Bypassing Android isolation with fuel gauges: new risks with advanced power ICs

Vincent Giraud[1,2] and David Naccache[1,2]

[1] DIENS, École Normale Supérieure, Université PSL, CNRS, Paris, France
[2] Ingenico, Suresnes, France
firstname.lastname@ens.fr

Abstract. Efficient power management is critical for embedded devices, both for extending their lifetime and ensuring safety. However, this can be a challenging task due to the unpredictability of the batteries commonly used in such devices. To address this issue, dedicated Integrated Circuits (ICs) known as "fuel gauges" are often employed outside of the System-on-Chip (SoC). These devices provide various metrics about the available energy source and are highly accurate. However, their precision can also be exploited by malicious actors to compromise platform confidentiality if the Operating System (OS) fails to intervene. Depending on the fuel gauge and OS configuration, several attack scenarios are possible. In this article, we focus on Android and demonstrate how it is possible to bypass application isolation to recover Personal Identification Numbers (PINs) entered in other processes.

Keywords: Fuel gauge · Embedded system · Confidentiality.

1 Introduction

Lithium-based batteries have been the go-to choice for embedded devices for several decades. These batteries offer high energy density and low self-discharge, and do not suffer from memory effect. However, predicting and analyzing their behavior can be a challenging task. The voltage at their poles is not directly proportional to the remaining energy level, and their discharge is affected by various factors such as the platform's dynamic consumption, temperature, age, and total capacity. Consequently, managing power on an embedded system is a complex process. Furthermore, end-user expectations have evolved significantly in recent years. Knowing a battery's charge level only up to the nearest quarter is no longer acceptable, as users now expect to have an estimate up to the nearest percentage.

Managing power consumption on embedded devices can be a challenging responsibility, requiring a significant investment of time and expertise. One approach is to implement the necessary operations and modeling at the OS level, or at least execute them on the central processor. However, this approach can impose an additional burden on an already heavily utilized component. Furthermore, obtaining reliable measurements from this environment, such as capturing

accurate temperature readings, can be complicated. This is because the processor itself can significantly influence the readings, rather than the battery. Additionally, it becomes more difficult to estimate the quality and age of the power source with this implementation. Managing multiple distinct batteries can exacerbate these challenges further.

To facilitate power management in embedded devices, many designers incorporate fuel gauges, which are integrated circuits dedicated to analyzing and monitoring various metrics related to the energy source. These metrics include the voltage of the source, the current drawn from or injected into it, and the temperature, among others. However, an overlooked aspect of these components that is worthy of interest in the field of security is their remarkable accuracy [3]. This accuracy allows fuel gauges to produce precise estimations of the battery's age, charge, and health, which are among their flagship features.

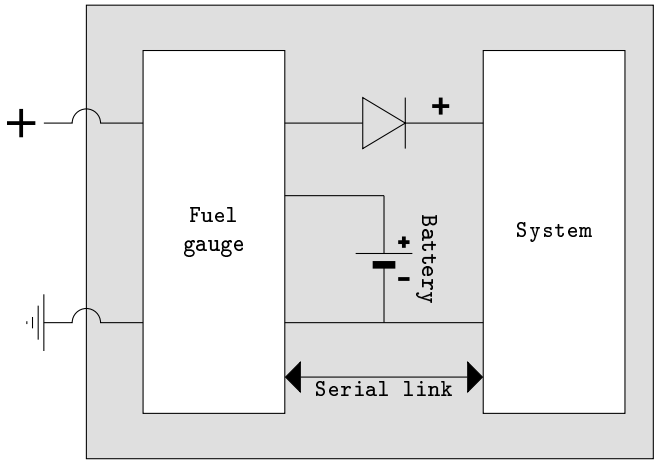


Fig. 1: Diagram representing a typical fuel gauge implementation in an embedded system.

The use of integrated fuel gauges in embedded devices frees device designers and OS developers from the responsibility of managing the power source. Fuel gauges provide more accurate measurements because they are located closer to the battery and handle the necessary calculations and algorithms. The software running on the SoC only needs to request the desired metrics or data, which are communicated through a serial link connecting the fuel gauge and the main system, as illustrated in Figure 1. This communication channel typically corresponds to an I^2C communication bus, which is managed by a driver residing in the kernel space. This delegation of responsibility is common in smartphones, tablets, and portable video game consoles, particularly in high-end products. However, since fuel gauges can be expensive, it should be noted that less accu-

rate measurements and estimates are often used in devices aimed at more modest price ranges.

