Practical Attestation for Edge Devices Running Compute Heavy Machine Learning Applications

Ismi Abidi
ismi.abidi@cse.iitd.ac.in
Indian Institute of Technology Delhi
New Delhi, India

Vireshwar Kumar
viresh@cse.iitd.ac.in
Indian Institute of Technology Delhi
New Delhi, India

Rijurekha Sen
riju@cse.iitd.ac.in
Indian Institute of Technology Delhi
New Delhi, India

ABSTRACT

Machine Learning (EdgeML) algorithms on edge devices facilitate safety-critical applications like building security management and smart city interventions. However, their wired/wireless connections with the Internet make such platforms vulnerable to attacks compromising the embedded software. We find that in the prior works, the issue of regular runtime integrity assessment of the deployed software with negligible EdgeML performance degradation is still unresolved. In this paper, we present *PracAttest*, a practical runtime attestation framework for embedded devices running compute-heavy EdgeML applications. Unlike the conventional remote attestation schemes that check the entire software in each attestation event, PracAttest segments the software and randomizes the integrity check of these segments over short random attestation intervals. The segmentation coupled with the randomization leads to a novel performance-vs-security trade-off that can be tuned per the EdgeML application's performance requirements. Additionally, we implement three realistic EdgeML benchmarks for pollution measurement, traffic intersection control, and face identification, using state-of-the-art neural network and computer vision algorithms. We specify and verify security properties for these benchmarks and evaluate the efficacy of PracAttest in attesting the verified software. PracAttest provides 50x-80x speedup over the state-of-the-art baseline in terms of mean attestation time, with negligible impact on application performance. We believe that the novel performance-vs-security trade-off facilitated by PracAttest will expedite the adoption of runtime attestation on edge platforms.

CCS CONCEPTS

• **Computer systems organization** → **Embedded systems**; *Attestation*; Security; Performance.

KEYWORDS

Internet of things (IoT); edge devices; machine learning (ML); EdgeML; security; attestation.

The first author was supported by the Visvesvaraya PhD Scheme, MeitY, Govt. of India, with awardee number MEITY-PHD-2673.

ACSAC '21, Dec 6–10, 2021, Virtual Event, USA

ACM ISBN 9978-1-4503-8579-4/21/12…$15.00
https://doi.org/10.1145/3485832.3485909

ACM Reference Format:
Ismi Abidi, Vireshwar Kumar, and Rijurekha Sen. 2021. Practical Attestation for Edge Devices Running Compute Heavy Machine Learning Applications. In *Annual Computer Security Applications Conference (ACSAC '21), Dec 6–10, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3485832.3485909

1 INTRODUCTION

Extensive efforts have been made to process the collected raw data locally on the edge devices and to facilitate intelligent functionalities in the modern Internet of things (IoT) networks,. This recent focus stems from the considerations of reducing the network latency, saving bandwidth costs of transferring the raw data from the device to the cloud server, and resolving the privacy issues involved in sharing the raw data with third-party cloud servers [62]. The recent advancements in the domain of machine learning (ML) algorithms, such as compression of deployed models using quantization and pruning, training model with simpler architectures, and facilitating publicly available EdgeML libraries, make it possible to effectively process the collected data at the edge devices [18, 42, 44, 71, 73]. Further, advanced hardware support with specialized embedded accelerator for fast matrix multiplication, multi-core processors at a moderate cost, and embedded GPU and FPGA, contribute towards building powerful edge devices including Raspberry Pi, Hikey, and Odroid boards [31, 40]. Hence, the IoT devices running EdgeML applications find their usage in a variety of fields ranging from personal health monitoring to smart city management [69].

However, when IoT devices become an integral part of our daily lives, they also become easy targets for attacks that could disrupt our daily activities. Since these devices are connected to the Internet through wired or wireless protocols (typically wireless like WiFi, LoRa, NB-IoT, or Bluetooth for ease of deployment), they are exposed to network-based attacks [51]. An attacker can compromise the device's software to perform malicious activities by exploiting the security vulnerabilities in such protocols. For instance, after hacking into a wireless pacemaker, the attacker can manipulate the embedded software to report false heart rate values and prompt the patient to take/abandon critical medication [27]. Also, if a camera is deployed to record some sensitive activities, the embedded software might be compromised by the attacker to leak the recorded data [7]. Further, an attacker can exploit the compromised IoT devices as bots to launch a distributed denial-of-service (DDoS) attack jeopardizing the entire IoT ecosystem [5]. Therefore, ensuring regular runtime software integrity assessment of the IoT devices is of paramount importance for the security and privacy of the processed data.

One way to address such issues is to employ the hardware-based trusted execution environment (TEE), such as the ARM TrustZone [55] and Intel SGX [15]. The TEE architecture facilitates two environments within the same device: (1) the untrusted environment (or normal world) running the operating system (OS) with the application software, and (2) the trusted environment (or secure world) executing functions which cannot be altered by any software running in the untrusted environment. In this architecture, the attestation is carried out in the secure world to validate that the kernel running in the normal world has not been tampered [1, 2, 19, 20, 35].

In the conventional software attestation (CSA) schemes [49], the secure world performs the integrity assessment by computing a cryptographic hash of the *current* kernel and comparing it with the pre-stored *gold* hash of the *deployed* kernel. We highlight that in CSA, the integrity check of the entire kernel has to be conducted in each attestation event. This is because attesting parts of the software sequentially in different attestation events can leave the system vulnerable to a roving malware which can switch its location among the unattested parts of the kernel to avoid detection [4, 11]. Unfortunately, computing the cryptographic hash of the entire kernel takes millions of clock cycles which can translate into a significant execution time in a typical IoT device. For instance, on a Raspberry Pi 3 Model B board running at 1.2 GHz, utilizing Raspbian OS in the normal world and OPTEE kernel in the secure world, CSA takes around 2 seconds.

In CSA, if the *inter-attestation time* (i.e., the time between two attestation events) is long, the attacker cannot be detected if it can compromise the kernel, carry out the malicious activities and remove its trace in between the attestation events [9, 53]. However, if the inter-attestation time is short, the device cannot find enough time to run its applications. Specifically, it is impractical to utilize CSA in the IoT devices running a compute-heavy application, e.g., traffic intersection control using continuous real-time image processing with a deep neural network algorithm. Moreover, it is impossible to implement CSA for devices running EdgeML algorithms for safety-critical applications, which cannot be halted.

