Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2024.0429000

Engineering Sensor Spoofing Protection into the Android Operating System

ROY HERSHKOVITZ[1], YOSSI OREN[1], (Senior Member, IEEE)
[1]Stein Faculty of Computer and Information Science, Ben-Gurion University of the Negev, Beersheba 8410501, Israel (e-mails: royhersh | yos@bgu.ac.il)
Corresponding author: Yossi Oren (e-mail: yos@bgu.ac.il).

ABSTRACT Sensor spoofing attacks are a serious threat to mobile phones, as they can manipulate sensor readings to subvert the behavior of applications that rely on these readings. Previous work has shown how machine learning defenses provide effective protection against sensor spoofing attacks without hardware modification. Unfortunately, these defenses require changes to the applications themselves. In this paper, we present SDIOS (Sensor Defense in the Operating System), an approach that engineers sensor spoofing protection into the operating system level, without requiring any modifications to the applications. At its core, SDIOS incorporates an autoencoder based on a Gramian Angular Field (GAF) representation of the sensor readings. We describe the design and implementation of SDIOS, and evaluate its performance and compatibility on a variety of devices. Our results show that SDIOS is able to detect and prevent sensor spoofing attacks in real time, while retaining compatibility with existing applications, but that its performance impact is significant, especially on resource-constrained devices where the machine learning pipeline is run on the central processing unit (CPU).

INDEX TERMS Android, Machine learning, Mobile phones, Operating system security, Sensor protection, Sensor spoofing

I. INTRODUCTION

MODERN smart phones are equipped with a variety of sensors, such as accelerometers, gyroscopes, magnetometers, and GPS. These sensors provide critical data that applications use to enhance user experience, interact with the environment and provide advanced functionality, including augmented reality, fitness tracking, and navigation. The gyroscope and accelerometer found in mobile phones are typically micro-electromechanical systems (MEMS) sensors, a variety of sensors which exhibits low cost, low power and easy hardware integration capabilities. Research has shown that attackers can manipulate sensor data by maliciously interacting with sensor hardware, a technique known as sensor spoofing. For example, Son et al. showed how an attacker can incapacitate drones equipped with MEMS gyroscopes using an external speaker [1], and Tu et al. showed how out-of-band analog signals can affect many types of devices equipped with MEMS inertial sensors, including scooters, robots and health-care devices [2]. Sensor spoofing attacks are particularly problematic in the mobile phone setting, since sensor readings may be used to make critical decisions in security-sensitive applications, such as authentication, location-based services, and health monitoring [3], [4]. In these settings, a successful sensor spoofing attack can lead to unauthorized access, privacy breaches, and even physical harm. This work proposes and evaluates a novel defense mechanism, *Sensor Defense in the Operating System* (SDIOS), that protects apps running on mobile phones from sensor spoofing attacks.

A. DEFENSE STRATEGIES

There are two main strategies to defend against sensor spoofing attacks: hardware-based and software-based.

Hardware-based defenses aim to secure the sensor hardware itself, by adding physical security mechanisms to the sensor hardware, such as protective padding, physical redundancy or tamper detection mechanisms [5]. Some of these defenses, such as the randomized sampling and out-of-phase sampling defenses considered by Trippel et al. [6], are described as software defenses, but are actually applied at the firmware level, at the interface between the sensor hardware and its dedicated microcontroller. Trivially, as long as a hardware-protected sensor has the same interface as a regular sensor, it is indistinguishable from the latter to the operating system and applications, offering immediate protection against sensor spoofing attacks. The main limitation of this approach, however, is that it requires changes to the sensor hardware, or to low-level components of its internal firmware, a modification which may not be feasible for existing devices.

Software-based defenses, most notably the *Sensor Defense in Software* (SDI) defense of Tharayil et al. [7], take a different approach. This family of defenses assumes that the sensor outputs may be corrupted, and aims to detect and mitigate sensor spoofing attacks at the software level. The general approach of these defenses is to train a machine learning model on benign sensor readings, and later use the model to detect anomalies in real time, and optionally use sensor fusion approaches, such as that of Ivanov et al., to replace corrupted sensor readings with simulated values [8].

The software-based defense strategy has a major advantage: it can be applied to existing devices, without requiring any changes to the sensor hardware. It does, however, have a significant challenge: in contrast to hardware-based defenses, which immediately protect the system against sensor spoofing attacks as soon as they are installed, currently proposed software-based defenses require all existing software to be modified to use the new defense mechanism. In addition, since the software defense requires the execution of a machine learning model on every sensor reading, its real-time performance impact may be so significant that the device becomes unusable, especially on resource-constrained devices such as mobile phones.

To address these challenges, we aimed to create a system that answers the following research question:

Can we design a software-based sensor spoofing protection that provides universal protection against sensor spoofing attacks? Can we do so without significantly impacting the device's performance?

B. OUR CONTRIBUTION

In this work, we answer the research question by proposing SDIOS, a novel software-based sensor spoofing protection mechanism that is integrated into a mobile phone operating system. By integrating the defense mechanism into the operating system, we ensure that all applications running on the device are protected against sensor spoofing attacks, without requiring any changes to the applications themselves. After careful selection of the machine learning model and the sensor data preprocessing pipeline, we investigate whether the defense mechanism can have a manageable impact on the device's performance. We discover that the CPU performance impact is beyond the abilities of older, resource-constrained mobile phone models, but that it is suitable for the processing capabilities of modern phones. We provide the software implementation of SDIOS, and evaluate its performance and compatibility on a variety of devices, as well as its ability to detect and prevent real sensor spoofing attacks. While our proof-of-concept implementation focuses on protecting gyroscope sensor readings in the Android operating system, the design of SDIOS is generic, and it can be extended to protect other types of sensors and other operating systems. We make our software implementation, and all of its associated machine learning models, available for public use through an open-source license.

C. DOCUMENT STRUCTURE

The rest of this document is structured as follows. In Section II, we provide background information on MEMS sensors, sensor spoofing attacks, and on sensor support in the Android operating system. In Section III, we describe the threat model, design goals, and methodology of our work. Next, in Section III, we describe the implementation of SDIOS. In Section IV, we evaluate the performance and compatibility of SDIOS, as well as its ability to detect and prevent sensor spoofing attacks. Finally, we conclude in Section V with a discussion of related work, limitations, and future work.

II. BACKGROUND

A. MEMS GYROSCOPE SENSORS

Micro-electro-mechanical systems (MEMS) are a class of electronic micrometric devices. Due to their small size and low power consumption, they are essential in devices where space and storage are limited, such as Internet of Things (IoT) devices, smartphones, drones, etc. The gyroscope is one such sensor. The output of the gyroscope is a vector of readings measuring the device's angular velocity in radians per second, for each one of the device's three axes.

MEMS gyroscopes take advantage of a physical phenomenon called the Coriolis force, an inertial force that appears to act on an object rotating with respect to an fixed reference frame. As described in Son et al., MEMS gyroscopes contain a small vibrating mass. As the device rotates, the Coriolis effect acts on this moving mass, causing it to vibrate with an amplitude directly related to the angular rotation rate [1]. The vibration amplitude is converted to voltage and sampled by an embedded microcontroller, which communicates the reading to the device's main processor through an internal serial bus interface.

B. SENSOR SPOOFING ATTACKS

The vibrating mass in MEMS gyroscopes is sensitive to external vibrations. As shown by Tu et al., an attacker can use an external speaker to generate vibrations at a frequency similar or close to the resonant frequency of the gyroscope's vibrating mass [2]. This causes the gyroscope to enter a state of resonance. Thus, the attacker can cause the gyroscope to output incorrect readings, which can be used to manipulate the device's behavior. In particular, if the rate at which the embedded microcontroller samples the gyroscope's output is known to the attacker, the attacker can use this attack to generate arbitrary readings on the gyroscope's output, subverting the behavior of apps that rely on the gyroscope's output, such as games, augmented reality apps, and fitness tracking apps.

C. SENSOR SUPPORT IN ANDROID

Sensor support in Android is provided through two components: the Sensor Manager API, which runs in the context of the application, and the Sensor Manager Service, which runs in the context of the operating system and communicates with the sensor hardware through the Android Hardware Abstraction Layer (HAL) [9].

