IAIK

SCIENCE ▪ PASSION ▪ TECHNOLOGY

TU Graz

Gergö Kranz

AndroGUARD: Mitigation of Sensor Fingerprinting on Android

BACHELOR'S THESIS

Bachelor's degree programme: Software Engineering and Management

Supervisor

Gerald Palfinger

Institute of Applied Information Processing and Communications
Graz University of Technology

Graz, August 2024

Abstract

Sensor fingerprinting is a technique by which it is possible to uniquely identify users based on specific characteristics of the sensors in their devices. This unique fingerprint has the potential to deanonymize mobile devices and track them.

In this thesis, we look into the fingerprintability of the sensors in an Android device. Our main contribution is the implementation of the masking of the built-in sensor error values to decrease Android fingerprinting efficiency. The built-in factory calibration error refers to the slight inconsistencies and inaccuracies introduced during the manufacturing and factory calibration process of sensors. These errors are unique to each device and remain constant over its lifetime, providing a consistent yet unique signature that can be used for fingerprinting. We examine proposed mitigation techniques and implement our solution to mitigate fingerprinting, that can be used with a number of Android apps. Our approach adds random noise to the original sensor values, before they are passed down to the function handling them. With the help of the A2P2 framework we can apply our patch to any compatible android app. We conclude that by adding the random noise we mask the consistency of the built-in factory error to make sensor fingerprinting more complicated.

Keywords: Android API · Sensor Fingerprinting · Privacy · Protection

1. Introduction

With the misuse of the Android API, apps can uniquely identify users based on specific characteristics of their device and system configuration [25]. Tracking Android users without their knowledge and explicit consent based on their device fingerprint can have various privacy and security implications. Individuals can be tracked across different apps and services without their explicit consent, which is a clear privacy concern. While advertisers may use fingerprints to build detailed profiles of users for targeted advertising, malicious actors on the other hand could exploit the information gathered through fingerprinting for targeted attacks. For example, they may use the fingerprint data to tailor phishing attempts or deliver malware specific to the user's device configuration. Detailed API fingerprinting could potentially expose a variety of sensitive information that, when combined, may lead to a range of fraudulent activities.

Fingerprints can be based on various characteristics of the software and hardware configuration of the device, providing a way to distinguish it from other devices [30, 6, 22, 1, 10]. Some of the used information is based on the system properties, like software and hardware properties of the device, and other more based on the software, network or device configuration by the user. Details about the Android build version, manufacturer, model, and build fingerprint are often included in the fingerprint. Information about the device hardware, such as the processor architecture, RAM size, storage capacity, and sensor availability, may be part of the fingerprint. Android also exposes various system properties that contain information about the device. These properties can include details about the operating system, kernel version, and runtime parameters.

In our work, we focus on reducing the fingerprintability of sensors. Sensor fingerprinting refers to the process of identifying and tracking individual devices based on the unique characteristics and imperfections of their built-in sensors [3, 16, 4]. These imperfections in sensors, exploited for fingerprinting, arise from manufacturing variations, material defects, calibration inaccuracies, and various environmental factors [2, 9, 31]. Android devices, which are equipped with various sensors like accelerometers and gyroscopes are vulnerable to this type of fingerprinting [8]. These sensors, while essential for providing enhanced functionality and user experiences, can inadvertently expose unique error patterns that can be exploited for tracking purposes. By focusing on sensor fingerprinting, this thesis aims to explore the methodologies used to extract these unique signatures, assess the potential privacy risks, and implement effective countermeasures to mitigate these risks, thereby enhancing user privacy and security in the Android ecosystem.

Multiple papers have been published on fingerprinting sensors in Android devices by recording their built-in imperfections, which arise from the manufacturing process [6]. These studies demonstrate that the measurement errors inherent in these sensors are unique to each device, creating a distinctive fingerprint [5, 1]. Furthermore, they show

that these errors are consistent and stable over the lifetime of the sensor, making them reliable indicators for identifying and tracking individual devices [3]. This consistency in sensor errors highlights the potential privacy risks associated with sensor fingerprinting, as it allows for the persistent and unique identification of devices based solely on their hardware characteristics.

Our implementation AndroGUARD builds on suggestions from some of the research papers which covered this issue [8, 1, 5, 7]. To apply our implementation in other applications we use the Android Application Patching Pipeline, short A2P2, for easy deployment [17]. By integrating these recommended strategies into our patch, we aim to effectively obfuscate sensor data and prevent the extraction of identifiable information [7, 8, 1]. By introducing random variations to sensor outputs, we can obscure unique error patterns, making it more challenging for adversaries to reliably fingerprint and track devices [8, 1]. Our approach not only addresses the vulnerabilities highlighted in the literature but also provides practical solutions to enhance the privacy and security of Android users.

Outline. In chapter 2, we offer an introduction to fingerprinting techniques. Moving forward to chapter 3, we focus on sensor fingerprinting, where we detail its mechanics, common sensor types targeted, and real-world implications. Building on this information, in chapter 4 we examine existing countermeasures against fingerprinting, evaluating their effectiveness and limitations to provide context for our proposed solution. Chapter 5 investigates the mitigation concepts presented in prior research, understanding relevant literature for our implementation of the selected mitigation strategy. Following this, chapter 6 provides a detailed breakdown of the patch creation process, including integration with the A2P2 framework and key functionalities such as the intercept method and noise generation. We then outline the step-by-step process of applying the patch to APKs in chapter 7. Chapter 8 shifts focus to validating the effectiveness of the patch, detailing the testing methodology, results, and areas for further improvement. Finally, chapter 9 discusses the limitations of our approach and the concluding chapter 10 summarizes our findings, reflects on the implications of the project, and outline directions for future research and development.

2. Background

With the help of fingerprinting, companies or individuals with malicious intent can track the user and their habits to create personalized profiles of them. Therefore, tracking users based on the fingerprint of their devices can have various privacy and security implications. Some of this impact includes privacy concerns, security risk, data misuse and user experience implications [23]. While advertisers may use fingerprints to build detailed profiles of users for targeted advertising, malicious actors on the other hand could exploit the information gathered for their targeted attacks. For example, they may use the fingerprint data to tailor phishing attempts or deliver malware specific to the user's device configuration. Detailed fingerprinting could potentially expose a variety of sensitive information that, when combined, may lead to identity theft or other fraudulent activities. Due to this, users might feel uncomfortable or even violated if they perceive that too much of their personal information is being used without their control. The practice of extensive fingerprinting may run against data protection regulations such as GDPR, which require explicit user consent for collecting and processing personal data.