In this paper, we propose *PracAttest*, an attestation framework that employs the TEE architecture to facilitate a novel performance-vs-security trade-off in IoT devices running extremely compute-intensive EdgeML workloads (Section 4). In PracAttest, the OS kernel is divided into short segments. Then, in each runtime attestation event, PracAttest randomly selects one segment and attests it. It then determines an *attestation interval bound* based on the CPU usage of the application. It finally selects the inter-attestation time randomly between zero and the attestation interval bound. PracAttest triggers the next attestation event after this random inter-attestation time.

We point out that the kernel segmentation allows PracAttest to attest any segment independently and check its integrity. The application-driven determination of inter-attestation time brings forth the trade-off between the device security and the application performance. Further, the runtime randomization in the inter-attestation time and in selecting the kernel segment introduces unpredictability in exactly when the attestation is triggered and what segment is attested, respectively. It makes it difficult for an attacker to monitor the attestation events and target the device during the time windows when the attestation is not happening.

To evaluate the efficacy of PracAttest, we develop three real-world EdgeML workloads – *PolIoT*, *TrafIoT*, and *FaceIoT* (Section 5). PolIoT measures air pollution data and correlates pollution with road traffic using deep neural networks (DNN and LSTM) algorithms on vehicle-mounted low-cost IoT devices. TrafIoT utilizes the background subtraction and optical flow-based computer vision methods for the traffic density estimation and uses Reinforcement Learning based intersection control [12]. FaceIoT recognizes the human face in a captured image and matches it against the stored valid faces of individuals using one DNN for face detection and a second DNN for face matching. We highlight that the post-deployment attestation on edge devices is effective only if, before deployment, the edge computing software is verified according to the privacy, security, and other specifications. Thus, we also statically analyze and verify these requirements of the three EdgeML applications using Java Object-sensitive ANAlysis (JOANA). We show that JOANA successfully verifies our IoT benchmarks with no false negatives and a limited number of false positives that are manually vetted.

We demonstrate empirically that the application performance and security trade-offs are not adequately balanced in CSA and its trivial extensions in the above mentioned edge computing applications. The conventional attestation mechanism gives precedence to the security over the performance of the device, i.e., the integrity of the software is monitored with the desirable granularity by stalling the application execution leading to lower application performance. We also consider an attestation scheme called Shadow-Box [35] as our baseline. Shadow-Box considers the application's performance at the cost of security. Our results validate that PracAttest provides 50x-80x speedup in the attestation time compared to this baseline, achieving the desired security requirements without significantly affecting EdgeML application performance (Section 6). Our EdgeML benchmarks and the PracAttest framework have been open-sourced for field adoption and further optimizations by the research community[1]. Our major contributions are as follows.

- We design and build PracAttest, a novel runtime attestation framework that triggers the attestation of a randomly selected segment of the OS kernel at a randomly selected time. This skillful combination of the kernel segmentation and randomization enables PracAttest to realize short inter-attestation time (giving very little opportunity for any software tampering to go undetected) without significantly degrading the device's EdgeML application performance.
- We present three real-world EdgeML benchmarks PolIoT, TrafIoT, and FaceIoT, that use state-of-the-art deep neural networks (e.g., DNN and LSTM) and other computer vision algorithms (e.g., background subtraction and optical flow). Since the post-deployment software attestation is typically facilitated along with the pre-deployment software verification of the security and privacy requirements, we verify our EdgeML benchmarks for the sake of completeness.
- Using the three realistic workloads, we empirically show the shortcomings in balancing security-vs-performance trade-off by existing attestation schemes and highlight the advantages of PracAttest on actual Raspberry Pi as an edge device.

[1] https://github.com/iabidi/attestation

Table 1: Qualitative comparison of PracAttest with the prior works.

Scheme	Security Feature	Security Technique	Workload/Application	Suitable for Securing Heavy Workloads
Sensing [21, 29, 54, 72]	×	×	Pollution Measurement	×.
Deep Learning [47, 58, 73]	×	×	Image, Speech, ECG Classification	×
Preech [3]	User Input (Speech) Privacy	Differential Privacy (DP)	Speech Recognition (DNN)	×
TrustShadow [34]	Protection from Untrusted OS	Access Policy+Encryption	LMBench [50], Embedded Web Server	×
DIAT [19]	Control Flow Integrity	Control Flow Attestation	Flight Controller	×
LiteHax [2]	Control Flow Integrity	Control Flow Attestation	Syringe Pump	×
Shadow-Box, SMARM, HAtt [4, 11, 35]	Malware Detection	Remote Kernel Attestation	×	×
Conventional Software Attestation [49]	Malware Detection	Remote Kernel Attestation	×	×
PracAttest [This Work]	Malware Detection	Remote Kernel Attestation	Pollution Measurement (DNN, LSTM or SVM), Traffic Intersection Control (Computer Vision, Reinforcement Learning), Face Analysis (DNN)	✓

2 RELATED WORK

In the existing literature, the attestation of control flow, data, or memory regions have been explored on various embedded platforms like Odroid, RISC-V SoC, and Raspberry Pi [1, 2, 10, 19, 20, 37]. These works focus on providing security to simple applications like a syringe pump [19] and a flight controller of a drone [2]. Unfortunately, either they do not discuss the attestation time, or their schemes take a few seconds even for attesting such simple systems [2, 35]. In this paper, we handle much more compute-intensive edge workloads and seek to balance the performance-vs-security trade-off for them. Most papers on sensing [21, 29, 54, 72] or learning [47, 58, 73] on edge devices ignore device security, and focus on optimizing the ML performance on the constrained platforms. Recent attacks, on edge devices, like Mirai, Ransomware, and Triada malware [5, 27, 32] are compelling researchers to design a holistic system that guarantees performance as well as privacy and security. Among such few recent works are OMG [6], Sanctuary [8], and Preech [3] which seek to balance both the high compute requirement of the ML/DL algorithms and the privacy/security requirement of the system. In this paper, we extend this recent trend by evaluating the Conventional Software Attestation (CSA) mechanisms, examining the shortcomings based on real edge computing workloads, and fixing the shortcomings through careful design of PracAttest. Table 1 summarises the prior works designed to provide solutions for IoT application performance, security or both.

3 SYSTEM AND THREAT MODEL

3.1 Threat Model

We let the IoT devices be equipped with a hardware root of trust (e.g., ARM TrustZone) which cannot be compromised by an attacker without physically accessing the device. We assume that while the attacker cannot access the deployed devices physically, it can conduct network-based attacks. The scope of this paper includes malware-based attacks that manipulate the application or kernel code. We do not consider dynamic attacks like Return Oriented Programming (ROP) or Data Oriented Programming (DOP) attacks, as demonstrating such attacks in the context of the considered EdgeML applications is non-trivial in the presence of existing mechanisms for preserving control and data flow integrity [2, 57, 63].