An application interested in handling sensor events uses the Sensor Manager API to register a *Sensor Event Listener* to a certain sensor [10]. The Sensor Manager API then communicates this request to the phone's Sensor Manager Service using Android's Binder Inter-Process Communication (IPC) mechanism. Once an Event Listener is registered, it will be called every time the sensor reading changes, and will be provided with a *Sensor Event* object containing the sensor's readings.

To interact with the low-level sensor hardware, the Sensor Manager communicates with the Hardware Abstraction Layer (HAL). The HAL directly controls and communicates with the dedicated sensor driver, which can boot up the sensor, configure its operating parameters, and poll the sensor chip for real-time readings.

III. METHODOLOGY

This section describes the threat model, design goals, and methodology of SDIOS.

A. THREAT MODEL

Our work focuses on protecting the gyroscope sensor of Android devices from denial-of-service (DoS) attacks. Our threat model assumes that the attacker was able to study the target device in an offline phase, and to determine the resonant frequency of its gyroscope sensor. Next, in the online phase, the attacker uses an external speaker to generate vibrations at this resonant frequency, causing the sensor to output corrupted readings.

Figure 1 shows an example of the gyroscope sensor outputs of a OnePlus 3T phone, comparing the readings when the phone is at rest, and when it is experiencing a DoS attack using the attack setup described by Farshteindiker et al. in [11]. As the Figure shows, the gyroscope readings under the DoS attack are significantly different from the readings at rest. It can also be observed that the resonance highly affects the gyroscope's x and y axes, while the z axis has a relatively lower amplitude; this may be due to the hardware structural implementation of this gyroscope.

In this work we assume that the sensor readings are corrupted by the attacker, but that the attacker does not have the ability to directly control the sensor readings. While we do not explore this direction in our current work, it is also possible to protect the gyroscope sensor from attacks which completely overwrite its output with attacker-controller readings. This can be done by applying sensor fusion to compare the gyroscope readings with other sensors, such as the accelerometer or magnetometer [7].

B. DESIGN GOALS

The primary *functional* goal of SDIOS is to provide an effective defense against sensor spoofing attacks. To achieve this goal, SDIOS must be able to detect sensor spoofing attacks in real time, and to prevent the corrupted sensor readings from reaching the applications. A secondary functional goal is to provide a *universal* defense — that is, is should protect

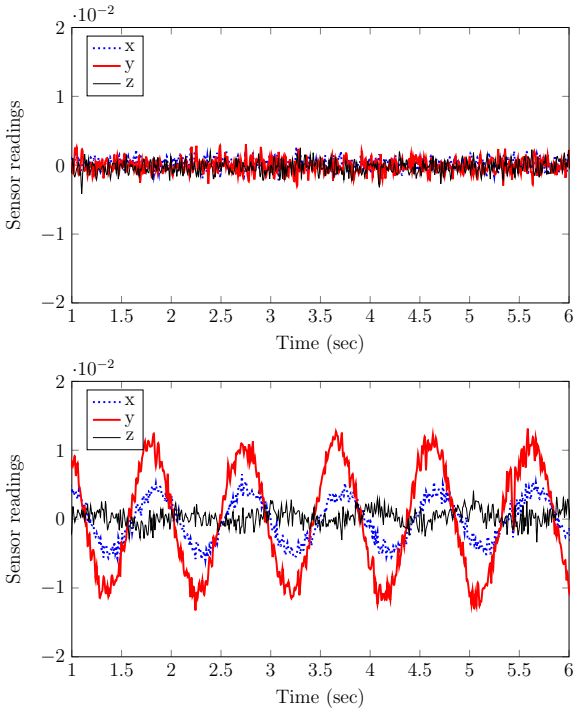

FIGURE 1. Comparison between a device's gyroscope readings at rest (top) and under a DoS attack (bottom)

all applications running on the device, without requiring any changes to the applications themselves.

Our *non-functional* goals are to minimize the performance impact of the defense mechanism, both in terms of latency and in terms of effect on battery life, and to ensure that the defense mechanism is compatible with a wide range of devices.

C. TRUST AND OS INTEGRATION

Different applications have different policies for dealing with attacks on the sensors they use. For example, a game application may choose to simply ignore the gyroscope readings when the readings cannot be trusted, while a drone flight-control application may choose to react more dramatically and safely land the drone when it detects an attack. To allow this flexibility, SDIOS assigns each sensor reading a *trust value*, indicating the likelihood that the reading is corrupted. This trust value is an output of the machine learning model applied by SDIOS to the sensor readings. To provide the full benefits of SDIOS, applications must be able to retrieve this *trust value* and use it to make decisions. This, however, requires changes to the application code, which may not be feasible for all applications.

To address this challenge, we propose multiple ways of integrating SDIOS: In *full support* mode, SDIOS is integrated into the Android operating system, and applications are modified to access sensor readings through a new API provided by SDIOS. This modified API provides applications with the trust values assigned to the sensor readings, in addition to the sensor readings themselves. This allows applications to

make their own application-specific decisions on how to treat potentially-corrupted sensor readings.

The second mode we propose is an *OS-only* mode, in which SDIOS is integrated into the Android operating system, but applications remain unmodified and continue accessing sensor readings through the standard Android Sensor Manager API. In this mode, SDIOS intercepts sensor readings with low trust values, and prevents them from ever reaching the application. This mode allows SDIOS to provide protection without requiring any changes to the applications themselves.

The final integration mode is *app-only*. This mode assumes that the operating system cannot be modified. In this mode, SDIOS can be bundled with individual applications, providing only these applications with access to the trust values assigned to the sensor readings, while all other applications continue to receive unprotected sensor readings from the operating system. Since this mode of integration was already considered in other works, we focus our discussion on the first two modes [7].

D. MACHINE LEARNING PIPELINE

A central component of SDIOS is the machine learning model, which processes all sensor readings created by the sensor hardware, before they are passed to the applications. We trained our model on a large dataset of benign sensor readings created through a multi-user experiment, as described in Section IV. The design we chose for the machine learning model is an autoencoder based on a Gramian Angular Field (GAF) representation of the sensor readings, inspired by the work of Wang et al. [12], as described below. A comparative study of alternative machine learning architectures is left for future work.

1) Autoencoders

Autoencoders are neural networks that learn compact and meaningful representations of data by reconstructing inputs with minimal loss. They consist of two parts: an encoder that compresses the input into a lower-dimensional representation, and a decoder that reconstructs the original input from this encoding. By minimizing reconstruction error, autoencoders capture the underlying structure of the dataset. A *model loss* function measures the difference between the input and the reconstructed data. As shown by Sakurada et al. and An et al., autoencoders are well-suited for the task of anomaly detection, since an autoencoder trained on benign data will perform poorly when encoding and decoding non-benign data [13], [14]. Essentially, a well-trained autoencoder will have a low loss value for benign data, and a high loss value for non-benign data. In our implementation of SDIOS, we set a threshold for the maximal loss value included in the benign class, turning the autoencoder into a binary classifier that can detect anomalous data. While we did not explore this approach in the current work, the loss value can also be transformed into a continuous value between 0 and 1, allowing apps to make more subtle decisions based on the relative trust value of a particular sensor reading.

2) Gramian Angular Fields (GAF)

Time series data is a collection of sequential data entries, with each entry associated with a specific timestamp. This unique characteristic allows for the analysis of trends and patterns over time. The trace of sensor readings used in our work is an example of such a time series. As shown by Wang et al., it is beneficial to convert such time series data into images, which can then be analyzed using advanced machine learning methods such as convolutional neural networks [12]. The Gramian Angular Field (GAF), in particular the Gramian Angular Summation Field (GASF) is one such preprocessing method. It applies a mathematical transformation to the time series data, converting it into a square matrix, using the following formula:

$$X = Minscale(Input, [-1, 1])$$

$$X' = \sqrt{1 - X^2}$$

$$GASF = outer(X, X) - outer(X', X')$$

Where *outer* is the standard two-vector outer product.