2.1. Browser Fingerprinting

Browser fingerprinting is a widely known technique, which is frequently exploited to track users across different websites [26]. By using mainly JavaScript the visited website can access system information like screen resolution or settings set by the user. These attributes can be combined to create a unique identifier for the device.

Browser fingerprinting. There are multiple different methods applied by an adversary as described in the paper "A classification of web browser fingerprinting techniques" [29]. An adversary can fingerprint the browser by checking the browser version, installed fonts and other browser specific attributes or the HTML5 by rendering text and WebGL scenes onto the screen using the canvas element, and then reading the pixel data back to create a fingerprint. They can also use JavaScript to create cross-browser fingerprints, which are browser independent and focus more on the hardware the browser is running on, like screen resolution or networking information. With the use of JavaScript one can also access sensor data, like an accelerometers calibration imperfections, to create a such fingerprint.

Browser fingerprinting protections. A number of solutions are already presented by multiple research papers, to identify and block fingerprinting websites, making browsing the internet more private for everyone [18, 26, 24, 28]. JShelter [26] blocks the execution

of JavaScript to mitigate the gathering of personal data used for fingerprinting to increase browser security. An other research, PriVaricator [24] focusses on making the fingerprints non-deterministic, by introducing randomization to break identifiability when visiting an the same website multiple times. These research projects can create some problems with embedded functionalities. To solve this issue FP-Block [28] creates a different unrelated fingerprint for the embedded resources to maintain their functionality, but prevent tracking.

2.2. Smartphone Fingerprinting

Fingerprinting Android devices is similar to browser fingerprinting. Fingerprinting can uniquely identify users based on specific characteristics of their device and system configuration [25]. Different sensors and system information are accessed through the use of the Android API, in order to create a unique identifier.

The study "Fingerprinting mobile devices: A short analysis" highlights the differences the distinct behaviours in the absence of plugins like Flash [21]. The analysis covers over 17,000 mobile fingerprints collected, finding that 81% of these fingerprints are unique. The dataset includes a multitude of operating systems, primarily Android ones. The user-agent attribute is the most distinctive, with over 3,000 unique values, due to detailed information about the device's operating system, version, and browser. The study also confirmed that mobile user-agents contain more specific information than desktop user-agents, sometimes indicating modified third-party ROMs. The canvas attribute, influenced by unique emoji sets from different manufacturers, is the second most distinctive with around 700 unique values. The study confirms that mobile browser fingerprinting is effective in uniquely identifying devices, due to specific mobile software environment characteristics.

Restricted access to sensitive identifiers is crucial to hinder fingerprinting attempts. Android advocates for the responsible use of device information, by blocking access to unique identifiers, like IMEI numbers [11, 23]. Users have the freedom to review and modify app permissions through system settings, enabling them to manage access to personal information effectively. When enforced with user education this permission model empowers the owner of the device to take informed decisions about the data they are willing to share with applications.

3. Sensor Fingerprinting

In this paper, our central focus is the complex problem of sensor fingerprinting present in a wide range of Android devices [2]. Sensors embedded within smartphones and tablets serve as the foundation for an extensive array of functionalities, spanning from location tracking and environmental monitoring to augmented reality experiences and health tracking applications. However, these features create a potential vulnerability regarding the sensor data, if exploited through fingerprinting techniques, can jeopardize user privacy and security [23].

It has been proven by multiple research papers that the built-in error of sensors remains consistent over the lifetime of the device [32, 16]. This inherent characteristic of sensors, stemming from manufacturing imperfections and the initial calibration, results in a unique and persistent signature that does not change significantly over time. Studies have demonstrated that these discrepancies can be reliably measured and used to fingerprint individual devices, posing a significant threat to user privacy. This stability of sensor errors forms the basis for sensor fingerprinting techniques, highlighting the need for effective countermeasures to disrupt these patterns and protect against unauthorized tracking and identification.

One way to create a fingerprint using built-in sensors is to play a tone from the device's speaker and record it with the microphone [3, 6, 2]. This method leverages the unique characteristics of both the speaker and microphone in each device. Dividing the recorded intensity by the original intensity, a feedback ratio can be calculated, which serves as a distinctive marker for that device. To enhance accuracy, multiple samples are recorded at various frequencies, and the Fourier coefficients are computed to isolate the main frequency and its harmonics. One downside of this technique is, that it relies on a relatively quiet environment and is easily influenced by the surroundings interfering with the acoustics.

The accelerometer is ideal for fingerprinting because users often leave their devices still, such as on a desk, allowing for consistent data collection [3, 16, 32, 2]. Unlike audio-based fingerprinting, which needs a known signal, accelerometer fingerprinting relies on passive background measurements when the device is stationary. When the device is at rest, the acceleration vector equals the gravitational constant, making detection straightforward. To estimate the accelerometer's calibration parameters, measurements are taken with the device facing up and down, then the sensitivity and the offset of the sensor are calculated. This method is effective even if the surface is not perfectly level and can be done without user interaction, as devices are often left in both orientations. With more data and advanced processing, all six accelerometer parameters can be estimated, improving device identification accuracy.

Sensor fingerprinting creates a significant threat to privacy because it often exploits the fact that apps do not require elevated permissions to access the sensors necessary for creating a fingerprint. Unlike other types of data access that might prompt user warnings or require explicit permissions, sensor data is typically more accessible, allowing malicious applications to gather detailed information without raising suspicion. This accessibility means that many apps can freely collect and analyze sensor data, such as accelerometer and gyroscope readings, to generate unique device fingerprints. These fingerprints can then be used to track users across different applications and sessions, severely compromising user privacy. The ease with which apps can access these sensors, coupled with the detailed and unique nature of the data they provide, makes sensor fingerprinting a potent privacy threat.

Not only can mobile applications access sensors through the Android API, but websites can also leverage this functionality using JavaScript. This extension of sensor access to web-based platforms presents additional challenges and considerations for privacy and security. With the broad range of web technologies such as the `DeviceOrientation` and `DeviceMotion` APIs, web developers can access sensor data directly from the browser, enabling a wide range of sensor-based interactions within web applications [9]. While this capability unlocks new possibilities for immersive web experiences, it also introduces potential risks, as websites can collect sensitive sensor data without explicit user consent. Unlike native mobile applications, where it can be known that they access sensors, websites may access sensor data without notifying users, raising concerns about unauthorized data collection and privacy infringement. Additionally, the diverse nature of web environments and the multitude of devices accessing them present challenges in ensuring consistent and secure sensor data handling across different platforms and browsers. To protect against fingerprinting inside browser applications one can apply the same protection as for any other android applications. By modifying the Android API queries before they reach the browser application, one can manipulate the values, queried by JavaScript inside the browser sent to the system.