3.2 System Architecture

Here, we provide a comprehensive ecosystem for securing IoT devices and point out the specific contribution made in this paper. Figure 1 presents the secure IoT system architecture. Before deploying the edge devices in the field, the EdgeML software needs to be formally verified to give security and privacy guarantees to the stakeholders. It ensures that IoT developers have not inadvertently introduced any vulnerability in the EdgeML software during the device's software development. This one-time verification component is shown in the top box in Figure 1. After deployment, as the application software starts running in the field, the remote attestation needs to be executed continuously. It ensures that the verified software is not manipulated at runtime. This recurring attestation component comprises of the following three components as shown in the bottom box in Figure 1.

3.2.1 EdgeML Application Integrity. The application software's integrity can be checked by comparing the hash of the source and executable files with the pre-stored gold hash. For calculating, storing, and regularly checking these hash values, the Integrity Measurement Architecture (IMA) can be used [39, 60] in the normal world of the device. IMA is a Linux kernel module that generates and appraises the hash value of the given files according to the specified policy. This not only ensures the integrity of the application software but also prevents other programs from altering the output files generated by the application software.

3.2.2 OS Integrity. While the IMA can monitor the integrity of the application software in the normal world, the integrity of the IMA must be checked by recurring attestation of the OS kernel. This attestation can be executed by the TEE in the secure world. In the conventional software attestation scheme, the cryptographic hash of the entire kernel running in the normal world is computed and then checked in the secure world. Unfortunately, in IoT devices, the whole kernel cannot be attested without significantly obstructing the application. In this paper, we propose *PracAttest* that runs in the secure world to practically assess the integrity of the OS kernel. PracAttest provides a framework for appropriate load balancing between the kernel attestation and EdgeML application execution to balance the security-vs-performance trade-off. The internal details of PracAttest are discussed in Section 4.

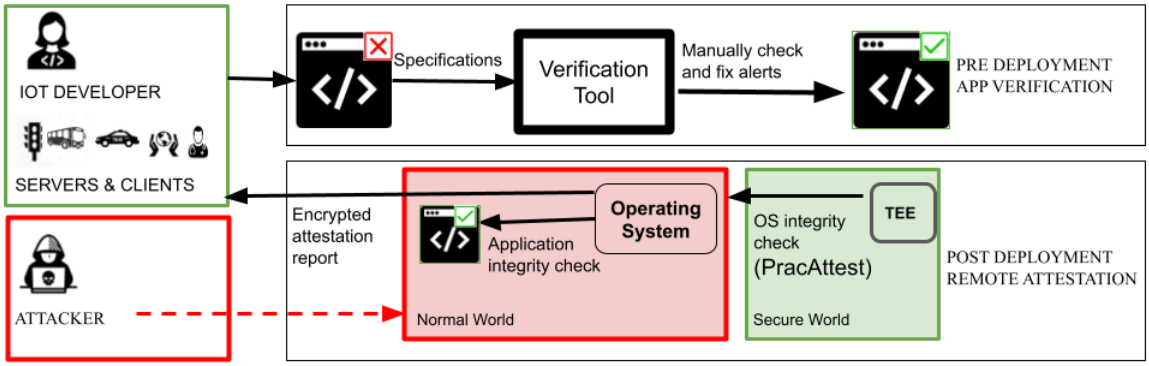


Figure 1: Comprehensive system architecture for securing IoT devices. The one-time pre-deployment verification of security requirements is conducted based on the policies specified by the IoT device developers and users (e.g., the EdgeML data servers and clients). The runtime attestation of the deployed software (running in the normal world) is conducted with the help of the hardware root of trust (in the secure world). The attacker aims to compromise the software running in the normal world.

Edge devices running ML applications typically use Linux OS, which has important kernel modules like IMA for application integrity checks, as mentioned above. The size of Linux is in MBs. There can be smaller custom OS like Uni-Kernels with smaller attack surface and lesser size than Linux to allow faster attestation. However, the support for ML frameworks is absent in such custom kernels [46]. Hence, Linux is still the default OS for practical edge devices running state-of-the-art EdgeML software (e.g., neural networks like DNN and LSTM). PracAttest, therefore, tries to optimize the Linux kernel attestation time through a careful design.

3.2.3 Attestation Reports. A remote server can request for the latest kernel attestation results. These results can be encrypted using a pre-shared secret key stored in the TEE. The query-response can also be associated with a nonce to ensure the response freshness. This way, any malware infection in the OS will be detected based on the encrypted response. If the remote server does not receive the response because the OS drops the network packets to and from TEE, it would consider the OS to be compromised.

4 PracAttest DESIGN

In this section, we focus on the problem of attesting the OS kernel while limiting the impact of the attestation event on the application performance. To solve this problem, there are two pertinent issues that need to be resolved by a runtime attestation scheme: (1) *how* should the kernel be attested in an attestation event, and (2) *when* should the attestation event be carried out or *what* should be the *inter-attestation time* (i.e., the time between two attestation events)?

We note that the conventional software attestation (CSA) scheme fails to resolve these issues. Figure 2 presents an illustration of the impact of attestation on an EdgeML application in CSA. In Figure 2, we observe that in the absence of any attestation scheme, the device can execute different instances of an EdgeML application. With CSA, whenever an attestation event is triggered, the entire kernel is attested consuming significant amount of CPU time. In CSA, halting the attestation event after partially attesting the kernel is not possible because this would make the device vulnerable to a roving malware-based attack [4, 11]. However, since the application

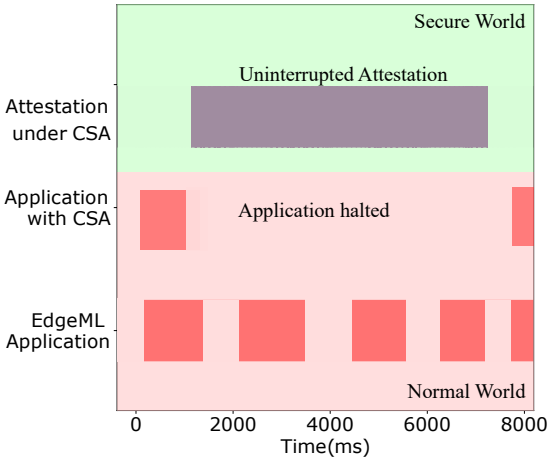


Figure 2: Continuous attestation hampers application in the Conventional Software Attestation (CSA) scheme.

cannot be executed during the attestation, CSA adversely affects the performance of the application. This illustration demonstrates that CSA gives an absolute priority to security over performance. We believe that this prioritization has been severely hindering the adoption of runtime attestation in real-world IoT devices.

