ISEC

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 Information Security
Graz University of Technology

Graz, February 2025

Abstract

Sensor fingerprinting is a technique that enables the unique identification of 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 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 characteristic 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 applied to Android application packages. 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 Android Application Patching Pipeline we can apply our patch to any compatible android app. We evaluate how varying noise levels affect the effectiveness of our obfuscation technique, finding that while increased noise enhances privacy by reducing fingerprintability, it also compromises the usefulness of sensor values. We conclude that by adding 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

Through the utilization of the Android API, applications can access a wide array of device-specific information essential for their operation. While this functionality is designed to support app performance by providing crucial data, it also exposes certain device features and configurations that, when combined, can be used to uniquely identify users based on the specific characteristics of their device and system settings [24]. The practice of tracking Android users via device fingerprinting, without their knowledge and explicit consent, raises significant privacy and security concerns. This method allows users to be tracked across multiple applications and services, often without their awareness, which constitutes a clear infringement of privacy.

Advertisers frequently employ fingerprinting techniques to compile detailed user profiles for targeted advertising. However, malicious actors may exploit the same fingerprinting information for nefarious purposes, such as orchestrating tailored phishing attacks or deploying malware optimized for the user's specific device configuration. Moreover, the collection of data from the API could reveal a substantial amount of sensitive information. When such data is aggregated, it may lead to a variety of fraudulent activities, leading to both security risks and privacy violations.

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. Some of the used information is based on the system properties of the device, network, or device configuration by the user [19]. These properties can include details about the operating system, kernel version, and runtime parameters [29]. Furthermore, details about the Android build version, manufacturer, model, and build fingerprint can be included in the fingerprint. Additionally, separate hardware components, like camera and speaker, can be uniquely fingerprinted [20, 6].

In this thesis, 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 [12]. These imperfections in sensors, exploited for fingerprinting, arise from manufacturing variations, material defects, calibration inaccuracies, and various environmental factors [2, 6, 30]. Android devices, which are equipped with various sensors like accelerometers and gyroscopes are vulnerable to this type of fingerprinting [5]. 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 [4]. These studies demonstrate that the measurement errors inherent in these sensors are unique to each device, creating a distinctive fingerprint [3, 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. 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. While these papers highlighted the problem there is currently no mitigation implemented in Android. We will address this problem by implementing a possible solution which masks these sensor imperfections.

Our implementation of AndroGUARD builds on suggestions from some of the research papers which covered this issue [5, 1, 3]. To implement and deploy our mitigation we use the Android Application Patching Pipeline (A2P2) [13]. We aim to effectively obfuscate sensor data and prevent the extraction of identifiable information, by integrating the recommended strategy from a reviewed research paper [5]. 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 [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 overview of general 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. Chapter 7 shifts focus to validating the effectiveness of the patch, detailing the testing methodology, results, and areas for further improvement. By highlighting the implications and discussing the limitations of our work, Chapter 8 offers directions for future research. Finally, Chapter 9 summarizes our findings, reflects on the implications of the project, and outlines directions for future research and development.

2. Background

With the help of fingerprinting, users and their habits can be tracked to create personalized profiles of them. However, tracking users based on the fingerprint of their devices raises various privacy and security concerns [21]. 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 collected 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. In the following, we will explore browser and mobile fingerprinting, focusing on their similarities and differences.

2.1. Browser Fingerprinting

Browser fingerprinting is a widely known technique, which is frequently exploited to track users across different websites [25]. 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 of the device.

Browser Fingerprinting Methodologies. Different methods can be applied to fingerprint a browser as shown by Upathilake, Li, and Matrawy [28]. Browsers can be fingerprinted by analyzing various browser-specific attributes, such as the browser version, installed fonts, and other identifiable characteristics. Additionally, HTML5 functionalities, particularly the rendering of text and WebGL scenes via the canvas element, can be leveraged to generate a fingerprint. By rendering content and subsequently retrieving pixel data from the canvas, a unique identifier based on subtle differences in how the browser displays these elements can be constructed. This technique allows for the creation of a detailed fingerprint that can be used to distinguish users across sessions. JavaScript can also be used to create cross-browser fingerprints, which is browser independent and focuses more on the hardware the browser is running on, like screen resolution or networking information. With the use of JavaScript also sensor data can be accessed to create sensor fingerprints.

Browser Fingerprinting Protections. A number of solutions have already been presented in multiple research papers to identify and block fingerprinting efforts, making browsing the internet more private for everyone. JShelter [25] blocks the execution of JavaScript to mitigate the gathering of personal data used for fingerprinting to increase browser security. Another study, PriVaricator [23], aims to reduce the determinism of browser fingerprints by introducing controlled randomization, thereby mitigating the ability to consistently identify users across multiple visits to the same website. This approach disrupts the reliability of fingerprinting techniques by making the data less predictable and harder to track. These research projects disrupt the consistent behavior required by features like adaptive layouts, multimedia rendering, and security mechanisms, leading to broken or degraded functionality. To solve this issue FP-Block [27] creates a different unrelated fingerprint for the embedded resources to maintain their functionality, but prevent tracking.