We analyze the subtleties of sensor fingerprinting methods deployed across the spectrum of Android devices. We recognize the significance of understanding the techniques leveraged to exploit the unique signatures inherent within sensor readings, to identify and track individual devices or users. By comprehensively examining the underlying mechanisms and implications of sensor fingerprinting, we aim to implement a solution to reduce the risks in order to protect user privacy.

We recognize the importance of developing robust frameworks or mechanisms capable of obfuscating or randomizing sensor data, thereby circumventing attempts at device or user identification through fingerprinting techniques.

Our methodology will encompass a versatile approach, incorporating comprehensive literature review about Android fingerprinting, patch development and testing of our method. The analysis of the existing research helps us gain insight to develop our fingerprinting counter measure.

Ultimately, our goal within this thesis is not only to deepen the understanding of sensor fingerprinting within the Android ecosystem but also to offer tangible solutions that empower users to maintain control over their personal data.

4. Methodology

A number of solutions are already present in order to protect browsers against fingerprinting [26, 18, 24, 28]. One fo them is to monitor and restrict access to properties commonly used for fingerprinting and prevent network traffic to tracking servers [26]. Another method is to create fake profiles to counteract online tracking. By altering fingerprinting data to mimic real world data, we can either hinder fingerprinting efforts or present a valid but fake profile to someone creating a fingerprint [19]. However, there are not so many solutions to protect Android devices.

4.1. State-of-the-Art Methods

The Android API has implemented a couple of proactive measures to counteract fingerprinting [11]. This includes regulated control over access to identifying information and a robust permission system. Access control is a highly effective tool to combat Android device fingerprinting. Android ensures that applications can only request and use the permissions that are explicitly granted to them [12]. Because applications must explicitly request user consent to access sensitive data and functionalities, ensuring transparency and user awareness.

These measures are designed to make it challenging for apps and services to create unique fingerprints by limiting access to certain APIs and data points, but do not protect accessible characteristics.

4.2. Proposed Solutions

There are two proposed countermeasures designed to enhance the privacy of sensor data and protect against fingerprinting: calibration and noise generation [7]. Each of these methods addresses the issue of sensor data consistency in distinct ways, offering a comprehensive approach to mitigating the risk of device fingerprinting.

Calibration. Calibration is the first countermeasure and involves the systematic adjustment of sensor readings to account for and eliminate inherent biases and errors. By carefully calibrating sensors, we can reduce the fixed discrepancies that arise from manufacturing variances and usage patterns, which are often exploited for fingerprinting. This process ensures that the sensor outputs are more uniform and less distinctive across different devices. Calibration works by applying specific corrections to the sensor data, aligning the readings more closely with standardized values. This reduces the unique

signatures that individual sensors might otherwise exhibit, making it more challenging to use these readings for identifying and tracking devices.

Noise. Noise Generation, the second countermeasure, directly targets the fingerprinting process by introducing variability into the sensor data. This method employs the deliberate addition of random noise to the sensor readings, effectively masking the original values. The noise generation technique ensures that each sensor output is slightly altered every time it is read, preventing the formation of a consistent and stable fingerprint. By applying this noise, the sensor data becomes less predictable and more resistant to fingerprinting efforts.

4.3. Disadvantages

Both methods, calibration and noise generation, have their own set of downsides that must be considered when implementing them to protect against sensor fingerprinting.

Calibration. Calibration as a countermeasure requires user awareness and interaction, which can be a significant drawback. Users must be informed about the necessity of calibrating their devices to mitigate the risk of fingerprinting, and they need to actively participate in the calibration process. This process can involve following specific instructions to adjust sensor settings or performing a series of actions to allow the device to calibrate itself accurately. Such requirements can be burdensome for users who may lack the technical expertise or the patience to carry out these procedures. Additionally, users may need to perform complex procedures or use specialized equipment to achieve accurate calibration, which can be impractical for the average user. Even minor errors in the calibration process can lead to significant deviations in sensor readings, undermining the effectiveness of this countermeasure. The complexity and precision required for perfect calibration make it a challenging task that may not always yield the desired results.

Noise. However noise generation introduces its own set of challenges. While it is highly effective at obfuscating sensor data and preventing the creation of consistent fingerprints, it can also degrade the functionality of applications that rely heavily on precise sensor readings. For instance, applications that depend on exact measurements, such as fitness trackers, gaming apps with motion controls, navigation tools, and certain professional or scientific applications, may experience reduced accuracy and reliability. The random noise added to sensor data can interfere with the app's ability to interpret user actions or environmental conditions correctly, leading to a compromised user experience. The empirical proof for this is in chapter 8 in the paragraph 8.1 about usability. This degradation is particularly problematic in scenarios where precise sensor data is critical for safety or performance, such as in medical monitoring apps or systems that assist with physical rehabilitation.

Moreover, the implementation of noise generation must be carefully balanced to ensure that the level of introduced noise is sufficient to disrupt fingerprinting attempts without excessively impairing the app's functionality. This balance can be challenging to achieve, as different applications and sensor types have varying tolerance levels for noise.

4.4. Selected Approach

In summary, while calibration requires user engagement and can be prone to inaccuracies if not performed correctly, noise generation can impact the performance of apps that depend on accurate sensor data. Both methods have inherent trade-offs that need to be carefully managed to effectively enhance privacy without significantly compromising the user experience or app functionality.

Noise generation offers a more straightforward approach that requires minimal user intervention, making it more accessible and user-friendly. Additionally, the impact on user experience is typically less pronounced with noise generation, as it does not require users to actively engage in the calibration process or make manual adjustments. Due to its simplicity and reduced user interaction compared to calibration, we opt to implement noise generation as the primary countermeasure against sensor fingerprinting. By prioritizing noise generation over calibration, we aim to strike a balance between effectiveness and usability, providing a practical solution for mitigating sensor fingerprinting while minimizing user burden.

5. Concept

The individual hardware instances of a particular sensor displays significant disparities, largely attributed to imperfections in the manufacturing and assembly processes. These variations introduce distinctive biases into the sampled data retrieved from the sensor, as described above [3, 32, 16, 6].