In this paper, we present a novel attestation framework, PracAttest, that regularly performs attestation of the OS kernel of an IoT device running extremely compute-intensive EdgeML workloads. In PracAttest, before the deployment of the device, the kernel is divided into multiple short segments, and the cryptographic hash of each segment is independently calculated and stored in the secure memory. This way, the pre-stored gold hash of each kernel segment provides the flexibility to independently verify the integrity of any selected segment. After the deployment, when an attestation event is triggered at runtime, the secure world performs the following.

(1) It takes over the control of the device halting any EdgeML application running in the normal world.

(2) It *randomly* selects only one segment among all the kernel segments, and attests that segment. This addresses the first open issue of what part of kernel to attest.
(3) It retrieves the last *CPU usage sample* recorded in the normal world before the start of the attestation event.
(4) It defines an *attestation interval bound* which is set to be directly proportional to the value of the CPU usage sample.
(5) It selects the inter-attestation time *randomly* between zero and the attestation interval bound. The next attestation event is triggered after this inter-attestation time. This addresses the second open issue of when to attest the kernel.
(6) Finally, it grants the control back to the normal world resuming any stalled application.

We highlight that in each attestation event, only one kernel segment is attested. The inter-attestation time is determined based on the attestation interval bound which in turn is directly proportional to the CPU usage. This ensures that the inter-attestation time is small when the CPU usage is low, i.e., the attestation is mostly performed when the device is not running the EdgeML application. The randomization of the inter-attestation time and the kernel segments introduces unpredictability in exactly when the attestation is triggered and what segment is attested, respectively. This makes it difficult for an attacker to guess and target specific kern/el segments of the device during the time windows when the attestation is not happening. In this way, PracAttest resolves the two aforementioned issues of when and what to attest while enabling an advantageous coexistence between the attestation and application execution. Below, we elaborate on the design choices undertaken in the runtime mechanisms of PracAttest. The notations utilized in this paper are presented in Table 2.

Table 2: Notation utilized for the parameters in PracAttest.

Notation	Description
n	Number of kernel segments
k	Number of malignant segments
l	Number of segments selected for attestation
P_f	Probability of failure to detect malware
T_d	Duration of an instance of the application
T_m	Maximum attestation interval for the device
δ	Design parameter affecting the application
T_e	Duration of an attestation event
t_b	Attestation interval bound
t_a	Inter-attestation time

4.1 Selecting Kernel Segment

In PracAttest, the OS kernel is not attested at once but rather in segments. During an attestation event, the cryptographic hash of a randomly selected segment is matched with the corresponding gold hash. If an adversary makes any change in the selected kernel segment, the attestation fails, and the attack is detected. Let the time taken in each attestation event be denoted by T_e. We note that the value of the duration of one attestation event T_e can be determined experimentally for an IoT device and is agnostic to the application.

During multiple attestation events spread over time, PracAttest is able to randomly select and attest multiple kernel segments. This decreases the impact of attestation on the application performance, but it also decreases the system security, i.e., the ability to detect any malignant kernel segments. To analyze the impact of PracAttest on security, we provide the following probabilistic guarantee. Let the kernel memory be divided into n equal segments. Also, let there be k malignant segments. Over a time period, PracAttest runs l attestation events in which it randomly selects l segments for the attestation. Out of these l segments, PracAttest fails to detect the attack, i.e., the intrusion by malware, if it does not detect any malignant segment. Let the probability of failure of PracAttest in detecting the attack be denoted by P_f. Now we consider two types of malware: roving and non-roving malware.

4.1.1 Roving Malware. A roving malware can first infect a kernel segment, carry out malicious actions, move to another segment, and restore the previous segment to its benign state. By utilizing the capability of moving among the kernel segments stealthily, the malware can attempt to avoid detection by PracAttest. In this case, the probability of failure can be expressed as:

$$P_f = \left(\frac{n-k}{n}\right)^l. \tag{1}$$

4.1.2 Non-Roving Malware. A malware that remains static in infected segments, and does not move between two kernel segments is called a non-roving malware. In this case, the probability of the failure of PracAttest in detecting the malware can be expressed as:

$$P_f = \frac{\binom{n-k}{l}}{\binom{n}{l}} = \frac{(n-k)! \cdot (n-l)!}{(n-k-l)! \cdot n!}. \tag{2}$$

It is clear from both the above equations that as the number of selected segments l increases or the number of malignant segments k increases, the probability of failure P_f decreases.

4.2 Determining Inter-Attestation Time

A typical EdgeML application performs real-time sensing, makes certain computations, stores the relevant data, and finally communicates its decisions to the concerned entities. This sense-compute-store-communicate cycle can either be periodic or event-driven based on the nature of the application. In a periodic EdgeML application, typically, there exists a sleep instruction to control the data processing frequency and avoid device throttling. On the other hand, in an event-driven EdgeML application, the CPU is usually not busy before and after the event. PracAttest performs a fine-grained CPU usage sampling to record such patterns to find the appropriate low CPU usage windows to attest the kernel segments.

PracAttest needs to halt the application and switch to the secure world to attest the kernel segment. Hence, to limit the impact on the application, the time for the attestation and the time for retrieving the CPU usage samples must be selected skillfully. To address this challenge systematically, PracAttest proceeds as follows.

4.2.1 Setting Maximum Attestation Interval. PracAttest selects the inter-attestation time randomly between zero and an attestation interval bound. The *maximum attestation interval*, denoted by T_m, is defined as the attestation interval bound when the CPU usage is

detected to be 100%. Let the mean inference time of an instance of the application's EdgeML algorithm running without any interruption be denoted by T_d. The time T_d can be observed experimentally by running the application. Then, the maximum attestation interval is selected based on this mean inference time such that $T_m = \delta \cdot T_d$. Here, δ is a design parameter such that $0 \leq \delta \leq 1$. This design ensures that at least one attestation event is triggered during each instance of the application, but the extension in the application's mean inference time due to the attestation remains limited.

4.2.2 CPU Usage Sampling. The CPU usage sampling allows PracAttest to attest the kernel by following the execution profile of the application. This way, when the application is not running, PracAttest can aggressively attest the kernel segments. When the application is being executed, it halts the application only briefly. Due to the architectural limitations, the processor can be either in the normal world or in the secure world. Since the application runs in the normal world and the CPU usage of the application can only be recorded when the application is running, the sampling is performed in the normal world (as shown in Figure 3). In PracAttest, the CPU usage samples are collected in the normal world through by an application and then retrieved by the secure world to determine the inter-attestation time.