Fuel gauge presence Determining whether a phone or tablet is equipped with a fuel gauge before purchase can be a challenging task, as manufacturers do not indicate the presence or absence of this component on their data sheets or documentation. However, after purchasing the device, one can visually inspect its printed circuit to confirm the presence of a fuel gauge. In the case of an Android device, it is possible to determine the presence of a fuel gauge in the system without necessarily having root access by probing the equipment related to power through a terminal.

Listing 1.1: Terminal output when determining the power-related equipments in the Pixel 6 smartphone.

```
$ ls -a /sys/class/power_supply
battery
dc
gcpm
gcpm_pps
main-charger
maxfg
pca9468-mains
tcpm-source-psy-i2c-max77759tcpc
usb
wireless
```

Upon inspection of the resulting list, fuel gauges are often found in Google's Nexus and Pixel lines, particularly in the Pixel 6, as illustrated in listing 1.1. While the number of listed elements may be substantial, the nomenclature can assist in identifying the IC of interest. In this specific case, the `maxfg` device draws our attention: `max` signifies the Maxim Integrated brand, while `fg` denotes the fuel gauge.

1.1 Software context in Android

The Android OS is based on a Linux kernel, with a user environment that is radically different from the ones usually found in conventional computer distributions. A decisive choice in the design of Android was to assign each application a different Unix user, thus allowing the system to benefit from the isolation traditionally imposed between processes. The SELinux module is deployed from version 4.3 to reinforce this policy. Outside the kernel space, on top of a minimal layer of libraries and native executables, Zygote acts as a model process for

instantiating applications: at each such request, it forks itself, and changes the user associated with the child process to respect the paradigm mentioned. The abstraction layers present on an Android system are illustrated in a simplified way in Figure 2.

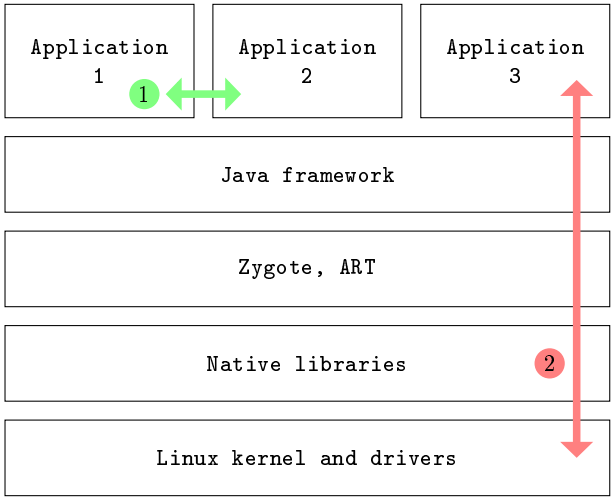


Fig. 2: Simplified view of the abstraction model in Android systems. The first, green arrow represents horizontal accesses. The second, red arrow represents vertical accesses.

At the application level, interactions between applications in Android are limited, and they are not possible directly. Instead, communications or calls between applications must be negotiated via Binder, the inter-process communication manager specific to Android[3]. This component takes care of calls between services and activities, which is a cardinal aspect of the Android operating system. While horizontal accesses, which refer to interactions between applications, are dictated by explicit rules, this is less the case for vertical interactions, where an application requires hardware resources provided by the platform. In the case of smartphones and tablets, these resources may include components such as a light meter, an accelerometer, a gyroscope, a microphone, and one or more cameras. Access to these features is regulated according to the Android version and the platform manufacturer, and it depends on the nature of the resource and the type of interaction desired. From an application's perspective, these permissions are often discovered at runtime. Access to energy monitoring is regulated

[3] Binder is not documented, but it can be found in Android's common kernel tree's drivers: `https://android.googlesource.com/kernel/common/+/refs/heads/android12-5.10/drivers/android/binder.c`

according to the same logic. The possibility of a risk resting on it has prompted us to explore this area further.