By manipulating the read sensor data, we can effectively disrupt the persistency of the factory measurement error inherent in many sensors. This manipulation involves the introduction of variability into the sensor readings, which counteracts the static nature of these factory errors. By adding a randomized value to certain sensor readouts each time they are queried, we ensure that the information collected does not remain constant over time. This dynamic alteration of sensor data makes it more challenging to uniquely identify and track devices and thus significantly enhances user privacy and security.

5.1. Our Solution

The patch utilizes the Android Application Patching Pipeline (A2P2) [17], to modify APK files and intercept function calls to specific classes and functions within the Android operating system. The A2P2 framework allows for extensive modifications at the application level, enabling us to effectively intervene in the normal operation of various sensor-related functions. By leveraging this framework, we can inject custom code into the APKs, ensuring that any attempt to read sensor data is first filtered through our randomized value generation algorithm. This process involves an analysis of the application's structure to identify the precise points where sensor data is accessed. Once these points are identified, the A2P2 framework facilitates the interception of these function calls, allowing us to alter the data being returned. This interception mechanism is crucial for implementing our countermeasure, as it provides the means to introduce noise or randomized values into the sensor data stream before the values are forwarded to the app. Consequently, this approach not only prevents the original, unaltered sensor data from being used to create persistent fingerprints but also maintains the overall functionality and user experience of the application. Through the application of the A2P2 framework, we achieve a seamless integration of our privacy-enhancing modifications, effectively shielding users from the privacy risks associated with sensor fingerprinting.

5.2. Modifying the Android API

After examining the Android API, we observe that every application utilizing sensor values must implement the abstract class `SensorEventListener` [14]. This class

serves as the main interface for receiving sensor data. The `SensorEventListener` interface defines two key methods:`onAccuracyChanged(Sensor sensor, int accuracy)` and `onSensorChanged(SensorEvent event)`. Out of these two, the `onSensorChanged` method is particularly significant, as it is the primary mechanism through which applications receive and process sensor data.

The `registerListener` method in Android is a crucial part of the sensor framework. It enables applications to listen for and respond to sensor events. When an application needs to interact with hardware sensors, such as accelerometers or gyroscopes, it utilizes this function to register an instance of `SensorEventListener` with the `SensorManager` [15] to a specific sensor. This registration process effectively sets up a communication channel between the sensor hardware and the application, allowing the application to listen for and respond to sensor events. The `onSensorChanged(SensorEvent event)` method is then invoked by a system interrupt whenever there is a change in the sensor's data, providing the application with real-time access to the sensor readings.

The `SensorEvent` object passed to the `onSensorChanged` method contains detailed information about the sensor event, including the type of sensor, the accuracy of the sensor data, the timestamp of the event, and the actual sensor values [13]. These values are used to drive a wide range of features, from motion detection and environmental sensing to user activity recognition and device orientation.

Given this architecture, in order to manipulate or intercept sensor data the `onSensorChanged` method has to be intercepted. By intercepting calls to this method we can effectively modify the sensor values passed down from the system to introduce our countermeasures to disrupt the sensor fingerprinting process. This involves modifying the application's APK to inject a custom code that alters the sensor values before they are processed by the application. Using A2P2, we can intercept the registerListener method calls which register a `SensorEventListener` class to handle returned sensor values.

Our patch replaces the original `registerListener` function with a customized version designed to enhance privacy. This customized function intercepts the instance of the `SensorEventListener` passed down to the `SensorManager` and replaces it with a custom instance. This instance contains the original `SensorEventListener` and a function which identifies those sensor values that are susceptible to fingerprinting, and applies a calculated layer of noise to them. After modifying these fingerprintable sensor values with random noise, the patch then calls the original `onSensorChanged` method of the original listener instance, passing the altered values instead of the original ones.

This process ensures that while the application continues to receive the sensor data it needs to function properly, the data has been obfuscated to prevent the creation of a consistent and unique fingerprint based on the sensor readings. This method effectively disrupts any attempt, when any custom class derived from the `SensorEventListener` class is used, to utilize these sensor values for fingerprinting purposes, enhancing the privacy of the user without compromising the application's performance.

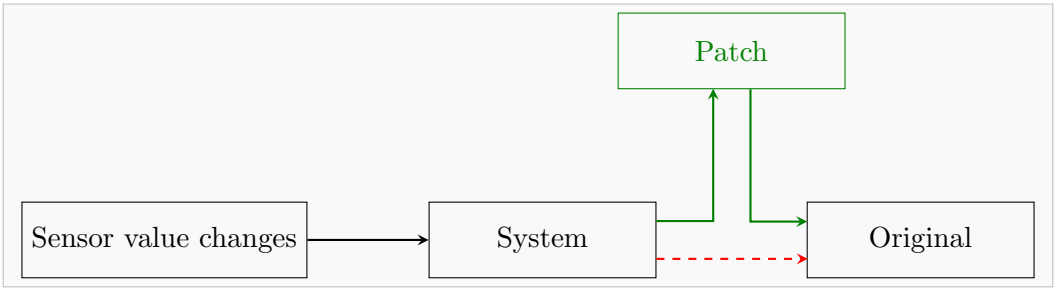


Figure 5.1.: The functioncalls from the system are intercepted by our patch and forwarded after modification to the original function.

5.3. Noise Generation

To create the noise, we employ a formula adapted from another study: $value_{new} = (value_{old} - offset_{sensor})/gain_{sensor}$ [8]. This approach leverages established principles of data obfuscation to effectively introduce variability into the sensor readings. The parameters for offset and gain are chosen within specific ranges determined by that study, ensuring that the introduced noise is effective in obfuscating the sensor data [8].

The formula works by first subtracting an offset value from the original sensor reading and then dividing the result by a gain factor. These offset and gain parameters are dynamically adjusted within their predefined ranges to introduce a degree of randomness while preserving the overall utility of the sensor data. This method ensures that each sensor reading is slightly altered in a way that makes it difficult to reproduce the exact original values, thereby preventing the creation of a consistent and reliable fingerprint.

By carefully selecting the offset and gain ranges, we achieve a balance between data obfuscation and practical usability. The dynamic adjustment of these parameters allows for consistent randomization of the sensor readings, disrupting any attempt to generate a stable fingerprint from the data. This technique aligns with methods demonstrated in previous research, enhancing the variability of sensor outputs to safeguard against fingerprinting attempts.