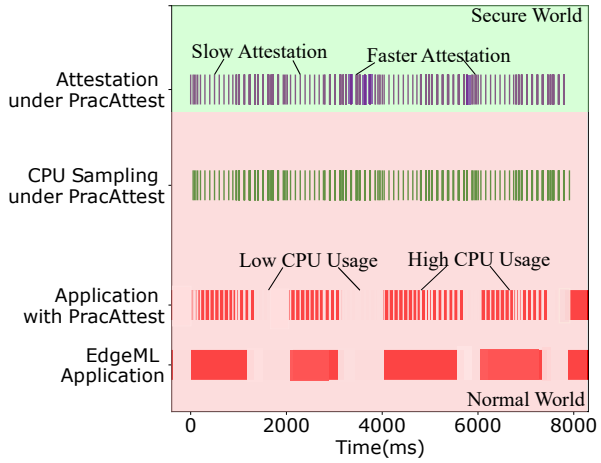


Figure 3: Flexible coexistence of attestation and application in PracAttest (the spaces represent the CPU idle time, the red lines represent the application execution time, the purple lines represent the kernel attestation time, and the green lines represent the CPU usage sampling time)

4.2.3 Determining Attestation Interval Bound. The attestation interval bound, denoted by t_b, is determined based on the retrieved CPU usage sample such that $t_b = u_c \cdot T_m$, where the retrieved CPU usage sample is denoted by u_c. This relationship ensures that the inter-attestation time is linearly dependent on the CPU usage values. This also ensures that if the CPU usage is high, a large attestation interval bound is selected. Similarly, if the CPU usage is low, a small attestation interval bound is selected.

4.2.4 Randomizing Inter-Attestation Time. We point out that determining the inter-attestation time using only the CPU usage value brings forth a critical security vulnerability. In this case, since the attacker can also track the CPU usage of the device, it can easily guess the inter-attestation time. To exploit this vulnerability, the attacker can execute the modified software right after an attestation event when the application is supposed to execute. To avoid detection, it can bring back the benign state of the software right before the next attestation event. To prevent such an attack, PracAttest also uses a pseudo-random number generator along with the CPU usage sampling to determine the inter-attestation time. The inter-attestation time, denoted by t_a, is selected randomly from the uniform distribution between zero and the attestation interval bound t_b. As seen from Figure 3, when the CPU usage is low, faster attestation takes place (denser purple lines), as small t_a is selected. We note that this randomness in inter-attestation time is bounded by the maximum attestation interval T_m. The random inter-attestation time mitigates the predictability of the attestation. Hence, even if the adversary obtains the CPU usage pattern or tweaks the CPU usage, the inter-attestation time remains unpredictable.

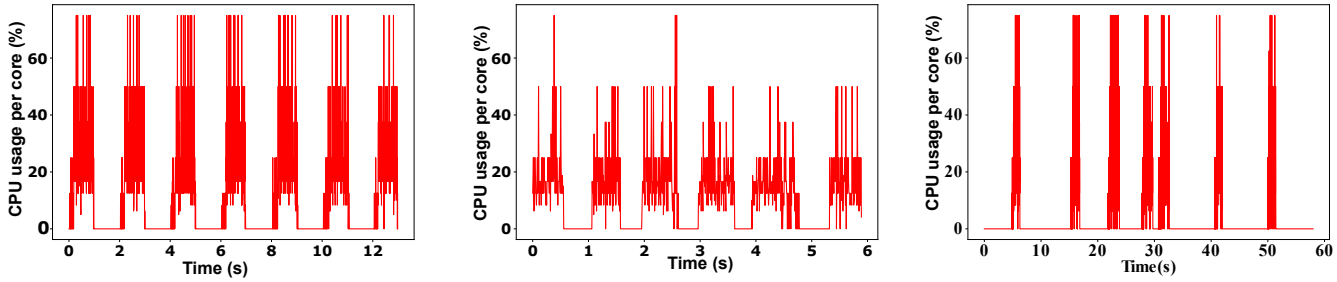
5 VERIFIED EDGEML BENCHMARKS

To assess the impact of attestation on edge computing applications, we need real-world workloads that can run on edge devices. These workloads will help us evaluate the efficacy of state-of-the-art attestation mechanisms, and then evaluate PracAttest's advantages over such baselines. Here, we describe three such workloads and discuss verification of some desirable properties of these applications.

5.1 Benchmark Applications

5.1.1 PolIoT: Air Pollution Measurement. Our first benchmark PolIoT measures the level of air pollution using vehicle-mounted low-cost IoT devices. In PolIoT, each edge device is equipped with multiple sensors to sample the relevant data, which is then processed locally at the device to avoid any overhead related to the wireless transmission of the data to the back-end servers. The device consists of five sensors: (1) particulate matter (PM) sensor to record the level of air pollution, (2) Global Positioning System (GPS) sensor to tag the PM value with the device's location, (3) temperature and humidity sensor (BME) to rectify the PM values, (4) inertial measurement unit (IMU) to detect any jerky movement leading to a noisy PM value, and (5) camera to take pictures of traffic in the device's vicinity to correlate the PM value with the vehicle counts. We keep the IMU sampling frequency at 50 Hz, based on literature review [41, 48, 61, 68]. All other sensors work at 0.5 Hz.

The device performs two heavy data processing tasks: (1) Deep Neural Network (DNN) based image processing for counting the number of vehicles and identifying the type of vehicles, and (2) Support Vector Machine (SVM) based analysis of the IMU data for tracking vehicle's movements including braking, turning and overspeeding. Overall, the IoT device runs four concurrent threads, which are shown in Table 3. As a result of this concurrency, the device records one PM value, one GPS value, one BME value, one DNN-based image inference result, and one SVM-based IMU inference result every two seconds. Figure 4a presents the CPU usage of the device running PolIoT.



(a) Air pollution measurement (PolIoT) (b) Traffic signal control (TrafIoT) (c) Event-driven face identification (FaceIoT)

Figure 4: CPU usage profile for the EdgeML benchmarks.

Table 3: Concurrent threads running in PolIoT.

1. Read PM on UART, GPS on UART, BME on I2C, write to SD card
2. Read image from camera, run DNN, write to SD card
3. Read IMU on I2C, run SVM, write to SD card
4. Read inferences from SD card, communicate to back-end server

5.1.2 TrafIoT: Traffic Signal Control. We further present another edge computing application, called TrafIoT, a road traffic density estimation-based traffic intersection control system [12]. To decide whether to keep/change the signal light, TrafIoT employs two metrics: (1) queue density corresponding to both static and moving vehicles, and (2) dynamic density corresponding to only moving vehicles. The queue density rises when the signal turns red and falls when the signal turns green. On the contrary, the dynamic density falls when the signal turns red and rises when the signal turns green. TrafIoT processes traffic images at five frames per second (FPS) and takes a signal control decision every 5 seconds (25 frames). To extract the information about the vehicles, TrafIoT subtracts the current image from a background mask. The background mask is periodically updated to handle changing lighting conditions. Further, an optical flow algorithm is used to detect changing pixels (moving vehicles) between consecutive images. These computer vision based inputs are fed into a Reinforcement Learning (RL) based signal control agent [12], which decides whether to keep or switch the current green signal. Figure 4b presents the CPU usage while running TrafIoT.