1.2 Key issues and contribution

Due to their possible presence in an essential brick of embedded systems, it is advisable to perform a risk analysis concerning fuel gauges. The state of the art of security assessment around these integrated circuits is non-existent. Although fuel gauges do not have control over the supply of electricity, unlike Power Management Integrated Circuits (PMICs) (although they may be included in a PMIC), there is reason to consider privacy risks, especially on platforms such as phones and tablets, which can contain a substantial amount of personal information. This is because fuel gauges can expose particularly precise measurements [3]. To mitigate these risks, it is essential to understand the possible attack scenarios and the potential impact of an attack. One approach is to perform a threat modeling exercise to identify potential attackers and their motivations, as well as the potential vulnerabilities of the system. This can help inform the selection of appropriate security controls, such as encryption and access controls, to protect the system against unauthorized access or disclosure.

Contribution In this article, we focus on embedded systems featuring the Android OS. We summarize the evolution of the access policy to hardware power sensors and outline the consequences of an unsuitable policy. By demonstrating that information about a PIN code can be recovered while it is being typed, we show that concrete and actual risks exist. We also address the question of what measures can be put in place to confront this security hazard. In Section 2, we describe the security policy in Android regarding hardware sensors and the risks it can create. In Section 3, we explain how we managed to exploit these risks. In Section 4, we discuss the implications of our findings. Finally, in Section 5, we provide closing remarks on the importance of addressing these security risks in embedded systems.

2 Risk analysis

2.1 Interactions between systems and their fuel gauge

The `BatteryManager` system service has existed since the early days of Android. At the beginning, it only allowed to know the status of the battery regarding its health (`GOOD`, `OVERHEAT`, `DEAD`, `OVER_VOLTAGE`...) or its use (`CHARGING`, `DISCHARGING`, `FULL`...), as well as, if applicable, the charging source (`USB` or `AC`). It has been expanded over time, until in version 5.0 (called *Lollipop*), constants were added to form queries that can be redirected to a fuel gauge[4]. These include `CURRENT_NOW` to obtain the instantaneous current entering or leaving the

[4] See: `https://android.googlesource.com/platform/frameworks/base/+/refs/heads/lollipop-release/core/java/android/os/BatteryManager.java`

battery in microamps, `CAPACITY` for the remaining capacity in percentage, and `ENERGY_COUNTER` for the remaining energy in nanowatt-hours.

Technically, any application running on an Android system can access the `BatteryManager` service and request any of the available attributes during its execution. The information in question is retrieved by probing the lower abstraction layers and, possibly, by consulting the fuel gauge. As a first step, we investigated whether any controls were in place in any of the layers of the system. We found that the SELinux configuration did not enforce any restrictions, although it could have. Other layers that are prone to this type of moderation include the Android framework in Java or the ART virtual machine, but no such measures were found here either. As expected, the native executables and libraries on the system did not block these requests either. After checking on version 12 and earlier, we can confirm that Android does not block these requests, regardless of the client application or the requested attribute.

This can already be a problem for the end-user, since he can't object to the sharing of power data. When an application puts in place the technical means to retrieve it, even if for legitimate purposes such as energy saving, one can then question the real use that is made of it. The company Uber, which was the target of such suspicions in 2016, had to publicly deny this kind of exploitation[5].

Incidentally, we should also note that some web browsers allow the Javascript code delivered by some sites to consult the status of the battery and its charge, via an interface of the same name, `BatteryManager`. The Javascript engine then transmits the request following the same procedure as any other application.

Another aspect that requires special attention is the ability to capture these measurements at any time, including when other applications are in use, or when the phone is in sleep mode. Since Android 9 (known as *Pie*), there is a `FOREGROUND_SERVICE` permission[6], required by the activity manager when an application requests to run a task normally in the background. Here too, this one is granted without any request from the user. However, to obtain it, you must have a notification in the list dedicated to this purpose in the system's graphical interface. There are now many applications that require a permanent notification, so as not to be sacrificed by the battery saver, or to be able to receive communications directly without going through the Google services. This could be a case of spoofing, where an application that is supposed to be for chatting or playing a video game is actually probing the fuel gauge in the background.