E. EVALUATION METRICS

We apply a series of functional and non-functional metrics to evaluate the performance of SDIOS. The main functional metric is the *sensor spoofing detection performance*, which measures the ability of SDIOS to detect sensor spoofing attacks while avoiding false alarms on benign sensor readings. We report both on the *accuracy* and *recall* of SDIOS, as well as the combined *F1 score*.

To make sure SDIOS is practical for real-world use, we also evaluate several non-functional metrics. The first is the *real-time impact* of SDIOS, which measures the latency and jitter added by the SDIOS machine learning pipeline to the sensor readings. The second is the *battery impact* of SDIOS, which measures the additional power consumption caused by the SDIOS machine learning pipeline. Finally, we turn to the *compatibility* of SDIOS. We measure both *device compatibility*, which measures the ability of SDIOS to run on a variety of devices, as well as *application compatibility*, which measures how easy it is to integrate SDIOS into existing applications.

IV. EVALUATION

A. DATA COLLECTION AND MACHINE LEARNING MODEL CREATION

To provide the machine learning model with training data, we conducted two experiments: one to collect benign sensor readings, and one to collect sensor readings under a sensor spoofing attack.

Benign data was collected with the help of a group of undergraduate students, using an application written using the Expo framework [15]. The application asks the user to perform seven different one-minute activities, and collects data from the accelerometer, gyroscope, and magnetometer while these activities are performed. The collected sensor data is later uploaded to a DynamoDB database for further

processing. The chosen activities included placing the phone at rest on a table and in the user's pocket, shaking the phone, typing a short story on the smartphone keyboard, walking, climbing stairs, and finally playing a game in which the player navigates an obstacle course by tilting the phone. Application screenshots can be seen in fig. 2. Prior to collecting data with the application, participants were asked to sign a human-subject agreement form approved by our department's Ethics Review Board. Participants were compensated for their time with a small grade bonus in an undergraduate course.

In total, we collected data from 82 different participants. After preprocessing, our data set contained 41599 GAF samples per axis for the accelerometer, 39315 GAF samples for the gyroscope, and 65168 GAF samples for the magnetometer, using phones made by Apple, Samsung, Xiaomi, Oneplus and others. As we note below in section IV-C, the performance of the machine learning model is significantly affected by the size and diversity of the training data.

To collect *sensor spoofing data*, we recreated the DoS attack of Tu et al. [2], which serves as a representative benchmark for attacks which perform signal manipulation. As shown in Figure 3, our attack setup consists of a PUI audio APS2509S-T-R piezoelectric transducer directly attached to the back of the smartphone under attack, controlled by the analog output of the arbitrary wave generator (AWG) function of a PicoScope 2000 series oscilloscope, which is later amplified by a Lepy LP-2051 Hi-Fi Stereo Power Amplifier. A custom-designed native application was used to find the resonant frequency of the MEMS gyroscope; next, we collected sensor readings while the phone was under attack. We repeated this experiment for phones made by LG, Oneplus and Xiaomi. In all cases, the phones were at rest during the attack. We discovered during our testing that the Xiaomi device we tested did not have a physical gyroscope sensor. Instead, it approximated the gyroscope readings using the accelerometer and magnetometer sensors [16]. Thus, we did not include the Xiaomi device in our evaluation of SDIOS.

The machine learning model was trained using TensorFlow and Keras on an NVidia GPU cluster. After training, the model was converted into a TensorFlow Lite model, which is optimized for running on mobile devices.

B. IMPLEMENTING OS INTEGRATION

The Android Open Source Project (AOSP), which is the open-source version of the Android operating system, can be only be compiled and deployed on a small subset of devices. LineageOS is an up-to-date open-source fork of AOSP which offers support for devices from a more diverse set of manufacturers [17]. Since we wanted to evaluate SDIOS on as many devices as possible, we chose to integrate SDIOS into LineageOS. We chose versions 18.1 and 20.0 of LineageOS, which are based on Android versions 11 and 13, respectively. Since the stock operating system provided with phones may contain different configurations and optimizations than the ones found in LineageOS, we also produced an unmodified version of LineageOS for performance comparisons.

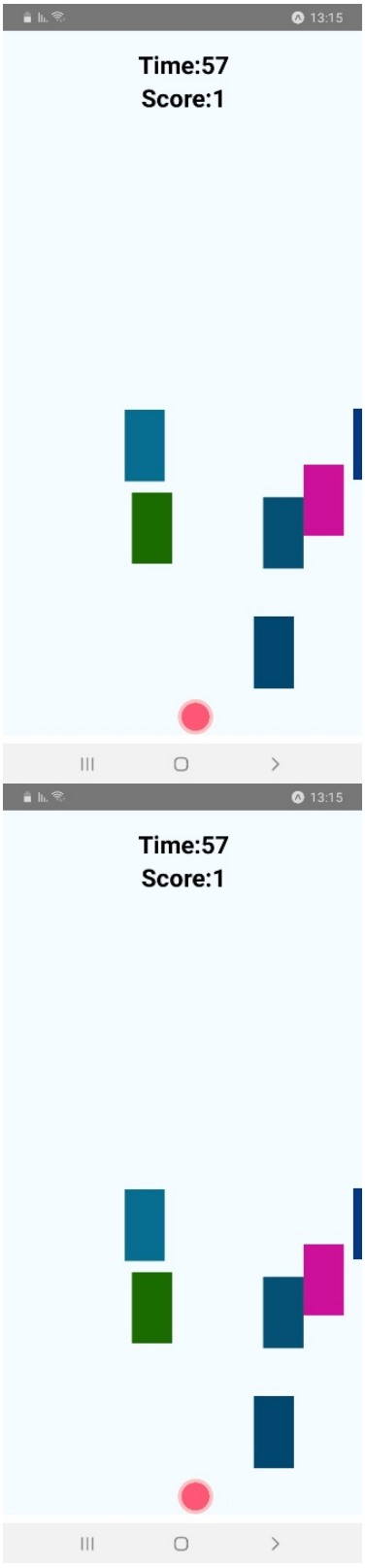

FIGURE 2. Screenshots of the SDI data collection application

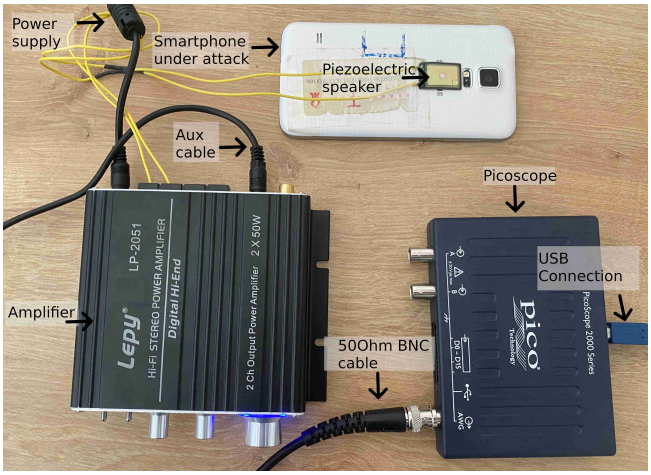

FIGURE 3. Data collection setup for anomalous data

	Accuracy	Recall	F1
Anomaly (TP+FN)	0.84	-	-
Benign (FP+TN)	0.98	-	-
Total	0.91	0.84	0.90

TABLE 1. Performance of four GAF autoencoders x, y, z axes, and l2norm with encoding size of 300

Device	Accuracy	Precision	Recall	F1
OnePlus 3T 1	0.88	0.75	0.75	0.75
OnePlus 3T 2	0.61	0.28	0.47	0.35
LG G2	0.42	0.17	0.43	0.25
Oneplus 7 Pro 1	0.64	0.34	0.67	0.45
Oneplus 7 Pro 2	0.79	0.52	0.75	0.62
Average	0.66	0.41	0.61	0.48

TABLE 2. Detection of a DOS attack on the MEMS gyroscope in different SDIOS protected devices

There were two main possible implementation strategies for SDIOS: adding it directly to the kernel as a native-code driver, or adding it as a system service as a Java application. While the native-code driver approach may have provided better performance, we chose the system service approach since the user-mode Android runtime provides better support for our machine learning pipeline, since using high-level code offers better cross-device compatibility across devices with different hardware abstraction layers (HALs), and since this allows the service application to be easily updated through application stores without requiring a firmware update.