2.2. Smartphone Fingerprinting

The concepts underlying browser fingerprinting can similarly be applied in the context of smartphone applications [24]. Similarly, mobile fingerprinting relies on the extraction of various device specific characteristics, such as the mobile operating system, device model, screen dimensions, app usage data, and other hardware or software attributes, to construct a unique identifier for the device.

A study by Gómez-Boix, Laperdrix, and Baudry, about mobile browsers, highlights the differences the distinct behaviors in the absence of plugins like Flash [18]. 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 devices. 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.

The aforementioned paper primarily focuses on the use of web browsers on mobile devices as a context for fingerprinting; however, it is important to recognize that web browsers are not the only environments in which fingerprinting techniques can be applied. Fingerprinting can occur in a variety of different contexts, including mobile applications. In the subsequent paragraphs we will focus on methods used to identify and track users through mobile applications.

There is a novel approach to identify Android devices through fingerprinting methods that bypass the need for user-granted permissions. It leverages zero-permission identifiers, such as hardware configurations, sensor data, and system settings, which can be accessed without explicit user consent. By combining these attributes, a fingerprint that uniquely

identifies a device across different applications and sessions can be developed. This technique is highly efficient, requiring minimal computational resources while achieving accurate identification. The study written about this methodology highlights the significant privacy risks associated with such methods, as they allow tracking of users without traditional permission mechanisms or user awareness [29].

In a similar fashion mobile devices can be uniquely identified through their personalized configurations, even without direct access to hardware identifiers. It was demonstrated that a combination of software and user-specific settings, such as installed apps, language preferences, timezone settings, and network configurations, can be used to create a distinct fingerprint for each device. This method allows accurate device tracking across different platforms and applications. The paper that studied this approach raised concerns about privacy, emphasizing that even seemingly benign configurations can reveal a great deal about a user and facilitate tracking without their consent [19].

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 [7, 21]. Users have the freedom to review and modify app permissions through system settings, enabling them to manage access to personal information effectively. When supported by 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 [22]. Despite the current permission system, there remains a significant possibility for fingerprinting through, for instance, the use of sensor values. Therefore, the following chapter will describe how a permissionless fingerprinting technique can be used to track users without restrictions from the Android operating system and the knowledge of the user.

3. Sensor Fingerprinting

In this thesis, our central focus is the complex problem of sensor fingerprinting applicable to 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 introduce potential vulnerabilities related to the misuse of sensor data. When exploited through fingerprinting techniques, such data can compromise user privacy and security [21].

It has been shown by multiple research papers that the built-in error of sensors remains consistent over the lifetime of the device [31, 12]. 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. Multiple studies [6, 5, 4, 3] 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 [4, 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 [12, 31, 2]. Unlike audio-based fingerprinting, which needs a known signal, accelerometer fingerprinting relies on passive background measurements taken when the device is stationary. When the device is at rest, the acceleration vector should equal the gravitational constant, making detection of sensor imperfections 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 [6]. 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. If the installed browser application has permissions to access sensor data, so do the websites viewed within the application. These 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 application. By intercepting and modifying Android API queries before they are processed by the browser application, it is possible to alter the values that are retrieved by JavaScript executed within the browser. This manipulation enables the alteration of system-level information that would otherwise be accurately reported to the browser, thereby controlling or obfuscating the data provided to web applications and potentially preventing the collection of fingerprintable attributes.

To develop our patch, we analyze sensor fingerprinting techniques used across Android devices, focusing on how unique sensor signatures are exploited to track users. By understanding these mechanisms, we aim to mitigate risks through controlled randomness or masking, disrupting consistent data collection and enhancing privacy. Our methodology involves intercepting and modifying sensor values, based on literature review, supported by patch development, and testing to provide a solution that empowers users to maintain control over their personal data.

4. Methodology

We investigate sensor fingerprinting and develop a countermeasure, detailing the approach, tools, and processes used in our study. A number of solutions are already present in order to protect browsers against fingerprinting [25, 14, 23, 27]. One of them is to monitor and restrict access to properties commonly used for fingerprinting and prevent network traffic to tracking servers [25]. This includes creating 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 [15]. Nevertheless, to the best of our knowledge, the number of available solutions for the protection of Android devices is limited.

The Android API has implemented a couple of proactive measures to counteract privacy violations [7]. This includes regulated control over access to identifying information and a robust permission system. Android ensures that applications can only request and use the permissions that are explicitly granted to them [8]. These measures aim to make it difficult for apps and services to collect user-related and privacy-sensitive data by restricting access to certain APIs and data points. However, they do not safeguard characteristics that remain accessible, such as sensor values [29]. As a result, while some data points are protected, others can still be exploited for purposes like device fingerprinting, leaving privacy vulnerabilities. In the following we examine the proposed solutions, evaluate their limitations, and select the most appropriate option for our use case.

4.1. Proposed Solutions

There are two proposed countermeasures designed to enhance the privacy of sensor data and protect against fingerprinting: calibration and noise generation [5]. Each of these methods addresses the issue of sensor data consistency in distinct ways, offering a comprehensive approach to mitigate the risk of device fingerprinting.

Calibration. Calibration involves the systematic adjustment of sensor readings to account for and eliminate inherent biases and errors. By 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 Generation. This 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 noise, the sensor data becomes less predictable and more resistant to fingerprinting efforts.