The next consideration is how often the fuel gauge can be checked. From Android 12 onwards, the `HIGH_SAMPLING_RATE_SENSORS` permission has appeared in the Java framework[7]. This permission is intended to limit scans above 200 Hz.

[5] See: `https://www.forbes.com/sites/amitchowdhry/2016/05/25/uber-low-battery/`

[6] See: `https://android.googlesource.com/platform/frameworks/base/+/refs/heads/pie-release/services/core/java/com/android/server/am/ActiveServices.java`

[7] See: `https://android.googlesource.com/platform/frameworks/base/+/refs/heads/android12-release/core/java/android/hardware/SystemSensorManager.java`

However, since it is a normal permission, it can be requested without visual warning to the user. Moreover, it does not concern fuel gauges anyway, as shown in the extract in Listing 1.2. In versions prior to 12, this measure does not exist.

Listing 1.2: Extract of Android's system sensor manager since version 12.

```
/**
 * Checks if a sensor should be capped according to
 * HIGH_SAMPLING_RATE_SENSORS permission.
 *
 * This needs to be kept in sync with the list defined on the native side
 * in frameworks/native/services/sensorservice/SensorService.cpp
 */
private boolean isSensorInCappedSet(int sensorType) {
    return (sensorType == Sensor.TYPE_ACCELEROMETER
            || sensorType == Sensor.TYPE_ACCELEROMETER_UNCALIBRATED
            || sensorType == Sensor.TYPE_GYROSCOPE
            || sensorType == Sensor.TYPE_GYROSCOPE_UNCALIBRATED
            || sensorType == Sensor.TYPE_MAGNETIC_FIELD
            || sensorType == Sensor.TYPE_MAGNETIC_FIELD_UNCALIBRATED);
    }
```

In practice, we see that while Android does indeed transmit measurement requests as quickly as it can, many consecutive readings return the same value. The explanation lies in the design of the fuel gauges themselves: for each metric, they contain a physical register that is updated with a certain frequency. If nothing (except the limitations of the serial link) prevents you from scanning as fast as you want, the data read will be limited by this frequency, which varies with the model of integrated circuit. On the market, one can find refreshments around 4 and 10 Hz, which disqualifies, among others, attacks aiming at the execution of cryptographic code: the attack presented in [8] requires, for example, measurements at the microsecond scale. These frequencies nevertheless leave malicious exploitations on human speed uses within reach. It should also be noted that even if the `HIGH_SAMPLING_RATE_SENSORS` permission mentioned above were applied to fuel gauges, it would still be useless due to this inherent limitation of the hardware.

2.2 State of the art and its applicability to fuel gauges

The most judicious category of attack in this context is the side channel attacks. They rely on exploiting information from the operation of a system, rather than a design, specification or protocol flaws. A founding example is the recovery of secrets for Diffie-Hellman, RSA or DSS based on execution times [8]. For personal identification codes, an attack based on electromagnetic emissions during

sequence verification is presented in [9]. On the targeted platforms, fuel gauges offer the potential to exploit a major side channel, real-time current consumption, without requiring any additional equipment.

PIN entry on phones has already been targeted in several ways. In [4] and [11], it is spied on Android 2, via motion or rotation sensors, and requires training data. This technique will be pushed in [2], where the authors merge readings on several different sensors of various natures, still requiring training, presumably on Android 5 or 6. [6] explains a spying technique applicable when the phone is charging through its USB port: specific sensing hardware, plugged into the line, intercepts the current and infers the position of touches using a convolutional neural network, also trained but only by the attacker.

By exploiting fuel gauges, we aim at proposing a PIN code attack that does not require training or scanning of a wide variety of onboard sensors.

2.3 Identified risks

Regarding the state of the art and fuel gauges applications, three risks are identified:

- The first one is a privacy risk. An attacker can log to the second events such as the use or not of the phone, the activation or deactivation of wireless connectivity, the reception or transmission of communication, etc.
- A second one due to the creation of a hidden communication channel on the platform: the real-time consumption tracking. We have seen that it would be accessible in reading mode by all applications. We must also consider that all the actors have a *de facto* inalienable right to write on it, since each one can, by its execution, cause a lesser or additional consumption. Thus, for example, an application `A`, having access to sensitive data but not to the network, could transmit them, by means of certain signal modulation techniques, to an application `B`, having access to the network but not to the sensitive contents.
- A third one, particularly aimed at implementations of secure and sensitive solutions: on many fuel gauges, the refresh rate, although low, is of the same order of magnitude as human interactions. One can then fear the harvesting of information during the entry of a secret data.