To implement this approach, we first created an SDIOS Service Application, as illustrated in Figure 4. The Service Application is launched when an application accesses the Android Sensor Manager API. It hosts the machine learning pipeline, and exposes an API which is functionally equivalent to the standard Android Sensor Manager API, but which also has the ability to block sensor readings with low trust values. Since Android applications request sensor readings through a listener API, a blocked sensor reading will simply not trigger the listener, and thus will not reach the application, minimizing the impact on app stability. To ensure the sensor inputs of all apps running on the phone are intercepted by the SDIOS Service Application, we modified the Android Application Framework to return a pointer to the SDIOS Service Application's API whenever an application requests the Sensor Manager API using the `getSystemService("sensors")` function call. The original Android Sensor Manager can still be accessed by calling `getSystemService("sensors_raw")`. This allows applications that require raw sensor readings to continue functioning as before, and also provides access to raw sensor readings for the SDIOS Service Application itself.

An extended API, which also provides trust values, is also available to applications that were modified to request it. This API is provided through a library which implements extended versions of `SensorEvent` and `SensorEventListener`. The library is designed to transparently fall back to the standard Android Sensor Manager API if the SDIOS Service Application is not detected, allowing the same application to be deployed both to devices with and without SDIOS.

C. EVALUATING SENSOR SPOOFING DETECTION

To evaluate the sensor spoofing detection performance of SDIOS, we used the benign and anomalous sensor readings collected in the experiments described in Section IV-A.

Figure 5 show the GAF autoencoder reconstructions of a benign sample and an anomalous sample. Since we trained an autoencoder for each sensor axis, the figure has four GAF matrixes, each with dimensions of $120x120$. As the Figure shows, the autoencoder is able to reconstruct the benign samples with a low reconstruction error, while the reconstruction of the anomalous samples is considerably different from the original GAF matrix. We measured the mean benign loss value to be approximately 0.02 and the mean anomalous loss value to be around 0.07, and empirically set a threshold of 0.027 for anomaly detection. Any measurements with a loss value above this threshold are considered anomalous, and are assigned a trust value of 0.

Evaluating this threshold on the collected data, we found that the autoencoder was able to detect 84% of sensor spoofing attacks, with a false positive rate of 2%.

We then evaluated the performance of the autoencoder in a real-time attack setting. The results for this setting, as shown in Table 2, are significantly worse than the offline evaluation. We propose some possible reasons for this discrepancy in Section V.

D. EVALUATING COMPATIBILITY AND PERFORMANCE

We evaluated the compatibility of SDIOS in two settings: first, we examined popular apps which use the sensor API, and made sure they ran correctly without modifications on a system with SDIOS installed. Next, we examined several open-source apps which use the sensor API, and measured the

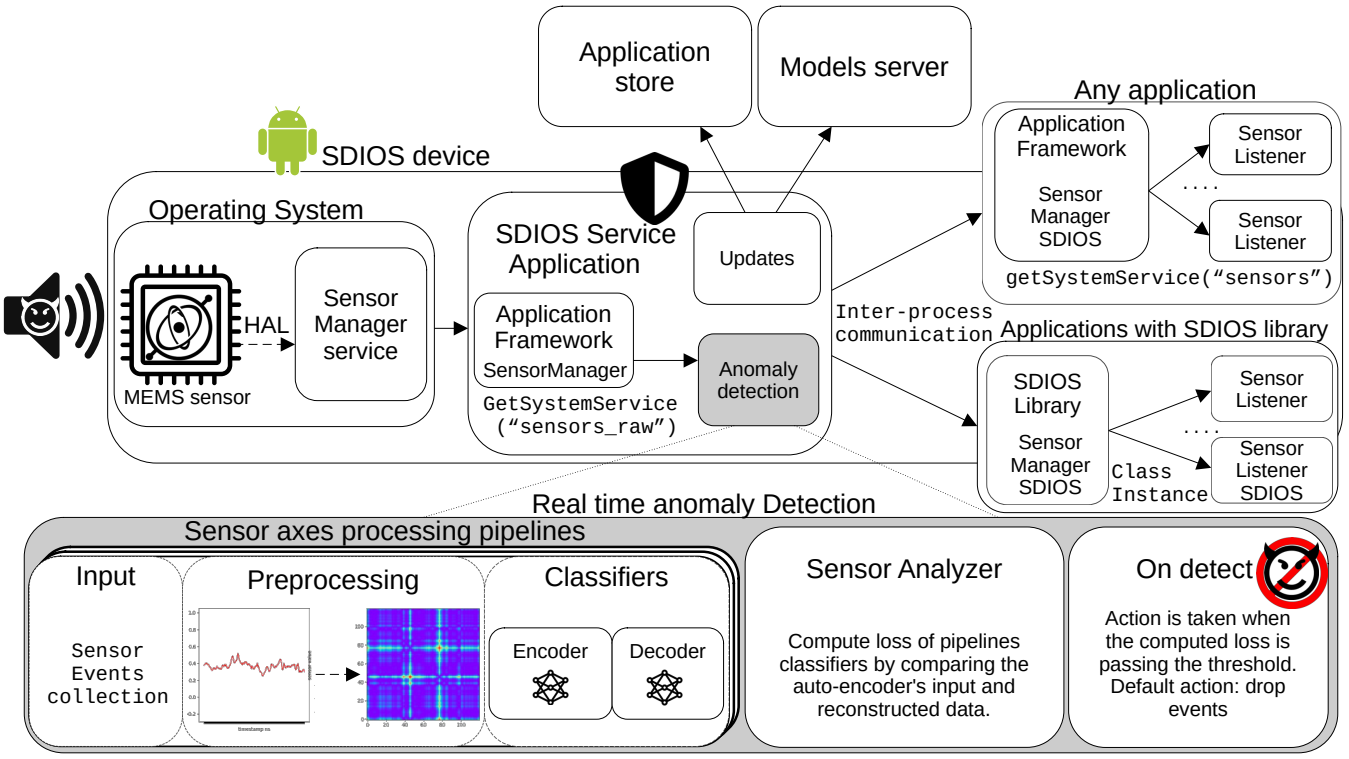

FIGURE 4. SDIOS high-level overview

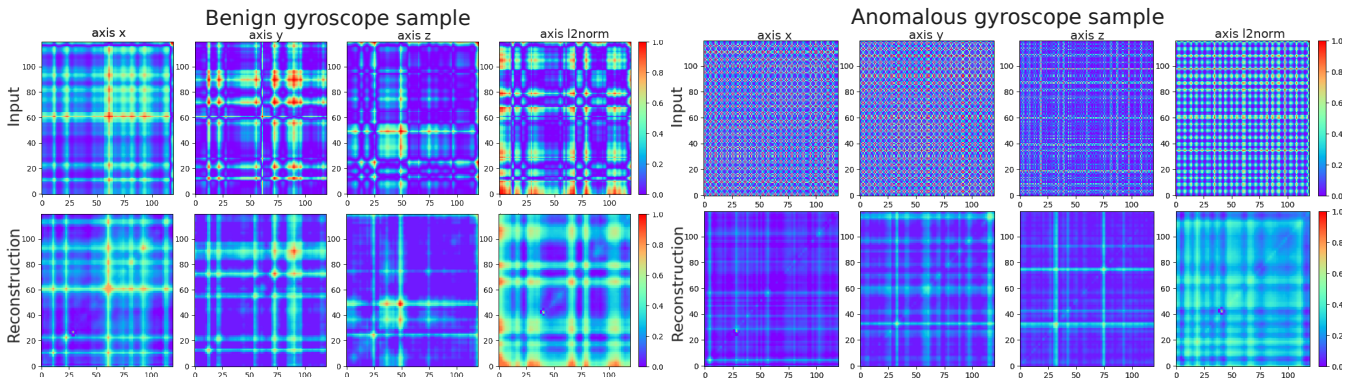

FIGURE 5. The GAF autoencoder reconstruction of anomalous and benign samples

effort involved in modifying them to use the extended SDIOS API which provides trust values.