4.2. Challenges

Both methods, calibration and noise generation, have their own set of limitations 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 generation. 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. 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 functionality of apps.

This balance can be challenging to achieve, as different applications and sensor types have varying tolerance levels for noise.

4.3. 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 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. Approach

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 [31, 12, 4]. 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 Methodology

Our methodology utilizes the Android Application Patching Pipeline (A2P2) [13], 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 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 Sensor API

After examining the Android API, we observe that every application utilizing sensor values must implement the abstract class `SensorEventListener` [10]. 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` [11] 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 [9]. 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 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 to the application.

This process ensures that while the application continues to receive the sensor data it needs to function properly, the data have been obfuscated to prevent the creation of a consistent and unique fingerprint based on the sensor readings. This method effectively disrupts attempts, when any custom class derived from the `SensorEventListener` class is used, to utilize these sensor values for fingerprinting purposes.

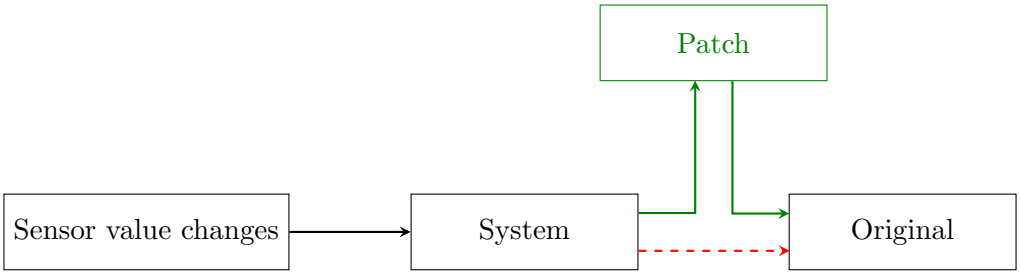


Figure 5.1.: The function calls from the system are intercepted by our patch and forwarded after modification to the original function.

5.3. Noise Generation

$$value_{new} = \frac{(value_{old} - offset_{sensor})}{gain_{sensor}} \tag{5.1}$$

To create the noise, we employ a formula (5.1) adapted from a study by Das, Borisov, and Caesar [5]. 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.

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 attempts 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.

The added noise 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 approach strengthens the overall security framework by protecting not only native applications but also web-based interactions. It offers users increased control over their sensor data, thereby reinforcing digital privacy and reducing the risk of unauthorized data collection across various platforms.

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 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.

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, on the users demand, without the need of reinstalling the unpatched version of the software. 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 ensures 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 introduce controlled randomness into the sensor readings, thereby disrupting attempts to create a stable and consistent fingerprint based on the sensor data.

Intercept Method. This function, shown in Appendix A.2, serves as the central hub of the patch, responsible for applying noise to the sensor data. 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. 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, displayed in A.3. 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 (A.4) 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 an application is sufficiently obfuscated to mitigate fingerprinting attempts while maintaining the functionality required for legitimate uses.

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

7. Evaluation

The following sections assess our methodology in terms of functionality, effectiveness, and usability. For testing we selected an APK, 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 serves 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.

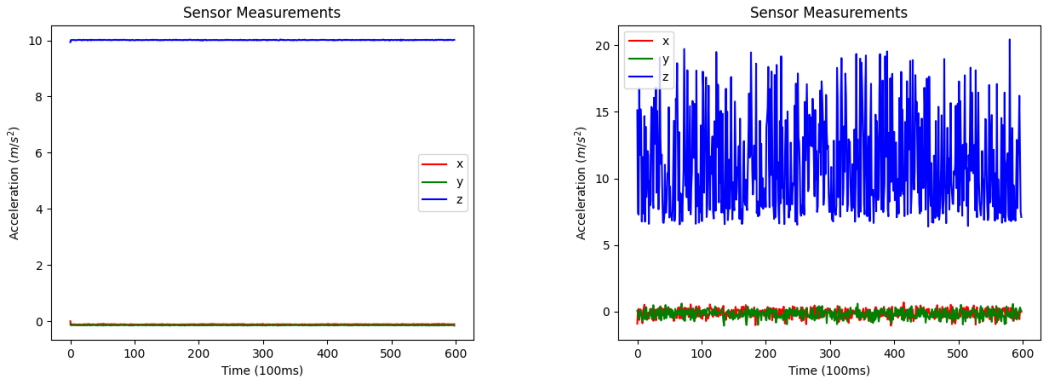
First, the functionality of the patch is validated through a process involving its application to an app called *SensorBox* [26], which is designed to gather sensor measurements. In order to test not only the functionality but also the effectiveness of the patch we created a simple data gathering app [16] to collect unaltered and altered data at the same time. During the evaluation 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. These recorded values are fed into a machine learning algorithm [1]. The training dataset is utilized to train a k-nearest neighbors (knn) or random forest classification algorithm, which are standard methods for assessing the variation of data points. By training the algorithms 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. We apply gridsearch to determine the best possible parameters for the highest accuracy. 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 reduced fingerprintability. The recorded precision quantifies the proportion of correctly predicted positive instances relative to the total number of positive predictions made by the model. A higher precision value indicates a lower rate of false positives, reflecting better prediction accuracy for positive classifications. The resulted recall value evaluates the model's ability to identify all relevant instances, with higher values indicating fewer false negatives. The f1-score provides a harmonic mean of precision and recall, offering a balanced metric that accounts for both false positives and false negatives. 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.

