</i>	Game (Casual)	AC (SENSOR_DELAY_GAME)
<i>Need for Speed: No Limits</i>	Game (Racing)	AC (SENSOR_DELAY_GAME)
<i>PK XD</i>	Game (Adventure)	AC (SENSOR_DELAY_GAME)
<i>Pokémon Go</i>	Game (Adventure)	AC (10ms), GR, GY, MF, LA, RV (SENSOR_DELAY_GAME)
<i>PUBG Mobile</i>	Game (Action)	AC, GY (SENSOR_DELAY_GAME)
<i>Real Racing 3</i>	Game (Racing)	AC (SENSOR_DELAY_GAME)
<i>Sberbank Online</i>	Finance	AC (23ms)
<i>Shadow Fight 2</i>	Game (Action)	AC (SENSOR_DELAY_GAME)
<i>Shadow Fight 3</i>	Game (Action)	AC (SENSOR_DELAY_GAME)
<i>Smule</i>	Music & Audio	AC (SENSOR_DELAY_UI)
<i>Sniper 3D</i>	Game (Action)	AC (SENSOR_DELAY_GAME)
<i>Special Forces Group 2</i>	Game (Action)	AC, GY (SENSOR_DELAY_GAME), MF (SENSOR_DELAY_NORMAL)
<i>Stand Off 2</i>	Game (Action)	AC (SENSOR_DELAY_GAME)
<i>Subway Surfers</i>	Game (Arcade)	AC (SENSOR_DELAY_GAME)
<i>Tango</i>	Social	RV (SENSOR_DELAY_NORMAL)
<i>Top Eleven Soccer Manager</i>	Game (Sports)	AC (SENSOR_DELAY_GAME)
<i>Traffic Rider</i>	Game (Racing)	AC (SENSOR_DELAY_GAME)
<i>Trivia Crack</i>	Game (Trivia)	AC (SENSOR_DELAY_UI)
<i>Uber</i>	Maps & Navigation	AC (10ms), MF (SENSOR_DELAY_GAME)
<i>Uber Eats</i>	Food & Drink	RV (SENSOR_DELAY_NORMAL)
<i>War Robots Multiplayer</i>	Game (Action)	AC (SENSOR_DELAY_GAME)
<i>Waze</i>	Maps & Navigation	AC, GR, LA (SENSOR_DELAY_GAME), MF (SENSOR_DELAY_UI)
<i>Yandex Go</i>	Maps & Navigation	AC, MF (SENSOR_DELAY_UI)
<i>YouCam Makeup</i>	Photography	AC (SENSOR_DELAY_UI)

Green rows: Unique sensor/sampling combinations; Yellow: Conflicting parameters with another app.