In the following, we will focus on the latter, in order to retrieve a PIN, intended for another process.

3 Sensitive data recovery through fuel gauges

3.1 Testing tools

To demonstrate this attack, we focused on exploiting Android versions between 9 and 12. Targeting earlier versions should not pose any obstacle except for the

lack of standard fuel gauge support before version 5 (*Lollipop*). Our testing was conducted on devices in Google's Nexus and Pixel lines, both with and without a USB cable connected to the platform. When the fuel gauge is not connected to a power source, it returns negative values for the instantaneous current consumption as energy is being drained from the battery. However, when a charging cable is connected, the charging current is stable enough not to question the attack, and positive values are obtained if the system consumes less energy than it absorbs via the cable. In our testing, we were able to prototype the attack successfully. To facilitate our temporal attack, we developed a simple target application, as shown in Figure 3, which prompts the user to enter a PIN code using a virtual numeric keypad. The entry must be confirmed using a validation key located in a known location on the screen, thus creating a time lapse between each press that can be exploited by the attacker. Additionally, the keypad is not perfectly square, with varying distances between certain keys, which can also slightly work in favor of the attacker. It is worth noting that most phones come with a factory configuration that includes a vibration feedback when a key is pressed, which requires substantial energy to activate, representing an aggravating factor in this situation. Overall, our prototype attack demonstrates the vulnerability of Android devices to temporal attacks, which can be exploited by an attacker with the knowledge and resources to do so.

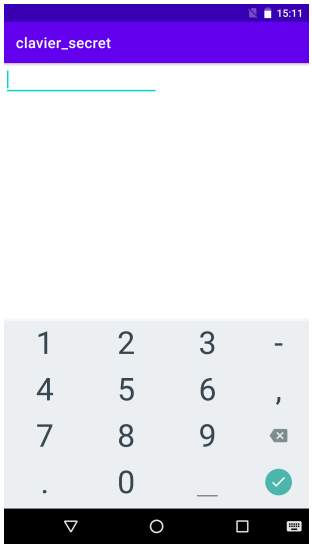


Fig. 3: Screenshot of the target application.

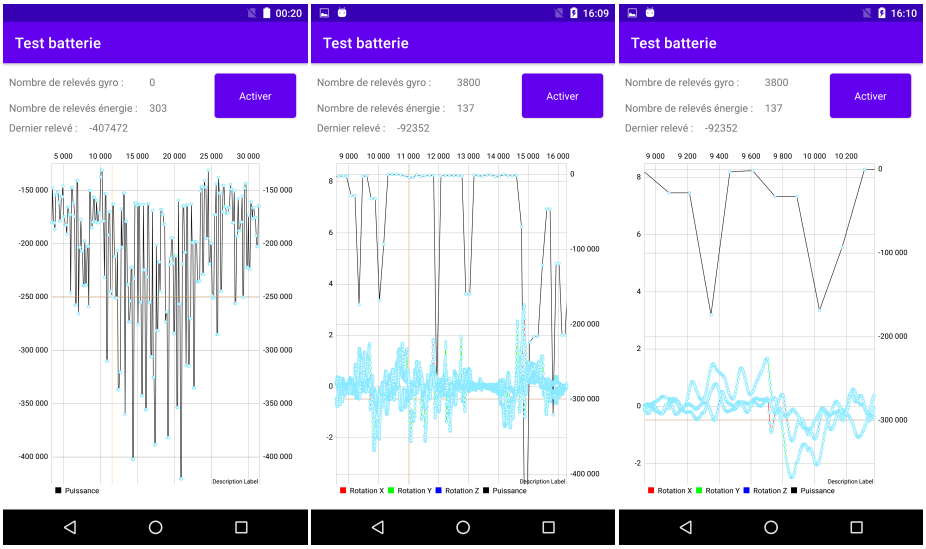
We also developed an Android application dedicated to the attack. The application includes an Android service that can scan the fuel gauge without inter-

fering with the user interface and can do so even when the phone is in standby mode or the user switches to another app. As energy readings accumulate, they are stored in memory and can be accessed later to avoid lowering the signal-to-noise ratio that could occur with real-time extraction via a wired or wireless Android Debug Bridge (ADB) link. For the purposes of this demonstration, we will display a graph directly showing the data collected from the fuel gauge. The X-axis represents the number of measurements collected, and the Y-axis indicates the instantaneous current consumption. It's important to note that we only rely on the instantaneous current consumption metric for this attack. Other metrics such as voltage, temperature, or remaining charge are either not precise enough or not representative of the instantaneous activity on the platform.