5.1.3 FaceIoT: Face Identification. We also design an edge computing application, FaceIoT, which captures and processes an image for identity verification. In FaceIoT, a passive infrared (PIR) sensor is triggered when a person enters a building. It then awakes the security camera to take a picture. The captured image is then processed to identify the face in the image using a DNN-based face detector. Once the face is detected, the face pixels are passed as an input to a DNN-based face classifier. The image classifier first classifies the face into a known or unknown face. If the face is known, it will record the person's details in the database, such as name and time of entry. However, if the face is unknown, it will raise an alert. We use FaceIoT as a representative event-driven EdgeML application. Figure 4c presents the CPU usage when running FaceIoT.

5.2 Benchmark Verification

Runtime attestation of these benchmarks after the field deployment uses the three components: (a) EdgeML application integrity check, (b) OS integrity check, and (c) attestation report communication to the remote server, as shown in the lower rectangle of Figure 1. But before deploying them in the field, the EdgeML benchmark software applications need to be verified based on the security and privacy specifications (top rectangle in Figure 1). We describe this verification procedure for our EdgeML benchmarks next, focusing on PolIoT, which has the camera along with other sensors and corresponding software. TrafIoT and FaceIoT comprise only the camera and related software, and therefore have security-privacy specifications as a subset of PolIoT security-privacy specifications.

5.2.1 Specification of Desired Properties. We assume that the IoT developers who build the hardware and software for the different applications (e.g., the traffic signal control or face biometric-based access control) and the deployment partners (e.g., the urban traffic control authority or security personnel for office buildings) will work in close collaboration for specifying the desired properties for these benchmark applications. We describe a sample of three desirable properties in the context of our benchmarks.

Data Privacy: TThe camera captures the faces of individuals with or without their consent. Additionally, the accelerometer, gyroscope, and GPS data are captured in PolIoT, using which sensitive information about the vehicle can be inferred. The collected data from IoT devices should assuredly go to only the dedicated server. There could be personal privacy concerns if face images get leaked in FaceIoT or public reputation-related concerns if the vehicle deploying PolIoT drives rashly (detected by GPS or IMU data in PolIoT). We must validate that the deployed software does not violate such privacy requirements.

Non-Interference: The PolIoT application has some specific non-interference requirements, as articulated by its users, e.g., environmentalists. Most policy debates related to air quality control are highly contentious. For instance, whether the urbanization should happen at the expense of the green cover [26, 65, 66], whether polluting industries should be shut down causing unemployment [16,

28, 38], whether farmers should incur economic losses to dispose of crop residues using non-polluting means [22, 23, 56], or whether on-road private vehicles should be reduced causing commuter hardships [24, 36, 64]. In this context, guaranteeing that the PolIoT software does not favor particular sides in a policy debate is necessary. For instance, the software should not deliberately reduce the PM values when the GPS value indicates that the device is near favored industries while boosting the PM values near industries targeted for shutdown. Ensuring non-interference across the sensors' data in the software is therefore needed.

Software Vulnerability Check: Our EdgeML benchmarks are safety-critical in nature, whether it is signal control at intersections (TrafIoT) or intrusion detection in buildings (FaceIoT). Data or code integrity violations or malware attacks on the system running these applications can lead to safety hazards. Therefore, it is important to check for the presence of known vulnerabilities in the software [27, 32, 70]. As there are many existing tools [13, 14] for checking software vulnerabilities like buffer overflow/underflow or string formatting, we do not discuss these specifications in detail. We instead focus on the data privacy and non-interference requirements, which need some analysis using off-the-shelf software verification tools.

5.2.2 Verification of Privacy and Non-Interference Properties. We employ the static information flow analysis which is a standard method for software verification [59]. Typically, the information flow control (IFC) is modeled in a system by defining the start point of the information flow in a program as the *source* and the end point as the *sink*. There are several methods to define information flow policies. One method is to label the data, variables and expressions in the program with security levels. These levels are then modelled as a lattice [17]. The lattice provides a way to check flows among different variables of the program, flagging all flows between higher to lower security levels as forbidden.

Data Privacy: To model this property, we select a lattice with two levels $\{low, high\}$ representing the low and high privacy requirements, respectively. We define that the information labeled as low is only allowed to flow into the information labeled as $high$, and not vice versa. This lattice is used to classify the sources and sinks in our software into the two privacy levels. Specifically, the data and the deployment partner's server URL are considered as the source and sink, and labeled with low and $high$ privacy level, respectively.

Non-Interference: To model the non-interference property, we employ the aforementioned lattice-based information flow verification. Here, we demonstrate it in the context of the PolIoT application. We model the lattice as shown in Figure 5. In the figure, each sensor datatype is given a separate label. The information labeled as $trusted$ is only allowed to flow into the information labeled as GPS and $server$, and not vice versa. Similarly, for all other sensors, the information can flow from $trusted$ to any of the sensor label and the $server$ label. Any flow between two sensor labels is illegal. For example, setting of the PM value based on the GPS value would lead to an illegal flow between the PM and GPS in the resultant dependence graph. To check these types of illegal flows, a non-interference IFC policy is defined such that one sensor node is considered as a source and other as a sink, and vice-versa.

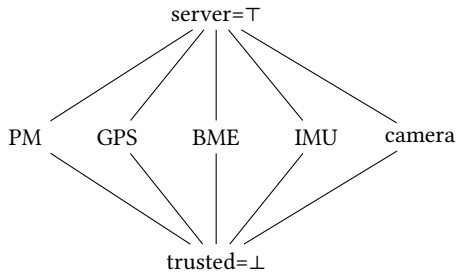


Figure 5: Lattice for PolIoT non-interference requirements (any information flow between sensor nodes is considered illegal).

Among various tools that we explored [25, 45, 52], JOANA is the one that works directly on Java bytecode. Hence, it is convenient to use for large external software libraries that are needed to implement realistic EdgeML applications. JOANA can verify both sequential as well as multi-threaded programs. It is one of the *sound* open-source verification tools available and requires few annotations. The above-mentioned properties make it a good choice for our verification requirements. The application executable is provided as the input to JOANA, where the IFC policies are specified. JOANA then raises alerts based on all violations of the IFC policies it finds in the executable. The developer has to iteratively go through these alerts and fix them unless they are false positives (vetted in collaboration with the respective stakeholders). When all true positive alerts are fixed, the executable is copied to the edge devices for deployment. As shown in Appendix A for the sample case of PolIoT application, JOANA correctly catches all data privacy and non-interference violations at very low and manually verifiable false positive rates.