7.1. Testing

Our testing process involved around ten devices running various versions of Android from 10 to the latest current release of Android 14. 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.

7.1.1. Functionality

To evaluate the applicability of the patch to existing applications, we selected the simple app *SensorBox* as a test case. The purpose of this test was to determine whether *SensorBox* could successfully receive and process the modified sensor values generated by the applied update. Upon implementing the patch, we observed that *SensorBox* seamlessly integrated the patched values without encountering any errors or operational issues. This successful application demonstrates the patch's compatibility with established applications and suggests its potential for broader adoption in similar mobile software, particularly those reliant on sensor data.



(a) Values before the patch

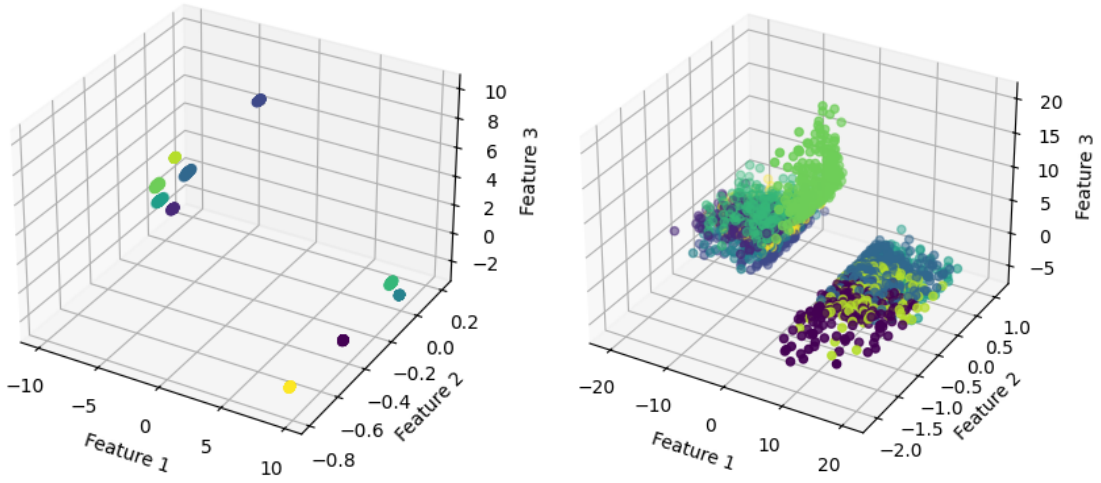
(b) Values after the patch

Figure 7.1.: Acceleration sensor values measured before and after the application of the patch on the same device.

7.1.2. 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. Comparing the consistency of the sensor readouts before and after applying the patch, a significant change can be observed in Figure 7.1, mirroring the findings of previous research studies. Before the application of our patch, 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 as it is visualized in Figure 7.2.



(a) knn decision boundaries before applying noise

(b) knn decision boundaries after applying noise

Figure 7.2.: 3D visualization of knn decision boundaries.

When the classification algorithm used the unpatched, unmasked data it was able to associate all of the devices with their measurement with 100% confidence. After applying the patch and retraining the algorithm it was only able to match the devices with 72.916% accuracy, even with our few samples of ten devices as shown in Figure 7.1. When analyzing the classification results for each individual device, it becomes clear that the performance metrics all declined, except one, from their previous values of 100% as shown in Figure 7.2. This reduction highlights a shift in the effectiveness of our fingerprinting method when applied to devices after being patched, making it more difficult to maintain a consistent fingerprint across different devices.

The observed drop in these metrics indicates that the classifier's ability to distinguish device-specific patterns has been compromised. This is due to the introduced variability by our patch in sensor data across devices. Our results demonstrate a low incidence of false positives, indicating a high degree of confidence in the classification of the observed instances. Notably, this level of accuracy was maintained even when distinguishing between devices of the same make and model, underscoring the robustness and reliability of this simple fingerprinting approach. These findings suggest that the model is capable of effectively identifying subtle differences between devices, even when hardware and software configurations are largely identical.

We believe that this accuracy would decrease even more with the use of a lot more devices, increasing the disruption of consistent error values. 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.

	precision	recall	f1-score
accuracy			1.00
macro avg	1.00	1.00	1.00
weighted avg	1.00	1.00	1.00

(b) Results before obfuscation

	precision	recall	f1-score
accuracy			0.73
macro avg	0.73	0.73	0.73
weighted avg	0.73	0.73	0.73

(c) Results after obfuscation

Table 7.1.: The performance outcomes of the knn classifier over the test dataset.

	precision	recall	f1-score	support
Galaxy A34 5G _01	0.82	0.66	0.73	74
Galaxy A54 5G _01	0.59	0.60	0.59	82
Galaxy A55 5G _01	0.95	0.96	0.95	76
Galaxy M23 5G _01	0.44	0.37	0.40	68
Galaxy M23 5G _02	0.61	0.65	0.63	66
Galaxy S21 5G _01	0.59	0.58	0.58	76
Galaxy S23 _02	0.77	0.94	0.85	62
Galaxy Tab S10+ _01	0.72	0.76	0.74	72
Galaxy Tab S9 5G _01	0.77	0.80	0.78	75
Nokia 6.1 _01	1.00	1.00	1.00	69

Table 7.2.: Detailed results of the knn classifier for each individual device.