3.2 Exploitation

When setting up a timing-based attack, there are two main aspects to consider: capturing the data and detecting significant events, and analyzing the data to reduce the secret's space. The first aspect can be based on various metrics or physical phenomena, including the one we introduced in this article. The second aspect relies on methods that should be applicable regardless of the technique used to capture the data. These methods are aimed at analyzing the data to reduce the space of possible secrets that could have generated the captured timing data. By applying these methods, an attacker can gain insights into the possible values of the secret and thus increase their chances of successfully cracking it. Overall, timing-based attacks are a powerful tool in an attacker's arsenal, and it is essential to consider both the capture and analysis of the data to successfully execute such an attack.

Temporal position detection Figure 4a illustrates a sequence where we have our finger resting on the touchscreen panel between the 12,500 and 22,500 measurements, counted on the x-axis. The increased downward variance in this region indicates that the fuel gauges are well able to detect the delta in power consumption of a phone or tablet when touching the screen, even without vibration. Using a device with a cracked but functional touchscreen does not change this result. The curve decreases when touching because the consumption increases during these moments (there is more current coming out of the battery). While we can guess that this consumption differential is due to the physical phenomenon at play on capacitive technologies, we can also assume that a software processing necessary to manage this input mode is also responsible. On the Figure 4b, we can see a typical sequence corresponding to a 4-digit code entry, with validation. The fact that the vibration is triggered at each press makes the reading obvious: the peaks corresponding to each entry are clearly visible to the naked eye. In this case, it is not necessary to deploy signal analysis techniques to conduct the attack. However, the low update rate of the fuel gauges makes us lack precision to conduct a temporal attack. The capture in Figure 4c illustrates the same dataset as in the middle one, but where we have magnified the first two peaks.



(a) (b) (c)

Fig. 4: Screenshots of the application used for the attack.

The three curves concentrated at the bottom of the screen correspond to the gyroscopic readings, which are also withdrawn. These can allow us to refine the temporal contact points: under an energy peak, we can retain the points where the derivatives of the three axes of rotation reach zero at the same instant after a variation.

Temporal attack Once the delays between the inputs have been precisely identified, we can carry out a temporal attack on the code.

On our side, we have developed a recursive and deterministic algorithm, which, depending from the time lapses provided, unrolls the tree of the possible codes starting from the end, as illustrated in listing 1.3 with values obtained from captures such as the one in 4b. It works in reverse order, since we know the user is required to confirm the PIN by pressing the validation key. This way, by observing the delays, we can infer on the most likely digits introduced right before, and so on. Thus, in the tree of most possible codes, the validation key, represented with the number 10, is the root, and the first digit of each possible PIN are the end of the branches. This method is conceptually close to the one presented in [5].

The state of the art shows that temporal analysis can provide convincing results by exploiting several possible techniques, often applied to beep tones emitted by physical pin pads. In [7], the possible PIN codes are extracted with the help of Hidden Markov Models (HMMs). However in [10], the use of machine

Listing 1.3: Example output from the developed deterministic algorithm. The 10-key represents the validation one, pressed at the end of the sequence. The proposed sequences are in reverse.

```
(arbre '(1.74 2 2.01 2.52) (cons '(10) '()) 0)
=> ((((((10 6 8 2 3) (10 6 8 2 1))) (((10 6 2 8 9) (10 6 2 8 7))))))
```

learning techniques was highlighted. In all cases, quantifying the rate of success is not easy, since it depends a lot on the PIN itself, mostly on the variability in the distances between the digits composing it.