The apps we tested are listed in Table 3. We note that, since LineageOS does not include support for the Google Play Store, we had to manually install the apps using the APK files provided by the developers. In general, we found that the apps ran correctly on the system with SDIOS installed, with two notable exceptions. First was the ARLOOPA app, which failed to load altogether. We noted that the error logs created by the application mentioned Google Play Services, which suggests that this problem is due to the use of LineageOS, and not due to SDIOS. We confirmed this by observing that the application also failed to run on a stock version of LineageOS without SDIOS modifications.

The second app which showed incorrect behavior was the Compass app. In this app, we observed that the application shows stale values when the user goes to the home screen, and then back to the application. This behavior was not observed in the unmodified LineageOS, and we suspect that it is due to an unexpected interaction between the sensor manager and the Android OS suspend logic.

Our second compatibility experiment measured the effort involved in modifying open-source apps to use the extended SDIOS API. The list of apps we tested, together with the effort required to modify each one, is presented in Table 4. As the table shows, we converted most applications with a minimal amount of effort, making only small changes to the code. The only exception was the Trail Sense application, which uses a

Application	Sensors	Review score	Amount of reviews	Compatible with **SDIOS**
ARLOOPA: AR Camera 3D Scanner	(unknown - failed to load)	4.7	33.9k	—
BBall — Marble Labyrinth	Accelerometer	3.7	359	✓
Compass (Altimeter, Sunrise, Sunset)	Gyroscope and magnetometer	4.8	106k	—
Digital Compass	Accelerometer, gyroscope and magnetometer	4.6	290k	✓
GyroDroid	Accelerometer and magnetometer	-	-	✓
Rally Fury — Extreme Racing	Gyroscope and magnetometer	4.1	821k	✓
Sensor Data	Accelerometer, gyroscope and magnetometer	-	-	✓
Sensors Multitool	Accelerometer, gyroscope and magnetometer	4.4	10.6k	✓
Step Counter — Pedometer, MStep	Accelerometer, gyroscope and step counter	4.9	1.32M	✓
Sygic GPS Navigation & Maps	Magnetometer	4.6	1.86M	✓

TABLE 3. Unmodified applications used for compatibility testing (download statistics as of September 2023).

Application	Sensors	Github stars	Github forks	Changed lines of code	No. of files changed
Balance IT — Space	Gravity	1	0	5	4
Compass	Accelerometer and magnetometer	236	100	9	3
DoorSignView	Rotation vector	563	84	15	12
Heaven	Accelerometer	6.5k	748	31	7
Pedometer	Stepcounter	1.4k	679	13	4
Sensey	Accelerometer	2.7k	279	44	22
Sensors	Accelerometer and magnetometer	5	0	8	3
Sensors Data Logger	Accelerometer, gyroscope and magnetometer	45	16	7	3
Trail Sense	Accelerometer, gyroscope and magnetometer	704	46	-	-
Tux Racer	Accelerometer	32	15	7	3
React Native Sensors integration	Accelerometer, gyroscope and magnetometer	856	226	9	3

TABLE 4. Open-source applications used for compatibility testing (download statistics as of September 2023).

third-party library named Andromeda to interact with the sensors, instead of directly querying SensorManager [18]. While it is possible, in theory, to modify the Andromeda library to use the SDIOS API, the effort involved in comprehensively testing the compatibility of this modified version with all downstream applications which use Andromeda is beyond the scope of our work.

Figure 6 shows, as an example, the changes required to integrate the SDIOS API into the Balance IT — Space application. As the Figure shows, the changes required to integrate the SDIOS API are minimal, and consist of adding a few extra import statements and changing the sensor manager initialization code.

We would like to highlight one special open-source project we investigated, namely the React Native Sensors library. React is a JavaScript website development framework, and ReactNative is a cross-platform smartphone app development framework based on React. Within this framework, the React Native Sensors library provides React apps with access to the gyroscope, accelerometer, and magnetometer sensors. We were able to modify the React Native Sensors library to use the SDIOS API with minimal effort, potentially allowing all React Native applications to benefit from the enhanced sensor trust API provided by SDIOS. As mentioned before, we did not comprehensively test downstream React Native apps which use this library, a step which is left for future work.

Our final evaluation experiment looked at the real-time performance impact of SDIOS. We measured the performance impact on two devices. The first was a relatively low-end device, the OnePlus 3T, manufactured in 2016 and running LineageOS 18.1, and the second device was a high-end Xiaomi Redmi 9, manufactured in 2020 and running LineageOS 20.0. As noted previously, the Xiaomi Redmi 9 does not

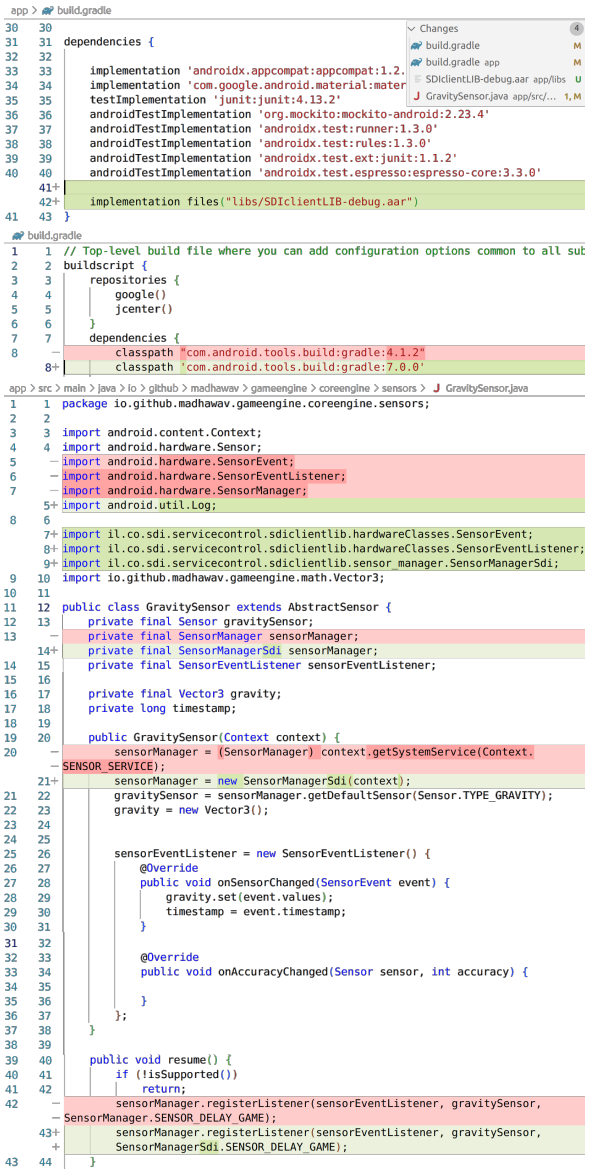

FIGURE 6. integrating SDIOS into the *Balance IT* app

have a physical gyroscope sensor. As such, we only evaluated the performance impact of SDIOS on this device, without evaluating detection accuracy. On both phones, we executed a simple custom application which continuously queried the accelerometer, gyroscope, and magnetometer, and measured the performance of the system while running the application using the Android Studio energy profiler. We measured three scenarios: in the baseline scenario, the application was run on a stock version of LineageOS, without any modification. The second scenario evaluated the *app-only* deployment mode, as proposed by Tharayil et al. [7]. In this scenario, the application was modified to use an app-only version of the SDI library, but the operating system was not modified. In the third scenario, we evaluated the *OS-only* deployment mode – the operating system was modified to include the SDIOS

Service Application, but the application itself was not modified. We used LineageOS for baseline calculations instead of the vendor-provided operating system image provided with the device, since the vendor-provided operating system may contain closed-source components and other optimizations which we were not able to reproduce in SDIOS.