This ensures that the sensor values received by the application are not consistent over time, thereby preventing the creation of a stable and persistent fingerprint. Through this approach, we can enhance the privacy and security of Android users by obfuscating the sensor data in a way that preserves the application's functionality while mitigating the risks associated with sensor fingerprinting. The reliance on the `SensorEventListener` and its `onSensorChanged` method provides a clear and effective point of intervention, allowing us to implement our countermeasures within the existing Android framework.

Our patch also addresses an important issue when applied to the APKs of mobile browsers. As mobile browsing is widely used, the integration of sensor functionalities into these browsers raises concerns regarding privacy and security. With sensors accessible through web technologies like JavaScript, websites can collect sensor data without explicit user consent, potentially leading to unauthorized data collection and privacy infringements.

By extending the protective measures of our patch to mobile browser APKs, we ensure comprehensive mitigation of sensor fingerprinting across all channels of sensor data access on Android devices. This broader scope enhances the security posture not only of native applications but also of web-based interactions, providing users with greater control over their sensor data and reinforcing digital privacy standards. Through these efforts, our patch contributes to fostering a safer and more transparent digital environment, where users' privacy rights are respected and upheld across diverse digital platforms and applications.

5.4. Loss of Precision

One notable disadvantage of this privacy-enhancing patch is that applications which rely heavily on highly accurate sensor values may suffer in performance. For instance, apps that are controlled by very fine movements, such as precision-based gaming applications, detailed motion tracking software, or certain fitness and health monitoring tools, require exact sensor readings to function optimally. The introduction of noise to the sensor data can lead to a degradation in the app's responsiveness and accuracy, resulting in a less satisfactory user experience for those particular applications. However, this trade-off is mitigated by providing users with the choice of whether or not to apply this privacy enhancement. Users are empowered to decide if they prefer to prioritize their privacy and accept a potential reduction in the accuracy and usability of specific apps, or if they wish to maintain the original performance of these applications at the expense of increased vulnerability to fingerprinting. This user-centric approach allows individuals to make informed decisions based on their specific needs and privacy concerns, balancing the trade-offs between enhanced privacy and app functionality.

The obfuscated sensor data generated by our patch introduces variability and noise to protect against fingerprinting, which inherently means that the data is not as accurate as the original, unaltered sensor readings. While this obfuscation is crucial for enhancing privacy and security by preventing the creation of consistent and identifiable fingerprints, it may create a challenge for applications that rely heavily on precise sensor data for their core functionalities.

Recognizing the need for a balance between privacy protection and the functional requirements of certain apps, one potential solution is the implementation of a custom permission system. This system could allow specific applications to request direct access to unaltered sensor data, bypassing the obfuscation introduced by the patch. This permission system would need to be designed with strict controls to ensure that only trusted applications, which genuinely require precise sensor data, are granted this level of access. Users would be able to review and approve these permissions, giving them full control over which apps can access accurate sensor data and under what circumstances.

The custom permission could be integrated into the patch, providing a new layer of security and user control. When an application requests access to direct sensor data, the system could prompt the user with a detailed explanation of why the app needs this

access and the potential privacy implications. Users would then have the option to grant or deny the request based on their understanding and comfort level.

Developers seeking access to direct sensor data would need to adhere to rigid guidelines. This system would create a transparent and accountable framework for managing sensor data access, providing users with both the protection they need and the functionality they expect from their applications.

6. Implementation

The code contains some very important methods, ensuring the effective obfuscation of sensor data to protect against fingerprinting while maintaining the usability of the application. These functions include an intercept method, a noise generating function, and a random value generation function. Together, they work seamlessly to introduce controlled randomness into the sensor readings, thereby disrupting any attempt to create a stable and consistent fingerprint based on the sensor data.

Intercept Method. This function serves as the central hub of the patch, responsible for applying noise to the sensor data when necessary. The intercept method is designed to replace the calls made to sensor-related functions within the application. When a sensor reading is requested, the intercept method activates, replacing the original function call that retrieves the sensor data. At this point, the method determines whether noise needs to be applied depending on the context and the type of the specific sensor being accessed. Provided the sensor can be fingerprinted and there are attributes for the generation of the appropriate range of noise are present, noise is added to mask fingerprintable values. If noise is required, the intercept method then calls the noise generating function to alter the sensor data before passing it back to the original function. By doing so, the intercept method ensures that any sensor data read by the application is appropriately obfuscated, thereby preventing the formation of consistent and reliable fingerprints.

Noise Generating Function. The primary role of the noise generating function is to apply the calculated noise to the sensor data, effectively masking the original readings. This function operates by receiving the original sensor values intercepted by the intercept method and then modifying these values according to the predefined algorithm. The noise generating function is crucial in balancing privacy protection with usability, ensuring that the altered sensor data remains practical for legitimate application use.

Random Value Generation Function. The random value generation function underpins the entire noise generation process by providing the random values needed to obfuscate the sensor data. This function is designed to generate random numbers within specific ranges, which are determined by the type of sensor being accessed. The random value generation function ensures that each sensor reading is unique and unpredictable, making it significantly more difficult for malicious actors to correlate readings and generate a consistent fingerprint. The generated random values are then used by the noise generating function to alter the original sensor data. By continuously producing new random values, this function guarantees that the noise applied to the sensor data varies

with each reading, thereby enhancing the overall effectiveness of the privacy protection mechanism.

Together, these methods form the basis for our sensor data obfuscation. The intercept method acts as a gatekeeper, ensuring that every sensor reading passes through the noise generation process. The noise generating function applies the necessary modifications to the sensor data, leveraging the random values produced by the random value generation function to introduce controlled variability. This ensures that the sensor data read by any application is sufficiently obfuscated to mitigate fingerprinting attempts while maintaining the functionality required for legitimate uses.

7. Applying of Patch

The application of our patch is designed to be straightforward and user-friendly. It only requires the user to have Java installed, along with the precompiled patch and the A2P2 framework, and an Android APK ready for modification. This simplicity ensures that even those with limited technical expertise can implement the patch without difficulty.

The APK we selected based on our specific testing needs, focusing particularly on applications that heavily utilize sensor data. By selecting such applications, we can effectively assess the impact of the patch on sensor data integrity and the overall performance of the app. This APK would serve as a robust testbed for validating the effectiveness of our patch in obfuscating sensor data and preventing fingerprinting. By conducting tests on applications with high sensor data dependency, we can evaluate both the privacy enhancements introduced by our patch and its practical implications on everyday app usage. This evaluation ensures that our solution not only enhances user privacy but also preserves the essential functions of the applications tested.