6 EVALUATION

In this section, we will evaluate our post-deployment recurring attestation mechanism for compute-heavy EdgeML workloads.

6.1 Experimental Setup

The experiments are performed on Raspberry Pi 3B board with quad-core ARM Cortex A53 processors @1.2 GHz and 1 GB RAM. The Linux OS attestation scheme, PracAttest, is implemented using the virtualization. We run Raspbian Linux OS in the normal world and OPTEE kernel in the secure world [67]. The Linux IMA is configured to check the application integrity. PolIoT and FaceIoT benchmarks are implemented using OpenCV DNN APIs, and TrafIoT benchmark utilizes the OpenCV Computer Vision Library APIs. The C++ APIs, called with Java wrapper applications, are utilized for enabling software verification with JOANA on the Java bytecodes. The verified EdgeML software runs in the normal world Linux, while OPTEE in the secure world runs PracAttest to ensure Linux integrity. To realize PracAttest, the Linux kernel of size around 8.5 MB is divided into 2130 segments each of size 4 KB.

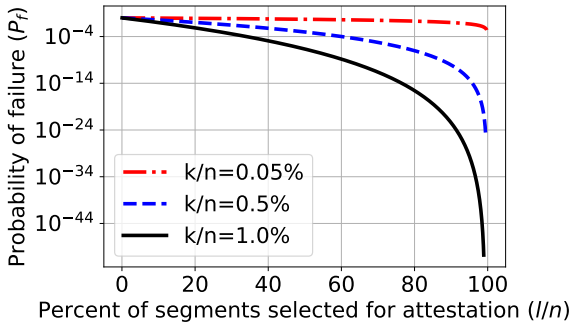


Figure 6: Ratio of segments selected for attestation and total segments ($n = 2130$) vs. P_f for a non-roving malware.

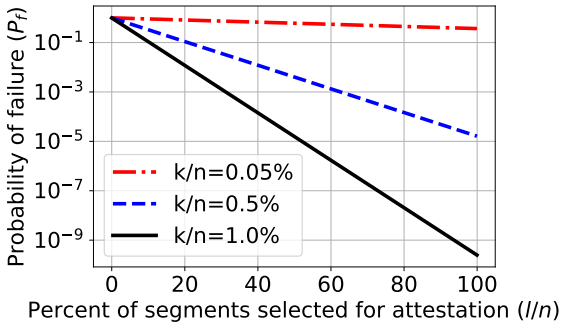


Figure 7: Ratio of segments selected for attestation and total segments ($n = 2130$) vs. P_f for a roving malware.

6.2 Evaluation Metrics

As our goal is to balance the performance-vs-security trade-off in EdgeML applications, we need to define metrics for both performance and security. ML application performance is typically measured by the inference latency, i.e., how much time each ML inference takes. The lower this number, the more performant is an EdgeML application. In our experiments, we run our benchmarks continuously for 15 minutes and report the mean inference times and their standard deviations (SD) as the metrics of performance.

Next, we define our security metric. Let PracAttest take T_p time to attest l segments. The probability of detecting at least one infected segment among the l segments is $1 - P_f$. Then the expected or mean attestation time, denoted by T_A, is calculated as

$$T_A = T_p(1 - P_f) + 2T_pP_f(1 - P_f) + 3T_pP_f^2(1 - P_f) + \ldots \quad (3)$$

$$= T_p/(1 - P_f) \quad (4)$$

The first term in Equation 3 refers to the case when the malware is detected within the first l segments (with probability $1 - P_f$) while the time to attest is T_p. The second term refers to the case when malware is not detected within the first l segments (with probability P_f), but detected in second l segments (with probability $1 - P_f$) while the overall attestation time is $2T_p$. The other terms in the equation can be inferred using a similar logic. We use this mean attestation time as our security metric. The lower the mean attestation time, the earlier the malware is detected, making the system more secure.

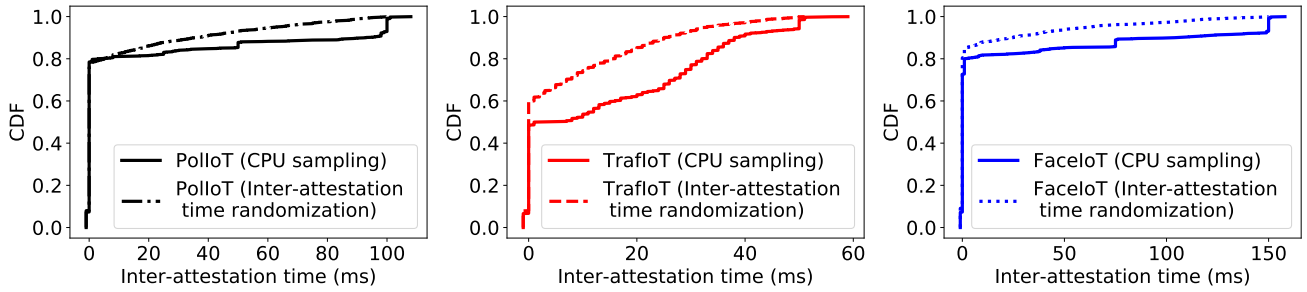
6.3 Number of Kernel Segments To Be Attested

PracAttest does not attest all kernel segments sequentially. Instead, it uses kernel segment randomization. Therefore it is important to evaluate how many kernel segments need to be attested for a desired low probability of failure P_f. Figure 6 shows the percentage of randomly selected kernel segments l/n along the x-axis vs. the probability of failure P_f to detect the malignant segments along the y-axis. The curves in this figure are plotted for non-roving malware using Equation 2. The trend shows that even if the number of infected kernel segments is only one, i.e., $k/n = 0.05\%$, there is a very low probability of missing the malware. In this case, if we attest at least 75% segments, the malware will be detected with the probability of 0.9. As the number of infected segments increases, the malware can be detected with fewer attestation of segments at a much higher probability. For instance, if the malware has infected around 21 kernel segments, i.e., $k/n = 1.0\%$, our system requires attestation of around 50% segments to ensure that it will fail to detect the malware only once in 10^6 runs. Figure 7 shows the same plots for the roving malware using Equation 1. Compared to the non-roving malware, this needs more kernel segments to be attested to detect the malware for the same k and P_f. For instance, when $k/n = 0.5\%$ and $l/n = 75.5\%$, the probability of detecting a non-roving malware is $1 - 10^{-8}$ while the probability of detecting a roving malware is slightly lower at $1 - 5 \cdot 10^{-4}$.