7.1.3. Usability

The patch was applied to a motion-controlled game [17] 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.

7.2. Noise Level Adjustment

Modifying the level of noise applied to sensor data enhances resistance against device fingerprinting, though it presents trade-offs that impact application functionality. By increasing the noise intensity, it becomes more difficult for fingerprinting algorithms to reliably distinguish unique device signatures as shown in Figure 7.3, thereby strengthening user privacy. In our testing, increasing the gain of the sensor values resulted in a more pronounced decrease in the effectiveness of fingerprinting methods. This adjustment disrupted the distinctiveness of sensor data, thereby enhancing privacy by making it more challenging for algorithms to identify unique device signatures. In contrast, modifying the offset applied to the sensor readouts had a relatively limited impact on the fingerprinting process. The offset adjustment did not significantly alter the underlying characteristics of the sensor data, thus allowing more identifiable patterns to remain detectable. However, excessive noise disrupts the accuracy and reliability of sensor data, which many applications rely upon for core functions, rendering them less usable or even unusable. Conversely, reducing the noise level to preserve application functionality diminishes the effectiveness of fingerprinting protection as demonstrated in Figure 7.4. Achieving an optimal balance between these competing priorities is thus important.

	precision	recall	f1-score
accuracy			0.51
macro avg	0.52	0.52	0.52
weighted avg	0.52	0.51	0.51

(b) Results with high gain

	precision	recall	f1-score
accuracy			0.41
macro avg	0.41	0.41	0.41
weighted avg	0.41	0.41	0.41

(c) Results with high gain and offset

	precision	recall	f1-score
accuracy			0.73
macro avg	0.73	0.73	0.73
weighted avg	0.73	0.73	0.73

(e) Results with default offset and gain

	precision	recall	f1-score
accuracy			0.54
macro avg	0.55	0.55	0.55
weighted avg	0.55	0.54	0.54

(f) Results with high offset

Table 7.3.: knn classifier results after applying the patch with increased noise levels.

	precision	recall	f1-score
accuracy			0.78
macro avg	0.79	0.78	0.78
weighted avg	0.78	0.78	0.78

(b) Results with low gain

	precision	recall	f1-score
accuracy			0.73
macro avg	0.73	0.73	0.73
weighted avg	0.73	0.73	0.73

(c) Results with default gain and offset

	precision	recall	f1-score
accuracy			0.91
macro avg	0.91	0.91	0.91
weighted avg	0.91	0.91	0.91

(e) Results with low offset and gain

	precision	recall	f1-score
accuracy			0.81
macro avg	0.81	0.81	0.81
weighted avg	0.81	0.81	0.81

(f) Results with low offset

Table 7.4.: knn classifier results after applying the patch with decreased noise levels.

8. Discussion & Limitations

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. 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. Unfortunately, our access to a diverse range of devices was limited. This scarcity of devices hindered our ability to perform extensive and comprehensive testing.

Previous studies have often introduced additional, artificially derived features to improve the reliability and accuracy of fingerprinting techniques [6, 30]. By contrast, our approach was tested solely by training our algorithm on the combined raw sensor readouts, without incorporating any supplementary, artificially generated or preprocessed features. This methodology allows for a direct assessment of the effectiveness of our approach based purely on the available sensor data. Consequently, our results provide insights into the potential of raw sensor readouts alone to serve as a basis for fingerprinting, in contrast to methods that rely on additional feature engineering to enhance accuracy.

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 in the robustness of our patch design and the alignment of our methods with those proven effective in existing research.

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.

9. Conclusion

Fingerprinting devices based on sensor value imperfections poses a significant privacy concern, as these subtle errors, often caused by manufacturing variances or environmental factors, can be exploited to uniquely identify and track devices. Unlike traditional identifiers, sensor-based fingerprints do not require explicit user permissions, making them a covert and persistent threat to user anonymity.

Our approach addresses the problem of consistent sensor error values which can be used for fingerprinting. We intercept sensor readouts with the help of the A2P2 framework and manipulate them with added noise. After limited testing, which showed promising tendencies, 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.

By introducing artificial noise, the distinctive patterns and unique characteristics that these sensors typically exhibit become harder to identify, which complicates efforts to track or profile devices. 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 heavily compromised. While the added noise effectively disrupts fingerprinting efforts, it simultaneously degrades the accuracy and reliability of the sensor data. Excessive noise can impair many applications depending on accurate sensor data for activities such as navigation, environmental monitoring, or gesture recognition, reducing user experience and the app's practical utility. Thus, while noise augmentation can provide privacy benefits, it must be carefully balanced against the need to maintain the operational effectiveness of applications.