The results of this experiment are summarized in Table 5. As the Table shows, enabling SDIOS, both in app-specific library form and at the operating system level, caused total CPU usage and RAM consumption to increase considerably. This increase in CPU and RAM consumption may be explained by the fact that the machine learning pipeline is running on the CPU, and is not optimized for the specific hardware of the device, as we discuss further in Section V. As the Table also shows, the resource consumption of the OS-only mode, with the unmodified app, was larger than that of the app-only deployment mode. This is expected, since all applications and services running on the the phone in OS-only mode receive SDIOS processing, while only the single modified application is processed by the API in app-only mode. This overhead is non-negligible, especially on lower-end devices, suggesting that app-only deployment may be a better fit in a resource-constrained setting. Interestingly, we observed that, while the overall CPU usage of the system increased due to the introduction of SDIOS, the CPU usage of the test application itself slightly *decreased* when SDIOS was enabled. This is probably due to the fact that SDIOS was filtering out some sensor events which were previously being sent to the application.

We also measured the added latency introduced to sensor event handling by SDIOS, and discovered that our implementation added an average of 7ms of latency to the sensor readings on the OnePlus 3T, and 3ms of latency on the Xiaomi Redmi 9. While these are relatively low average values, we noted that there was a significant amount of jitter in the latency, with the latency sometimes spiking to 20ms or more on the lower-end OnePlus 3T.

Finally, we measured the impact of SDIOS on battery life using the Android Battery Historian tool. We discovered that enabling SDIOS had no significant effect on the power consumption of apps which did not use sensors (for example, the YouTube player), but that it caused the power consumption of sensor-intensive applications to increase by a factor of up to 3. This effect was especially pronounced on the OnePlus 3T, where the power consumption of the basic sensor API is already high. This increased power consumption will likely have a significant impact on the battery life of the device, and is a direct effect of the high CPU consumption of our CPU-bound machine learning model, as we discuss further in Section V.

V. DISCUSSION

The results presented in the previous sections show that integrating advanced machine-learning defenses into the Android operating system is a practical and effective way to protect all applications from sensor spoofing attacks. Our proposed solution, SDIOS, provides a universal defense against sensor spoofing attacks, and is compatible with a wide range of devices and applications. Applications that require access to the trust values assigned to the sensor readings can easily be modified to use the extended SDIOS API, and the modifications required to do so are minimal. Despite its advantages, the currently implemented version of SDIOS has several limitations, which we discuss below.

A. LIMITATIONS

a: Machine Learning Model Performance

As shown in Section IV-C, our machine-learning model was less effective in detecting sensor spoofing attacks in real-time than it was in the offline evaluation stage. We believe this reduced accuracy can be attributed to the relatively low amount of traces produced in our crowd-source data collection experiment, both in terms of the number of traces collected, and in terms of the physical activity modes represented in the benign data set. We consider that this is not a fundamental limitation of our approach, a claim which is supported by existing works that achieve higher accuracies when machine learning is used to detect sensor spoofing, as described in Section V-B.

b: CPU Use, Power Consumption, and Latency

The high CPU consumption of the machine learning model has a significant impact on the performance and battery life of the device, especially on low-end devices. The latency introduced by SDIOS, while low on average, has a significant amount of jitter, which may be problematic for applications that require precise timing. The main contributing factor to this high CPU consumption is the fact that the machine learning model is running on the CPU, and is not optimized for the specific hardware of the device. Modern smartphones have on-board graphics processing units (GPU), and some even have machine learning accelerators, commonly referred to as Neural Processing Units (NPUs). Moving the machine learning stack from the CPU to the GPU or NPU would likely significantly reduce the CPU consumption of the machine learning model, and would also likely reduce the power consumption of the device and reduce the jitter in the latency of sensor readings. Moving from CPU to GPU/NPU is a natural next step for making this research more practical.

B. RELATED WORK

This work is related to three independent strands of research: attacks on position sensors, detecting anomalies in data streams, and engineering security features into Android.

1) Attacks on Position Sensors

MEMS sensors have an important role mobile devices, and it is essential to keep them safe. Despite this, several works have shown how malicious entities can exploit MEMS sensors to compromise user privacy, harm electronic infrastructure, or risk human lives.

Tu et al. demonstrated three attacks on MEMS gyroscopes and accelerometers: DOS, side-swing, and switching-

	OnePlus 3T		Xiaomi Redmi 9	
	CPU (%)	RAM (MB)	CPU (%)	RAM (MB)
Stock LineageOS, unmodified app	30	628	17	686
Stock LineageOS, modified app	60	655	53	704
SDIOS, unmodified app	80	840	74	680

TABLE 5. CPU and RAM usage comparison across configurations

attack [2] . They also explained the mathematical theory behind each method and demonstrated the attacks on various devices. They discussed countermeasures such as isolation foam, random sampling, and hardware filtering requiring hardware or firmware updates. Our work uses the attacks proposed by Tu et al. to evaluate our defense mechanism. Contrary to the mitigations suggested Tu et al., our defense method is software-based, and requires only an operating system update to deploy to existing devices.

Farshteindiker et al. used a mobile phone gyroscope to transfer data at a rate of thousands of bits per second using a piezoelectric speaker placed close to the phone [11]. Their work shows how sensor spoofing can be used to exfiltrate sensitive data over an unmonitored channel. We believe that a device equipped with SDIOS would be protected from this attack, as the attack would cause the sensor readings to deviate from their expected values, resulting in a low trust value and potentially blocking the sensor readings from being sent to the exfiltrating application.

Son et al. examined 15 kinds of MEMS gyroscopes for their resonant frequencies, and found that many of them can be caused to output invalid sensor readings after receiving a jamming signal sent using a consumer-grade speaker [1]. They have also shown that drones with vulnerable gyroscopes can lose control and crash under such an acoustic attack. Similarly, Gao et al. presented a way to control drones' self-navigation via transduction attacks on their inertial sensors [19]. This fundamental research highlights the risks of sensor spoofing attacks, and demonstrates why additional research should be done to defend vulnerable sensors from these attacks.

Trippel et al. have shown that accelerometer values can also be spoofed using acoustic attacks [6]. They also suggested random sampling of the sensor, as well as other hardware implementations that can reduce the attack's effectiveness. We note that while random sampling is nominally a software-based defense, it effectively requires the low-level code of the microcontroller interfacing with the hardware to be modified. This is a significant barrier to deployment on existing devices. This contrasts with our approach, which consists entirely of high-level code running on the phone's main CPU.

Soobramaney et al. showed that surrounding MEMS gyroscopes with 12 mm of microfibrous cloth reached a 65% reduction in noise signal amplitude, and that up to 90% reduction could be achieved with additional padding [5]. While this defense is very instructive toward the design of future inertial sensors, it can not be applied to existing devices.

A significant body of research showed that sensors can be exploited to harm user privacy. Hupperich et al. showed that each sensor in a device has a unique fingerprint generated from its output, and that a user's device can therefore be identified in many different applications only by its sensor output [20]. These results suggest that the machine-learning anomaly detection model should not be trained on a single device, but on a diverse set of devices, as we did in our research. He et al. showed how trained models can predict lifestyle activities using data from an accelerometer [21]. Similarly, spying applications can use sensors' output to predict what activity a user is doing while holding his phone. Finally, Michalevsky et al. showed how to eavesdrop on a user with a gyroscope; they used signal processing and machine learning and showed that gyroscope readings are sufficient to identify a speaker's information and even parse speech [22]. Position and motion sensors are generally available to apps, and even to websites, with minimum permissions. Das et al. created a crawler that finds online hosted scripts that access smartphone sensors [23]. Their results show that many sites use sensor data to enhance user experience and to perform tracking and analytics.

On the defensive side, Chen et al. proposed TBAuth, a continuous authentication framework for smart phones based on tap behavior, which correlates the vibrations measured by positional sensors with user identity information embedded in tap behavior to enhance user security [24]. While these works focus on the output of sensors (privacy), our work focuses on the input to sensors (integrity), though both involve the OS sensor stack.