Finally, a further reduction in the remaining code space can be achieved by making a kinematic study of the gyroscopic readings, which are harvested anyway to refine the peaks. For example, the slight rotation of the device needed to press the 1 key is different from the one corresponding to the pressing of the 0 key: this bias can help to choose the most probable first digit.

4 Discussion

In this work, we have demonstrated the concrete existence of a privacy risk on many Android-based platforms. It has been illustrated with an example involving the recovery of personal code, but this danger should not be neglected in general, including activity spying and the establishment and exploitation of a hidden communication channel. If a software environment is intended to host executable content from various third-party actors, then the system designer should pay particular attention to the integration of a fuel gauge. Delegation of responsibility for energy management may indeed bring additional security considerations. Similarly, while this article has focused on the case of Android due to its widespread presence today, the risk presented is not specific to this OS. There are several devices on the market that embed a fuel gauge in different environments. One example is the Nintendo Switch, which is based on a FreeBSD kernel and has such an integrated circuit as part of its battery management. This danger can easily be taken into account when one controls the platform and its OS, since in this case it is sufficient to act on the security policy governing access to such components. This approach has been applied in [1] to regulate access to sensors in general, by means of modifications to the native system libraries, and to the applications before their installation. However, mitigation is much more complex when one is an actor with access only to the application layer, such as a third-party developer. Securing such a sensitive process is then a new task, where one can no longer rely on inter-application isolation. Moreover, thwarting auxiliary channels attacks is particularly complex when working with intermediate code generated from Java or Kotlin sources, as it is mostly the case on Android. This work is currently under study.

5 Conclusion

In this article, we have seen that some autonomous devices embed a fuel gauge and that this, due to its capabilities, can imply privacy risks. We have shown that their inclusion in the Android system is vulnerable to this, by setting up one of the possible exploits, namely secret spying. Since this danger contradicts the guarantee of isolation normally provided by the environment, it has serious consequences on the production of sensitive applications. It is therefore necessary for third-party developers to adopt measures and precautions. These solutions are currently being studied.

Acknowledgments

The authors would like to thank Guillaume Bouffard, for his creative contributions and support throughout this work.

References

1. Bai, X., Yin, J., Wang, Y.P.: Sensor guardian: prevent privacy inference on android sensors (2017). https://doi.org/10.1186/s13635-017-0061-8
2. Berend, D., Jungk, B., Bhasin, S.: There goes your PIN: Exploiting smartphone sensor fusion under single and cross user setting (2017), `https://eprint.iacr.org/2017/1169`
3. Bokhari, M.A., Xia, Y., Zhou, B., Alexander, B., Wagner, M.: Validation of internal meters of mobile android devices (2017). https://doi.org/10.48550/arXiv.1701.07095, `http://arxiv.org/abs/1701.07095`
4. Cai, L., Chen, H.: TouchLogger: Inferring keystrokes on touch screen from smartphone motion (2011)
5. Cardaioli, M., Conti, M., Balagani, K., Gasti, P.: Your PIN sounds good! on the feasibility of PIN inference through audio leakage (2019). https://doi.org/10.48550/arXiv.1905.08742, number: arXiv:1905.08742
6. Cronin, P., Gao, X., Yang, C., Wang, H.: Charger-surfing: Exploiting a power line side-channel for smartphone information leakage (2021)
7. Foo Kune, D., Kim, Y.: Timing attacks on PIN input devices. In: Proceedings of the 17th ACM conference on Computer and communications security. Association for Computing Machinery (2010). https://doi.org/10.1145/1866307.1866395
8. Kocher, P.C.: Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In: Advances in Cryptology — CRYPTO '96 (1996)
9. Le Bouder, H., Barry, T., Couroussé, D., Lanet, J.L., Lashermes, R.: A Template Attack Against VERIFY PIN Algorithms. In: SECRYPT 2016 (2016), `https://hal.inria.fr/hal-01383143`
10. Panda, S., Liu, Y., Hancke, G.P., Qureshi, U.M.: Behavioral acoustic emanations: Attack and verification of PIN entry using keypress sounds (2020)
11. Xu, Z., Bai, K., Zhu, S.: TapLogger: inferring user inputs on smartphone touchscreens using on-board motion sensors. In: Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks. WISEC '12 (2012). https://doi.org/10.1145/2185448.2185465