Furthermore, implementing an additional permission for apps to access unobfuscated sensor data, could provide the user to grant explicit consent on demand, ensuring that user experience is not compromised for applications requiring precise sensor measurements. By offering this user-controlled flexibility, we could 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 devices based on the sensors used.

Bibliography

[1] Irene Amerini, Rudy Becarelli, Roberto Caldelli, Alessio Melani, and Moreno Niccolai. "Smartphone Fingerprinting Combining Features of On-Board Sensors". In: *IEEE Trans. Inf. Forensics Secur.* 12.10 (2017), pp. 2457–2466. DOI: 10.1109/TIFS.2017.2708685. URL: https://doi.org/10.1109/TIFS.2017.2708685.

[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 Commun. Surv. Tutorials* 19.3 (2017), pp. 1761–1789. DOI: 10.1109/COMST.2017.2694487. URL: https://doi.org/10.1109/COMST.2017.2694487.

[3] Anupam Das, Gunes Acar, Nikita Borisov, and Amogh Pradeep. "The Web's Sixth Sense: A Study of Scripts Accessing Smartphone Sensors". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. Ed. by David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang. ACM, 2018, pp. 1515–1532. DOI: 10.1145/3243734.3243860. URL: https://doi.org/10.1145/3243734.3243860.

[4] Anupam Das, Nikita Borisov, and Matthew Caesar. "Do You Hear What I Hear?: Fingerprinting Smart Devices Through Embedded Acoustic Components". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*. Ed. by Gail-Joon Ahn, Moti Yung, and Ninghui Li. ACM, 2014, pp. 441–452. DOI: 10.1145/2660267.2660325. URL: https://doi.org/10.1145/2660267.2660325.

[5] Anupam Das, Nikita Borisov, and Matthew Caesar. "Tracking Mobile Web Users Through Motion Sensors: Attacks and Defenses". In: *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society, 2016. URL: http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/tracking-mobile-web-users-through-motion-sensors-attacks-defenses.pdf.

[6] Anupam Das, Nikita Borisov, and Edward Chou. "Every Move You Make: Exploring Practical Issues in Smartphone Motion Sensor Fingerprinting and Countermeasures". In: *Proc. Priv. Enhancing Technol.* 2018.1 (2018), pp. 88–108. DOI: 10.1515/POPETS-2018-0005. URL: https://doi.org/10.1515/popets-2018-0005.

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

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

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

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

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

[12] Sanorita Dey, Nirupam Roy, Wenyuan Xu, Romit Roy Choudhury, and Srihari Nelakuditi. "AccelPrint: Imperfections of Accelerometers Make Smartphones Trackable". In: *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*. The Internet Society, 2014. URL: https://www.ndss-symposium.org/ndss2014/accelprint-imperfections-accelerometers-make-smartphones-trackable.

[13] Florian Draschbacher. "A2P2 - An Android Application Patching Pipeline Based On Generic Changesets". In: *Proceedings of the 18th International Conference on Availability, Reliability and Security, ARES 2023, Benevento, Italy, 29 August 2023- 1 September 2023*. ACM, 2023, 1:1–1:11. DOI: 10.1145/3600160.3600172. URL: https://doi.org/10.1145/3600160.3600172.

[14] Amin FaizKhademi, Mohammad Zulkernine, and Komminist Weldemariam. "FPGuard: Detection and Prevention of Browser Fingerprinting". In: *Data and Applications Security and Privacy XXIX - 29th Annual IFIP WG 11.3 Working Conference, DBSec 2015, Fairfax, VA, USA, July 13-15, 2015, Proceedings*. Ed. by Pierangela Samarati. Vol. 9149. Lecture Notes in Computer Science. Springer, 2015, pp. 293–308. DOI: 10.1007/978-3-319-20810-7_21. URL: https://doi.org/10.1007/978-3-319-20810-7%5C_21.

[15] Ugo Fiore, Aniello Castiglione, Alfredo De Santis, and Francesco Palmieri. "Countering Browser Fingerprinting Techniques: Constructing a Fake Profile with Google Chrome". In: *17th International Conference on Network-Based Information Systems, NBiS 2014, Salerno, Italy, September 10-12, 2014*. Ed. by Leonard Barolli, Fatos Xhafa, Makoto Takizawa, Tomoya Enokido, Aniello Castiglione, and Alfredo De Santis. IEEE Computer Society, 2014, pp. 355–360. DOI: 10.1109/NBIS.2014.102. URL: https://doi.org/10.1109/NBiS.2014.102.

[16] Kranz Gergö. *SensorPRINT*. Version 3.1. Nov. 6, 2024. URL: https://github.com/KGeri201/SensorPRINT.

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

[18] Alejandro Gómez-Boix, Pierre Laperdrix, and Benoit Baudry. "Hiding in the Crowd: an Analysis of the Effectiveness of Browser Fingerprinting at Large Scale". In: *Proceedings of the 2018 World Wide Web Conference on World Wide Web, WWW 2018, Lyon, France, April 23-27, 2018*. Ed. by Pierre-Antoine Champin, Fabien

Gandon, Mounia Lalmas, and Panagiotis G. Ipeirotis. ACM, 2018, pp. 309–318. DOI: 10.1145/3178876.3186097. URL: https://doi.org/10.1145/3178876.3186097.