2) Detecting Anomalies in Data Streams

Denning published one of the first papers to present a mechanism capable of anomaly detection in a data stream. The paper showcases a theoretical model that is capable of detecting intrusions based on upcoming system events, and acting upon detection [25]. Ferdousi et al. discussed finding anomalies in time series financial data using an unsupervised detection technique called PGA (Peer Group Analysis), aiming to detect financial fraud [26]. They showed that their tool could detect brokers who suddenly start selling the stock differently from other brokers to whom they were previously similar, which can indicate financial fraud.

Tharayil et al. showed how a software implementation can be used to protect against acoustic sensor spoofing attacks [7] . Their implementation detects attacks by observing statistical features of position sensor outputs, and uses classical classification machine learning models such as KNN, SVN, ensemble, and a binary decision-tree. The paper also proposed

detection by sensor fusion – a mathematical transformation that reveals if two sensors agree on the change in inertia. We follow up on the work of Tharayil et al. in this paper, by presenting an mechanism that integrates into the operating system, and is thus capable of protecting any application running on the device from sensor spoofing attacks.

Sheetrit et al. [27] propose a new dynamic-behavior-based model, named Temporal Probabilistic proFile (TPF), for classification and prediction tasks of multivariate time series. TPF was shown to detect sepsis health conditions more accurately than alternative machine learning practices. The algorithm abstracts raw, time-stamped data into a series of higher-level, meaningful concepts, which hold over intervals characterizing periods. Then, the authors use it to discover frequently repeating temporal patterns within the data. Using the identified patterns, they create a probabilistic distribution of the overall entity population's temporal patterns, each target class, and each entity. The work of Sheetrit et al. demonstrates the advantage of processing raw time series data, in order to create a more accurate anomaly detection system. Time series modelling was also used by Novitski et al. to predict mortality from chronic kidney disease [28]. In the network security domain, Mirsky et al. presented a network intrusion detection system based on an ensemble of autoencoders, which can differentiate between normal and abnormal traffic patterns collectively [29].

Gramian Angular Fields have been used in other works for anomaly detection and fault analysis. Recently, Li et al. [30] used a combination of GAF and the ResNet architecture to perform fault diagnosis of drilling pump fluid ends.

3) Engineering Security Features into Android

Xu et al. published a tool called Aurasium that can repackage Android applications to make their execution more secure [31]. Aurasium automatically repackages arbitrary applications to attach user-level sandboxing and policy enforcement code, which closely watches the application's behavior for security and privacy violations, such as attempts to retrieve a user's sensitive information. Also can also track SMS and IP requests and can detect and prevent cases of privilege escalation attacks.

Pham et al. observed that many applications search for other installed applications using an Android API which enumerates every existing package on the device [32]. Then, with this listing, these apps can steal data from sensitive applications and harm user privacy. Their mitigation, HideMyApp, manages installing and launching applications that users want to keep private, such as health applications, making these sensitive applications invisible to other applications installed on the device.

Roesner and Kohno showed how to secure user information in Android from embedded interfaces, for example, by giving fixed GPS value for location queries [33]. This was achieved by modifying the Android application framework. The authors evaluated their work by the number of code lines that need to be added to an application to make it secure, an evaluation approach that we also use.

Zhang et al. studied the data storage process in Android system services, and discovered design flaws that can lead to serious denial of service attacks [34]. They also proposed a fuzzing-based approach, applied it on three Android systems with the latest security updates, and identified many unique vulnerabilities affecting hundreds of interfaces across many system services.

Sikder et al. [35] considered sensor security in Android under a different threat model than ours – instead of focusing on sensor spoofing attacks, they focused on the security of the sensor data itself. Specifically, the authors considered malicious apps that are triggered via sensors, that leak data to other devices through the sensors, and that try to steal user data using sensors. The authors presented a machine-learning framework called 6thSense which can detect this specific class of sensor attacks, and showed that it is practical and has minimal overhead.

Machine learning approaches have widely been used to detect Android malware. Recently, Ma et al. presented CADroid, a cross-combination attention-based framework for Android malware detection, which utilizes a novel attention mechanism to improve detection performance while maintaining efficiency [36].

Outside the Android problem domain, O'Neill et al. presented how to create a simple-to-use API which prevents developers from using misconfigured and insecure TLS settings [37]. Their objective was to minimize the number of code lines that need to be changed to make the code work with the new library. We also use these metrics to test how many code lines an application needs to change to be compatible with the SDI library and how many lines of code an application needs to add to get the confidence value of the sensor event.

C. FUTURE WORK AND OPEN QUESTIONS

The most immediate next step for this research is to *move the machine learning inference task from the CPU to the GPU/NPU*. We believe that this move will likely significantly reduce the CPU and power consumption of the machine learning model, as well as reduce the jitter in the latency of sensor readings. It is also natural to apply our defense to *additional sensors*, starting with the other motion sensors (the accelerometer and the magnetometer), and then moving on to other sensors such as the GPS, the proximity sensor, and the microphone.

Our system is designed to accept any machine learning model formatted as a TensorFlow Lite model. As such, it is also a natural next step to *experiment with different machine learning models* to see if they can provide better performance than the autoencoder model we used in this research. Our trace processing used a Gramian Angular Field transform to convert the sensor readings into a format suitable for the autoencoder. It would be interesting to see if other transforms, such as the Wavelet Transform, can provide better

performance, or whether the raw sensor readings can be used directly with an appropriate time-series model. It would also be interesting to see how much *training can be made generic* — it is possible to train one model on a diverse set of devices, and then deploy it on any device, or is it necessary for the model to be trained for each device separately?

A final possible direction for future work is to consider *how applications should respond to readings with low trust values*, and especially how should this condition be relayed to the user: a false positive situation where some sensor reading values are ignored is tolerable; a situation where the used is needlessly bombarded with spurious warnings is not. In particular, our current implementation reports a binary trust value to applications; it would be interesting to see if a more fine-grained trust value can be used by apps to make better decisions about how to respond to low-trust sensor readings.

D. CONCLUSION

Sensor spoofing attacks are a real threat to user privacy and device security. In this paper, we presented SDIOS, a defense mechanism that protects all applications on an Android device from spoofing attacks on the device's gyroscope. We showed that it is practical to integrate SDIOS into the Android operating system, and that it is easy to modify existing applications to use the extended SDIOS API. While the current implementation of SDIOS has several limitations, mainly stemming from the high CPU consumption of the machine learning model, we believe that it proves that it is conceptually possible to protect all applications on a device from sensor spoofing attacks. We have made all the artifacts of this research available as open-source software, and we hope that this will encourage further research in this area.

ARTIFACT AVAILABILITY

We plan to open-source the source code of SDIOS, together with the trained models we used in our experiments, after the conclusion of the anonymous peer-review process. The raw data used to train the models will not be released, due to concerns about the privacy of the participants.

REFERENCES

[1] Y. Son, H. Shin, D. Kim, Y. Park, J. Noh, K. Choi, J. Choi, and Y. Kim, "Rocking drones with intentional sound noise on gyroscopic sensors," in *USENIX Security Symposium*. USENIX Association, 2015, pp. 881–896.

[2] Y. Tu, Z. Lin, I. Lee, and X. Hei, "Injected and delivered: Fabricating implicit control over actuation systems by spoofing inertial sensors," in *USENIX Security Symposium*. USENIX Association, 2018, pp. 1545–1562.

[3] M. Abuhamad, A. Abusnaina, D. Nyang, and D. Mohaisen, "Sensor-based continuous authentication of smartphones' users using behavioral biometrics: A survey," *CoRR*, vol. abs/2001.08578, 2020.

[4] C. V. Anikwe, H. F. Nweke, A. C. Ikegwu, C. A. Egwuonwu, F. U. Onu, U. R. Alo, and Y. W. Teh, "Mobile and wearable sensors for data-driven health monitoring system: State-of-the-art and future prospect," *Expert Syst. Appl.*, vol. 202, p. 117362, 2022.