8. Validation

The patch is validated through a process involving its application to an app designed to gather sensor measurements. During this period, the app collects a significant amount of sensor data, which is then split into two distinct sets: a training dataset and a test dataset. The training dataset is utilized to train a k-nearest neighbors (k-NN) classification algorithm, which is a standard method for assessing the variation of data points. By training the k-NN algorithm on the modified sensor data, we aim to evaluate how effectively the introduced noise disrupts the unique patterns of the sensor data and prevents accurate fingerprinting. The test dataset is subsequently used to validate the trained algorithm, allowing us to measure the classification accuracy and determine whether the patch has successfully mitigated fingerprintability. This validation process provides crucial insights into the effectiveness of the patch in enhancing sensor data privacy while maintaining a balance with the functionality of the app. Through this evaluation, we can ensure that the patch introduces the necessary privacy protections.

8.1. Testing

Our testing process involved around ten devices running various versions of Android from 10 to 13. We applied the patch to each device and monitored its impact on sensor data handling. By selecting devices with different hardware configurations, we ensured a comprehensive evaluation. We recorded sensor values from the accelerometer and gyroscope over extended periods to assess the patch's influence. Additionally, we tested various applications to understand how the noise added to sensor data affected functionality and user experience.

Usability. The patch was applied to a motion-controlled game [20] to assess its impact on user experience. This specific test was designed to determine how the introduction of noise to sensor readouts would affect applications that rely heavily on precise sensor data. When the app was running with the patch installed, the added noise in the sensor readouts caused noticeable shaking of the controlled object. This unintended side effect made the game more challenging to play, as the smooth and accurate movement necessary for controlling the game object was compromised. The increased difficulty highlighted the potential trade-offs between enhancing privacy and maintaining the usability of certain types of applications, particularly those that depend on fine-tuned sensor accuracy for optimal performance.

Effectivenes. The patch was applied to a data recording app [27] to evaluate its functionality and effectiveness. To conduct this test, sensor values from the accelerometer and gyroscope were recorded over the duration of a minute. This experiment aimed to observe how the patch influenced the consistency of sensor data captured by the application. By analyzing the recorded data, we could determine the extent to which the patch successfully obfuscated the sensor values.

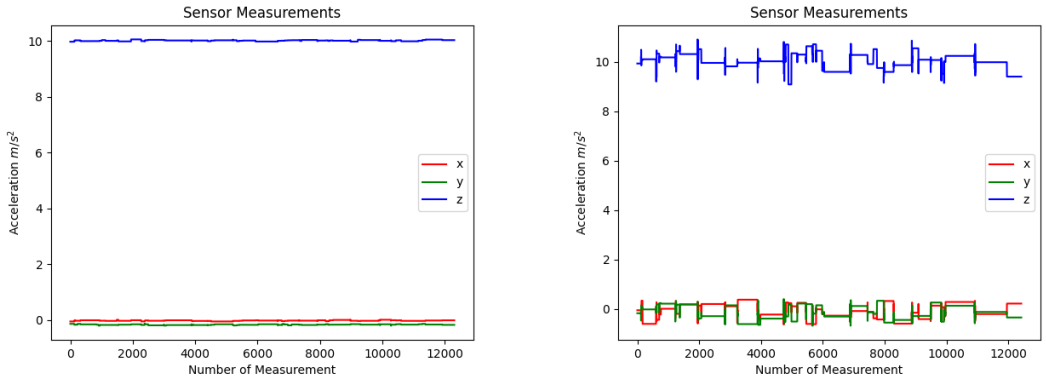
By comparing the consistency of the sensor readouts before and after applying the patch, a significant change can be observed, as shown in Figure 8.1, mirroring the findings of previous research studies. Initially, the sensor values exhibit a high degree of consistency, which can be exploited for fingerprinting. However, after the patch introduces randomization and noise, the sensor readouts become visibly less consistent. When the classification algorithm used the unpatched, unmasked data it was able to associate all of the devices with their measurement. After applying the patch and retraining the algorithm it was not able to match all the devices with 100% accuracy, even with our few samples, when using the measurements of a single type of sensor. This disruption in the data stability effectively hinders the ability to create a reliable fingerprint, aligning with the results documented in other papers. This observed change underscores the effectiveness of the patch in enhancing privacy by preventing the formation of stable and persistent sensor-based identifiers.

During our limited testing, we were unable to test our implementation on a sufficient number of devices to definitively state whether the patch works as intended across a broad spectrum of hardware configurations. This limitation is derived primarily from the constraints inherent in the A2P2 framework we used, which is specifically designed for Android version 11. As a result, our testing was restricted to devices running this particular version of the Android operating system.

While the initial results were promising, indicating that the patch could successfully obfuscate sensor data and disrupt fingerprinting attempts, the limited sample size means that these results cannot be generalized to all devices running Android 11, let alone devices on other versions of the Android OS. Also training our classifier with more features from multiple sensors the inaccuracy disappeared. We believe it is only due to our limited amount of devices used and misclassifications would increase with the use of more devices. Unfortunately, our access to a diverse range of devices running Android 11 was limited. This scarcity of compatible devices hindered our ability to perform extensive and comprehensive testing.

The patch was tested in a limited laboratory setting, which did not sufficiently simulate real-world scenarios. During our testing, the device remained stationary on a table, which does not reflect the dynamic conditions under which mobile devices typically operate. This controlled environment allowed us to focus on the fundamental aspects of the patch functionality without the interference of external factors. However, it also means that the testing did not account for variables such as user movement or environmental changes. Consequently, the results obtained from this limited testing environment may not fully represent the patch's performance in real-world usage.

Despite the limited scope of our testing, we are still convinced of the results of previous studies that state noise generation is a successful method for mitigating fingerprinting. These studies provide a solid foundation for our approach, demonstrating that introducing variability into sensor data can effectively prevent the creation of consistent and identifiable fingerprints. We are confident that, had we access to a sufficient number of test devices, we would have been able to reproduce these findings using our implementation. We are confident in the robustness of our patch design and the alignment of our methods with those proven effective in existing research.



(a) Values before the patch

(b) Values after the patch

Figure 8.1.: Acceleration sensor values measured before and after the application of the patch on the same device.

9. Limitations

While the obfuscated sensor data generated by our patch enhances privacy by disrupting fingerprinting attempts, it does impact the accuracy of sensor readings. To address the needs of applications that rely on precise sensor data, a custom permission system could be used, allowing specific apps to access direct sensor data under controlled and user-approved conditions. This approach balances the importance of both privacy protection and functional accuracy, ensuring that users retain control over their data while enabling high-precision applications to operate effectively.