[19] Andreas Kurtz, Hugo Gascon, Tobias Becker, Konrad Rieck, and Felix C. Freiling. "Fingerprinting Mobile Devices Using Personalized Configurations". In: *Proc. Priv. Enhancing Technol.* 2016.1 (2016), pp. 4–19. DOI: 10.1515/POPETS-2015-0027. URL: https://doi.org/10.1515/popets-2015-0027.

[20] Chang-Tsun Li. "Source camera identification using enhanced sensor pattern noise". In: *IEEE Trans. Inf. Forensics Secur.* 5.2 (2010), pp. 280–287. DOI: 10.1109/TIFS.2010.2046268. URL: https://doi.org/10.1109/TIFS.2010.2046268.

[21] 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: *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023*. The Internet Society, 2023. URL: https://www.ndss-symposium.org/ndss-paper/post-gdpr-threat-hunting-on-android-phones-dissecting-os-level-safeguards-of-user-unresettable-identifiers/.

[22] Solomon Negash and Hossain Shahriar. "Mobile app permissions awareness". In: *5th International Conference on Information & Communication Technology and Accessibility, ICTA 2015, Marrakech, Morocco, December 21-23, 2015*. IEEE, 2015, pp. 1–4. DOI: 10.1109/ICTA.2015.7426873. URL: https://doi.org/10.1109/ICTA.2015.7426873.

[23] Nick Nikiforakis, Wouter Joosen, and Benjamin Livshits. "PriVaricator: Deceiving Fingerprinters with Little White Lies". In: *Proceedings of the 24th International Conference on World Wide Web, WWW 2015, Florence, Italy, May 18-22, 2015*. Ed. by Aldo Gangemi, Stefano Leonardi, and Alessandro Panconesi. ACM, 2015, pp. 820–830. DOI: 10.1145/2736277.2741090. URL: https://doi.org/10.1145/2736277.2741090.

[24] Gerald Palfinger and Bernd Prünster. "AndroPRINT: analysing the fingerprintability of the Android API". In: *ARES 2020: The 15th International Conference on Availability, Reliability and Security, Virtual Event, Ireland, August 25-28, 2020*. Ed. by Melanie Volkamer and Christian Wressnegger. ACM, 2020, 94:1–94:10. DOI: 10.1145/3407023.3407055. URL: https://doi.org/10.1145/3407023.3407055.

[25] Libor Polcák, Marek Salon, Giorgio Maone, Radek Hranický, and Michael McMahon. "JShelter: Give Me My Browser Back". In: *Proceedings of the 20th International Conference on Security and Cryptography, SECRYPT 2023, Rome, Italy, July 10-12, 2023*. Ed. by Sabrina De Capitani di Vimercati and Pierangela Samarati. SCITEPRESS, 2023, pp. 287–294. DOI: 10.5220/0011965600003555. URL: https://doi.org/10.5220/0011965600003555.

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

[27] Christof Ferreira Torres, Hugo L. Jonker, and Sjouke Mauw. "FP-Block: Usable Web Privacy by Controlling Browser Fingerprinting". In: *Computer Security - ESORICS 2015 - 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21-25, 2015, Proceedings, Part II*. Ed. by Günther Pernul, Peter Y. A. Ryan, and Edgar R. Weippl. Vol. 9327. Lecture Notes in Computer Science. Springer, 2015, pp. 3–19. DOI: 10.1007/978-3-319-24177-7_1. URL: https://doi.org/10.1007/978-3-319-24177-7%5C_1.

[28] Randika Upathilake, Yingkun Li, and Ashraf Matrawy. "A classification of web browser fingerprinting techniques". In: *7th International Conference on New Technologies, Mobility and Security, NTMS 2015, Paris, France, July 27-29, 2015*. Ed. by Mohamad Badra, Azzedine Boukerche, and Pascal Urien. IEEE, 2015, pp. 1–5. DOI: 10.1109/NTMS.2015.7266460. URL: https://doi.org/10.1109/NTMS.2015.7266460.

[29] Wenjia Wu, Jianan Wu, Yanhao Wang, Zhen Ling, and Ming Yang. "Efficient Fingerprinting-Based Android Device Identification With Zero-Permission Identifiers". In: *IEEE Access* 4 (2016), pp. 8073–8083. DOI: 10.1109/ACCESS.2016.2626395. URL: https://doi.org/10.1109/ACCESS.2016.2626395.

[30] Jiexin Zhang, Alastair R. Beresford, and Ian Sheret. "Factory Calibration Fingerprinting of Sensors". In: *IEEE Trans. Inf. Forensics Secur.* 16 (2021), pp. 1626–1639. DOI: 10.1109/TIFS.2020.3039685. URL: https://doi.org/10.1109/TIFS.2020.3039685.

[31] Jiexin Zhang, Alastair R. Beresford, and Ian Sheret. "SensorID: Sensor Calibration Fingerprinting for Smartphones". In: *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 2019, pp. 638–655. DOI: 10.1109/SP.2019.00072. URL: https://doi.org/10.1109/SP.2019.00072.