[5] P. Soobramaney, G. Flowers, and R. Dean, "Mitigation of the effects of high levels of high-frequency noise on mems gyroscopes using microfibrous cloth," vol. Volume 4: 20th Design for Manufacturing and the Life Cycle Conference; 9th International Conference on Micro- and Nanosystems, 2015. [Online]. Available: https://doi.org/10.1115/DETC2015-47378

[6] T. Trippel, O. Weisse, W. Xu, P. Honeyman, and K. Fu, "WALNUT: waging doubt on the integrity of MEMS accelerometers with acoustic injection attacks," in *EuroS&P*. IEEE, 2017, pp. 3–18.

[7] K. S. Tharayil, B. Farshteindiker, S. Eyal, N. Hasidim, R. Hershkovitz, S. Houri, I. Yoffe, M. Oren, and Y. Oren, "Sensor defense in-software (SDI): practical software based detection of spoofing attacks on position sensors," *Engineering Applications of Artificial Intelligence*, vol. 95, p. 103904, 2020.

[8] R. Ivanov, M. Pajic, and I. Lee, "Attack-resilient sensor fusion for safety-critical cyber-physical systems," *ACM Trans. Embed. Comput. Syst.*, vol. 15, no. 1, pp. 21:1–21:24, 2016.

[9] J. Levin, *Android internals: a confectioner's cookbook: the power users's view*, 1st ed. Cambridge, MA: Technologeeks.com, 2015, vol. volume 1. [Online]. Available: http://newandroidbook.com/

[10] Android open source project, "Sensormanager." [Online]. Available: https://developer.android.com/reference/android/hardware/SensorManager

[11] B. Farshteindiker, N. Hasidim, A. Grosz, and Y. Oren, "How to phone home with someone else's phone: Information exfiltration using intentional sound noise on gyroscopic sensors," in *WOOT*. USENIX Association, 2016.

[12] Z. Wang and T. Oates, "Imaging time-series to improve classification and imputation," in *IJCAI*. AAAI Press, 2015, pp. 3939–3945.

[13] M. Sakurada and T. Yairi, "Anomaly detection using autoencoders with nonlinear dimensionality reduction," in *MLSDA@PRICAI*. ACM, 2014, p. 4.

[14] S. C. Jinwon An, "Variational autoencoder based anomaly detection using reconstruction probability."

[15] "Expo," https://docs.expo.io/.

[16] Xiaomi Mi Community, "Redmi 9: Frequently asked questions," https://c.mi.com/bd/post/10046/, accessed: 2025-04-17.

[17] The LineageOS Project, "Lineageos android." [Online]. Available: https://lineageos.org/

[18] K. Corry, "Andromeda: A collection of android utilities for developers," https://github.com/kylecorry31/andromeda, accessed: 2023-10-01.

[19] M. Gao, L. Zhang, L. Shen, X. Zou, J. Han, F. Lin, and K. Ren, "KITE: exploring the practical threat from acoustic transduction attacks on inertial sensors," in *SenSys*. ACM, 2022, pp. 696–709.

[20] T. Hupperich, H. Hosseini, and T. Holz, "Leveraging sensor fingerprinting for mobile device authentication," in *DIMVA*, ser. Lecture Notes in Computer Science, vol. 9721. Springer, 2016, pp. 377–396.

[21] B. He, J. Bai, V. V. Zipunnikov, A. Koster, P. Caserotti, B. Lange-Maia, N. W. Glynn, T. B. Harris, and C. M. Crainiceanu, "Predicting human movement with multiple accelerometers using movelets," *Medicine and Science in Sports and Exercise*, vol. 46, no. 9, pp. 1859–1866, Sep. 2014. [Online]. Available: http://dx.doi.org/10.1249/MSS.0000000000000285

[22] Y. Michalevsky, D. Boneh, and G. Nakibly, "Gyrophone: Recognizing speech from gyroscope signals," in *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*, K. Fu and J. Jung, Eds. USENIX Association, 2014, pp. 1053–1067. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/michalevsky

[23] A. Das, G. Acar, N. Borisov, and A. Pradeep, "The web's sixth sense: A study of scripts accessing smartphone sensors," in *CCS*. ACM, 2018, pp. 1515–1532.

[24] Y. Chen, G. Liu, L. Yu, H. Kang, L. Meng, and T. Wang, "Tbauth: A continuous authentication framework based on tap behavior for smartphones," *Expert Systems with Applications*, vol. 264, p. 125811, 2025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0957417424026782

[25] D. E. Denning, "An intrusion-detection model," *IEEE Trans. Software Eng.*, vol. 13, no. 2, pp. 222–232, 1987.

[26] Z. Ferdousi and A. Maeda, "Unsupervised outlier detection in time series data," in *22nd International Conference on Data Engineering Workshops (ICDEW 06)*. IEEE, 2006, pp. x121—x121. [Online]. Available: http://dx.doi.org/10.1109/ICDEW.2006.157

[27] E. Sheetrit, N. Nissim, D. Klimov, and Y. Shahar, "Temporal probabilistic profiles for sepsis prediction in the ICU," in *KDD*. ACM, 2019, pp. 2961–2969.

[28] P. Novitski, C. M. Cohen, A. Karasik, G. Hodik, and R. Moskovitch, "Temporal patterns selection for all-cause mortality prediction in t2d with anns," *Journal of Biomedical Informatics*, vol. 134, p. 104198, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1532046422002039

[29] Y. Mirsky, T. Doitshman, Y. Elovici, and A. Shabtai, "Kitsune: An ensemble of autoencoders for online network intrusion detection," in *NDSS*. The Internet Society, 2018.
[30] G. Li, J. Ao, J. Hu, D. Hu, Y. Liu, and Z. Huang, "Dual-source gramian angular field method and its application on fault diagnosis of drilling pump fluid end," *Expert Systems with Applications*, vol. 237, p. 121521, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0957417423020237
[31] R. Xu, H. Saïdi, and R. J. Anderson, "Aurasium: Practical policy enforcement for android applications," in *USENIX Security Symposium*. USENIX Association, 2012, pp. 539–552.
[32] A. Pham, I. Dacosta, E. Losiouk, J. Stephan, K. Huguenin, and J. Hubaux, "Hidemyapp: Hiding the presence of sensitive apps on android," in *USENIX Security Symposium*. USENIX Association, 2019, pp. 711–728.
[33] F. Roesner and T. Kohno, "Securing embedded user interfaces: Android and beyond," in *USENIX Security Symposium*. USENIX Association, 2013, pp. 97–112.
[34] L. Zhang, K. Lian, H. Xiao, Z. Zhang, P. Liu, Y. Zhang, M. Yang, and H. Duan, "Exploit the last straw that breaks android systems," in *SP*. IEEE, 2022, pp. 2230–2247.
[35] A. K. Sikder, H. Aksu, and A. S. Uluagac, "6thsense: A context-aware sensor-based attack detector for smart devices," in *USENIX Security Symposium*. USENIX Association, 2017, pp. 397–414.
[36] K. Ma, B. Lu, S. Yin, C. Zhen, and H. Zhu, "Cadroid: A cross-combination attention based framework for android malware detection," *Expert Systems with Applications*, p. 129446, 2025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0957417425030623
[37] M. O'Neill, S. Heidbrink, J. Whitehead, T. Perdue, L. Dickinson, T. Collett, N. Bonner, K. E. Seamons, and D. Zappala, "The secure socket API: TLS as an operating system service," in *USENIX Security Symposium*. USENIX Association, 2018, pp. 799–816.

ROIE HERSHKOVITZ Received his M.Sc. degree in Information Systems Engineering at the Department of Software and Information Systems Engineering from Ben-Gurion University of the Negev, Beer-Sheva, Israel. His research interests include mobile security and privacy.

YOSSI OREN (SM'17) received his M.Sc. degree in computer science from the Weizmann Institute of Science, Rehovot, Israel, and his Ph.D. degree in electrical engineering from Tel Aviv University, Tel Aviv, Israel.

He is an Associate Professor in the Stein Faculty of Computer and Information Science at Ben-Gurion University of the Negev, Beer-Sheva, Israel. His research interests include implementation security, side-channel attacks, and embedded systems security.

Dr. Oren has served as a technical program committee member for the IEEE Symposium on Security and Privacy, as well as other security conferences.