10. Conclusion

We conclude that it is feasible to protect against fingerprinting based on the unique built-in errors of sensors by masking these errors with randomly generated noise. Our approach demonstrates that by introducing controlled variability into sensor data, we can effectively disrupt the consistency needed for reliable fingerprinting. This method not only enhances privacy but also maintains the integrity of sensor data for general use, ensuring that the overall functionality of the device is not compromised.

Furthermore, implementing an additional permission for apps to access unobfuscated sensor data, provided the user grants explicit consent, ensures that user experience is not compromised for applications requiring precise sensor measurements. This permission mechanism allows users to balance their privacy needs with the functional demands of specific apps, granting them control over which applications can bypass the noise generation.

By offering this user-controlled flexibility, we ensure that critical applications can operate without degradation in performance while still protecting against potential privacy breaches. This combined strategy of noise generation and user-controlled permissions provides a comprehensive and balanced solution, enhancing privacy without sacrificing the functionality and accuracy required by essential applications. Through this approach, we can significantly increase the difficulty for adversaries attempting to create a consistent fingerprint of our device based on the sensors used.

By introducing controlled noise into sensor data, we effectively disrupt the stable and unique patterns that adversaries rely on to identify and track devices. This noise generation method ensures that the sensor readings vary enough to prevent the formation of a reliable fingerprint, thereby enhancing the privacy and security of the device.

Bibliography

[1] Irene Amerini, Rudy Becarelli, Roberto Caldelli, Alessio Melani, and Moreno Niccolai. "Smartphone fingerprinting combining features of on-board sensors". In: *IEEE Transactions on Information Forensics and Security* (2017). URL: https://ieeexplore.ieee.org/abstract/document/7934347.

[2] Gianmarco Baldini and Gary Steri. "A survey of techniques for the identification of mobile phones using the physical fingerprints of the built-in components". In: *IEEE Communications Surveys & Tutorials* (2017). URL: https://ieeexplore.ieee.org/abstract/document/7902125.

[3] Hristo Bojinov, Yan Michalevsky, Gabi Nakibly, and Dan Boneh. "Mobile device identification via sensor fingerprinting". In: *arXiv preprint arXiv:1408.1416* (2014). URL: https://arxiv.org/abs/1408.1416.

[4] Oscar Casarez-Ruiz. "Sensor Fingerprinting of Mobile Devices". In: (2021). URL: https://scholarworks.calstate.edu/downloads/z316q659n.

[5] Anupam Das, Gunes Acar, Nikita Borisov, and Amogh Pradeep. "The Web's Sixth Sense: A Study of Scripts Accessing Smartphone Sensors". In: 2018. URL: https://doi.org/0.1145/3243734.3243860.

[6] Anupam Das and Nikita Borisov. "Poster: Fingerprinting smartphones through speaker". In: 2014. URL: https://www.ieee-security.org/TC/SP2014/posters/DASAN.pdf.

[7] Anupam Das, Nikita Borisov, and Matthew Caesar. "Exploring ways to mitigate sensor-based smartphone fingerprinting". In: *arXiv preprint arXiv:1503.01874* (2015). URL: https://arxiv.org/abs/1503.01874.

[8] Anupam Das, Nikita Borisov, and Matthew Caesar. "Tracking Mobile Web Users Through Motion Sensors: Attacks and Defenses." In: 2016. URL: https://www.ndss-symposium.org/wp-content/uploads/2017/09/tracking-mobile-web-users-through-motion-sensors-attacks-defenses.pdf.

[9] Anupam Das, Nikita Borisov, and Edward Chou. "Every move you make: Exploring practical issues in smartphone motion sensor fingerprinting and countermeasures". In: (2018).

[10] Anupam Das, Nikita Borisov, Edward Chou, and Muhammad Haris Mughees. "Smartphone fingerprinting via motion sensors: Analyzing feasibility at large-scale and studying real usage patterns". In: *arXiv preprint arXiv:1605.08763* (2016). URL: https://arxiv.org/abs/1605.08763.

[11] Google Developers. *Privacy in Android 10*. 2023. URL: https://developer.android.com/about/versions/10/privacy (visited on 05/10/2024).

[12] Google Developers. *Privacy in Android 11*. 2023. URL: https://developer.android.com/about/versions/11/privacy (visited on 05/10/2024).

[13] Google Developers. *Sensor Class*. 2024. URL: https://developer.android.com/reference/android/hardware/SensorEvent (visited on 05/10/2024).

[14] Google Developers. *SensorEventListener Class*. 2024. URL: https://developer.android.com/reference/android/hardware/SensorEventListener (visited on 05/10/2024).

[15] Google Developers. *SensorManager Class*. 2024. URL: https://developer.android.com/reference/android/hardware/SensorManager (visited on 05/10/2024).

[16] Sanorita Dey, Nirupam Roy, Wenyuan Xu, Romit Roy Choudhury, and Srihari Nelakuditi. "AccelPrint: Imperfections of Accelerometers Make Smartphones Trackable." In: 2014. URL: https://web.archive.org/web/20160428235950id_/http://web.engr.illinois.edu:80/~sdey4/AccelPrint_Presentation_SanoritaDey_withoutBackup.pdf.

[17] Florian Draschbacher. "A2P2-An Android Application Patching Pipeline Based On Generic Changesets". In: 2023. URL: https://dl.acm.org/doi/abs/10.1145/3600160.3600172.

[18] Amin FaizKhademi, Mohammad Zulkernine, and Komminist Weldemariam. "FPGuard: Detection and prevention of browser fingerprinting". In: 2015. URL: https://link.springer.com/chapter/10.1007/978-3-319-20810-7_21.

[19] Ugo Fiore, Aniello Castiglione, Alfredo De Santis, and Francesco Palmieri. "Countering browser fingerprinting techniques: Constructing a fake profile with google chrome". In: 2014. URL: https://ieeexplore.ieee.org/abstract/document/7023976.

[20] Gh05t-1337. *krassesSpiel*. Version 1.1.4. Apr. 3, 2022. URL: https://github.com/Gh05t-1337/krassesSpiel.

[21] Alejandro Gómez-Boix, Pierre Laperdrix, and Benoit Baudry. "Fingerprinting mobile devices: A short analysis". In: 2017. URL: https://inria.hal.science/hal-01611101/.