A. Appendix

```
/**
* Checks if a PatchListener is already associated with the SensorEventListener received.
* Creates new one of not or binds additional sensors to the existing ones.
* @param listener the original SensorEventListener instance.
* @param sensor the Sensor the listener should receive values from.
* @see SensorEventListener
* @see Sensor
* @see PatchListener
* @return return custom PatchListener object containing the original listener.
*/
private static SensorEventListener addListener(SensorEventListener listener, Sensor
    sensor) {
  PatchListener plistener = PATCH_MAPPINGS.stream().filter(pl -> pl.getListener().equals
    (listener)).findAny().orElse(new PatchListener(listener));
  plistener.sensors.add(sensor);
  PATCH_MAPPINGS.add(plistener);
  return plistener;
}

/**
* Finds the PatchListener is already associated with the SensorEventListener
* received and removes the sensor it is unregistering from.
* Removes the PatchListener if no more Sensors are bound to it.
* @param listener the original SensorEventListener object.
* @param sensor the Sensor the listener should receive values from.
* @see SensorEventListener
* @see Sensor
* @see PatchListener
* @return return custom PatchListener object containing the original listener. If none
    found it returns the original listener
*/
private static SensorEventListener removeListener(SensorEventListener listener, Sensor
    sensor) {
  PatchListener plistener = PATCH_MAPPINGS.stream().filter(pl -> pl.getListener().equals
    (listener)).findAny().orElse(null);
  if (plistener == null) return listener;
  if (sensor != null) plistener.sensors.remove(sensor);
  else plistener.sensors.clear();
  if (plistener.sensors.isEmpty()) return PATCH_MAPPINGS.remove(PATCH_MAPPINGS.indexOf(
    plistener));
  return plistener;
}
```

Listing A.1: Interception Helper Methods

```
/**
* Intercepts the function call to this function and
* calls it with the custom PatchListener wrapper for the original.
* @see SensorManager
* @see PatchInstanceMethod
* @see OriginalMethods
* @return value from original registerListener method
*/
@PatchInstanceMethod
public static boolean registerListener(SensorManager sm, SensorEventListener listener,
    Sensor sensor, int samplingPeriodUs) {
  return OriginalMethods.android_hardware_SensorManager.registerListener(sm, addListener
    (listener, sensor), sensor, samplingPeriodUs);
}

/**
* Intercepts the function call to this function and
* replaces the SensorEventListener object with the registered PatchListener
* before calling the original function.
* @see SensorManager
* @see PatchInstanceMethod
* @see OriginalMethods
*/
@PatchInstanceMethod
public static void unregisterListener(SensorManager sm, SensorEventListener listener) {
  OriginalMethods.android_hardware_SensorManager.unregisterListener(sm, removeListener(
    listener, null));
}
```

Listing A.2: 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 void manipulateValues(SensorEvent event) {
  float offset = Utils.getOffset(event.sensor.getType());
  float gain = Utils.getGain(event.sensor.getType());

  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.3: 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 float applyNoise(final float original, final float lambda_offset, final float
    lambda_gain) {
  float offset = generateRandomValue(0 - Math.abs(lambda_offset),
                                     0 + Math.abs(lambda_offset));
  float gain = generateRandomValue(1 - Math.abs(lambda_gain),
                                   1 + Math.abs(lambda_gain));

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

Listing A.4: 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.5: SensorEventListener Methods

```
def classify(path: str,
  classifier: str = 'knn',
  gridsearch: bool = True
  ) -> KNeighborsClassifier | RandomForestClassifier:
  """
  Loads the data of the given directory and trains a knn classification algorithm,
  then tests the accuracy.

  Args:
    path (str): Folder containing the directories with the CSV files.
    classifier (str): Selected classifier method.
    gridsearch (bool): Should the parameters for the model be optimised via gridsearch
                  or use default ones.

  Returns:
    Classifier: model trained on the dataset.
  """
  x, y = load_data(path)

  x_train, x_test, y_train, y_test = train_test_split(
  x, y, test_size=0.3, random_state=42)

  model = None
  param_grid = None

  if classifier == 'knn':
    param_grid = {
      'n_neighbors': [1, 2, 5],
      'weights': ['uniform', 'distance'],
      'algorithm': ['auto', 'ball_tree', 'kd_tree', 'brute']
    }
    model = KNeighborsClassifier(
      algorithm='auto', n_neighbors=1, weights='uniform')
  elif classifier == 'rf':
    param_grid = {
      'n_estimators': [200, 250, 300],
      'max_depth': [30, 40, 50]
    }
    model = RandomForestClassifier(n_estimators=250, max_depth=40)

  if gridsearch:
    if param_grid is None:
      raise ValueError("Parameters for GridSearch are not set.")

    grid_search = GridSearchCV(model, param_grid, cv=5, scoring='accuracy')
    grid_search.fit(x_train, y_train)
    best_model = grid_search.best_estimator_
  else:
    best_model = model

  if best_model is None:
    raise ValueError("Model is not set.")
```

```
y_pred = best_model.predict(x_test)

print("Accuracy:", accuracy_score(y_test, y_pred))
print("Best Parameters:", best_model.get_params())
print("")
print(classification_report(y_test, y_pred))

return best_model
```

Listing A.6: Classifier Function