[22] Chang-Tsun Li. "Source camera identification using enhanced sensor pattern noise". In: *IEEE Transactions on Information Forensics and Security* (2010). URL: https://ieeexplore.ieee.org/abstract/document/5439866.

[23] Mark Huasong Meng, Qing Zhang, Guangshuai Xia, Yuwei Zheng, Yanjun Zhang, Guangdong Bai, Zhi Liu, Sin G Teo, and Jin Song Dong. "Post-GDPR threat hunting on android phones: dissecting OS-level safeguards of user-unresettable identifiers". In: 2023. URL: https://baigd.github.io/files/NDSS23-U2I2.pdf.

[24] Nick Nikiforakis, Wouter Joosen, and Benjamin Livshits. "Privaricator: Deceiving fingerprinters with little white lies". In: 2015. URL: https://dl.acm.org/doi/abs/10.1145/2736277.2741090.

[25] Gerald Palfinger and Bernd Prünster. "AndroPRINT: Analysing the Fingerprintability of the Android API". In: 2020. URL: https://doi.org/10.1145/3407023.3407055.

[26] Libor Polčák, Marek Saloň, Giorgio Maone, Radek Hranickỳ, and Michael McMahon. "JShelter: Give Me My Browser Back". In: *arXiv preprint arXiv:2204.01392* (2022). URL: https://arxiv.org/abs/2204.01392.

[27] Tomáš Repčík. *SensorBox*. Version 4.3.2. Feb. 21, 2024. URL: https://github.com/Foxpace/SensorBox.

[28] Christof Ferreira Torres, Hugo Jonker, and Sjouke Mauw. "FP-Block: usable web privacy by controlling browser fingerprinting". In: 2015. URL: https://link.springer.com/chapter/10.1007/978-3-319-24177-7_1.

[29] Randika Upathilake, Yingkun Li, and Ashraf Matrawy. "A classification of web browser fingerprinting techniques". In: 2015. URL: https://ieeexplore.ieee.org/abstract/document/7266460.

[30] Wenjia Wu, Jianan Wu, Yanhao Wang, Zhen Ling, and Ming Yang. "Efficient fingerprinting-based android device identification with zero-permission identifiers". In: *IEEE Access* (2016). URL: https://ieeexplore.ieee.org/abstract/document/7738449.

[31] Jiexin Zhang, Alastair R Beresford, and Ian Sheret. "Factory calibration fingerprinting of sensors". In: *IEEE Transactions on Information Forensics and Security* (2020). URL: https://ieeexplore.ieee.org/abstract/document/9265280.

[32] Jiexin Zhang, Alastair R. Beresford, and Ian Sheret. "SensorID: Sensor Calibration Fingerprinting for Smartphones". In: 2019. URL: https://ieeexplore.ieee.org/abstract/document/8835276.

A. Appendix

```java
/**
* Intercepts the function call to this function and replaces with the addListener.
* @see SensorManager
* @see PatchInstanceMethod
* @return true on success
*/
@PatchInstanceMethod
public static boolean registerListener(SensorManager sm, SensorEventListener listener,
    Sensor sensor, int samplingPeriodUs) {
    return addListener(sm, listener, sensor, samplingPeriodUs, Integer.MIN_VALUE, null);
}

/**
* Intercepts the function call to this function and replaces with the removeListener.
* @see SensorManager
* @see PatchInstanceMethod
*/
@PatchInstanceMethod
public static void unregisterListener(SensorManager sm, SensorEventListener listener,
    Sensor sensor) {
    removeListener(sm, listener, sensor);
}
```

Listing A.1: Intercept Methods

```
/**
* Selects offset and gain for the appropriate sensor and
* applies noise to the value if the right sensor is read.
* @param event SensorEvent.
* @see SensorEvent
*/
private static void manipulateValues(SensorEvent event) {
  final float offset = lambda_offsets.getOrDefault(event.sensor.getType(), 0.0f);
  final float gain = lambda_gains.getOrDefault(event.sensor.getType(), 0.0f);

  switch(event.sensor.getType()) {
    case Sensor.TYPE_ACCELEROMETER:
    case Sensor.TYPE_GYROSCOPE:
      event.values[AXIS_X] = applyNoise(event.values[AXIS_X], offset, gain);
      event.values[AXIS_Y] = applyNoise(event.values[AXIS_Y], offset, gain);
      event.values[AXIS_Z] = applyNoise(event.values[AXIS_Z], offset, gain);
      break;
    default:
      break;
  }
}
```

Listing A.2: Noise Generating Function

```
/**
* Applies noise to the original sensor value.
* @param original Original sensor value.
* @param lambda_offset +/- offset to be applied to the original value.
* @param lambda_gain 1 +/- gain to be applied to the original value.
* @return float obscured sensor value.
*/
private static float applyNoise(final float original, final float lambda_offset, final
    float lambda_gain) {
  final float offset = generateRandomValue(0 - Math.abs(lambda_offset),
                                           0 + Math.abs(lambda_offset));
  final float gain = generateRandomValue(1 - Math.abs(lambda_gain),
                                         1 + Math.abs(lambda_gain));

  return (original - offset) / gain;
}
```

Listing A.3: Random Value Generation Function

```java
/**
  * Implements the abstract method from the SensorEventListener.
  * Calls the same function of the original listener after manipulating the received
    SensorEvent and passes is down.
  * @see SensorEventListener
  * @see SensorEvent
  * @see Patch
  */
@Override
public void onSensorChanged(SensorEvent event) {
    Patch.manipulateValues(event);
    listener.onSensorChanged(event);
}

/**
  * Implements the abstract method from the SensorEventListener.
  * Calls the same function of the original listener passes down the received parameters.

  * @see SensorEventListener
  * @see Sensor
  */
@Override
public void onAccuracyChanged(Sensor sensor, int accuracy) {
    listener.onAccuracyChanged(sensor, accuracy);
}
```

Listing A.4: SensorEventListener Methods

```
def classify(path: str) -> None:
  # Read CSV files containing the measurements
  X, y = loadData(path)

  # Split into tarining and test set
  X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5, random_state
    =42)

  # Initialize the k-NN classifier
  knn = KNeighborsClassifier()

  # Train the classifier
  knn.fit(X_train, y_train)

  # Make predictions on the test set
  y_pred = knn.predict(X_test)

  # Calculate and print the accuracy
  print(f"Accuracy: {accuracy_score(y_test, y_pred):.2f}")
```

Listing A.5: Classifier Function