6.4 Inter-Attestation Time Distribution

In addition to choosing random kernel segments to attest, an important design choice for PracAttest is when to perform attestation, i.e., selecting appropriate inter-attestation time t_a. Figure 8 shows cumulative distribution functions (CDFs) of t_a values in msecs, as observed in experimental runs of the three benchmark applications. Recall that T_m is set by the IoT developer based on the EdgeML application's mean inference time by selecting the value of the parameter δ. In our experiments, we choose $\delta = 0.1$, i.e., T_m is set to be 10% of the EdgeML application's mean inference time – 100 msecs for PolIoT, 50 msecs for TrafIoT and 150 msecs for FaceIoT.

Recall that PracAttest samples the CPU usage to be $u_c\%$ where $0 \leq u_c \leq 100$. Equipped with *inter-attestation time randomization*, it randomly chooses t_a between 0 and $u_c\%$ of T_m. The dotted lines in Figure 8 show the CDFs of these t_a values. To highlight the significance of time randomization, we also consider a scheme where PracAttest is utilized without the time randomization, i.e., PracAttest chooses t_a as $u_c\%$ of the maximum attestation interval T_m. The solid lines in Figure 8 show the CDFs of these t_a values. Thus, compared to the solid curves, which have a step-like function for discrete CPU usage levels u_c, the dotted curves are smoother and have a higher proportion of smaller t_a values. The smoothness should make it more difficult for the attacker to guess the exact attestation time, and the smaller t_a values should help in reducing mean attestation time. We see the overall effect of these design choices on the performance-security metric trade-off next.



(a) PolIoT (b) TrafIoT (c) FaceIoT

Figure 8: CDF plots of inter-attestation time in different EdgeML benchmarks.

Table 4: Mean kernel attestation and inference time comparison.

Attestation Scheme	Mean Attestation Time/SD (s)			Mean Inference Time/SD (s)			Malware Detection Rate
	PolIoT	TrafIoT	FaceIoT	PolIoT	TrafIoT	FaceIoT	
No Attestation	-	-	-	0.9	0.53	1.36	0
Conventional Software Attestation [49]	2	2	2	-	-	-	1
Shadow-Box [35] against Non-Roving Malware	701/147	947/120	18.9/2	1.04/0.3	0.58/0.1	1.45/0.2	1
PracAttest without Time and Segment Randomization against Roving Malware ($k/n = 0.5\%, l/n = 100\%$)	35.2/1.8	29.7/1.4	43.6/1.8	0.96/0.2	0.54/0.1	1.39/0.1	0
PracAttest without Segment Randomization against Roving Malware ($k/n = 0.5\%, l/n = 100\%$)	19.1/1.9	16.2/1.0	20.4/1.4	0.95/0.2	0.54/0.1	1.39/0.2	0
PracAttest against Roving Malware ($k/n = 0.5\%, l/n = 75.5\%$)	14.3/2	12.2/0.8	15.4/1.1	0.95/0.2	0.54/0.1	1.4/0.2	$1 - 5 \cdot 10^{-4}$
PracAttest against Non-Roving Malware ($k/n = 0.5\%, l/n = 75.5\%$)	14.3/2	12.2/0.8	15.4/1.1	0.95/0.2	0.54/0.1	1.4/0.2	$1 - 10^{-8}$

6.5 PracAttest Performance-Security Trade-off

Table 4 shows the security and performance metrics for the three benchmark applications in different attestation schemes, including Conventional Software Attestation (CSA) [49], Shadow-Box [35] and PracAttest. In CSA, the entire kernel is attested in each attestation event. While only one gold hash (of length 256 bits) of the entire kernel is needed in CSA, PracAttest must use 66 KB of secure memory to store the gold hash values corresponding to 2130 kernel segments. PracAttest achieves advantage of application performance at the cost of a mild increase in secure memory usage and attestation time. We also consider Shadow-Box as a reasonable baseline that considers the application performance at the cost of security. Shadow-Box samples CPU usage every second and categorizes the measured CPU usage to three different levels: $0 - 30\%$, $31 - 70\%$ and $71\% - 100\%$. Then, the inter-attestation time is selected as 5 *msec*, 500 *msec*, and 2 *sec* corresponding to the three levels, respectively. In a periodic application like PolIoT and TrafIoT, sampling every second means that the sample will fall in the high CPU usage value with high probability before hitting a low CPU usage value. This coarse-grained CPU usage monitoring misses low CPU usage windows quite frequently. Hence, the inter-attestation times remain high, giving a high mean attestation time. Most importantly, interrupting the attestation to sample the CPU usage makes Shadow-Box vulnerable to the roving malware.

PracAttest significantly improves the mean attestation times of periodic applications such as PolIoT and TrafIoT. In Table 4, we also observe the impact of the successive design choices utilized in PracAttest. For instance, we consider PracAttest without the time and segment randomization while using the inter-attestation time t_a proportional to the CPU usage with a bound T_m on maximum values based on application. This scheme gives 20x and 33x improvement over Shadow-Box in terms of the mean attestation time for PolIoT and TrafIoT, respectively. We also consider PracAttest without segment randomization while randomizing the inter-attestation time. This scheme gives 37x-59x improvement over Shadow-Box. Finally, we observe that PracAttest gives 50x-80x improvement, with a very high probability of catching the malware. While Table 4 gives the value of the selected metrics for a particular value of l/n in segment randomization, Figure 9 shows how mean attestation time grows with l/n for a roving malware.

The event-driven application, FaceIoT, has plenty of idle windows, as shown in Figure 4(c), which can be utilized for aggressive kernel attestation. Therefore, Shadowbox and PracAttest have the same order of mean attestation times for FaceIoT. PracAttest's security advantages are thus more pronounced when the EdgeML application is compute-heavy, with small idle windows as in PolIoT and TrafIoT.

We also note that PracAttest slightly improves the mean inference time compared to Shadow-Box for all three benchmark

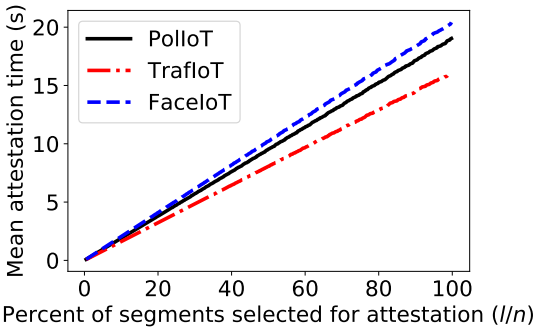


Figure 9: Mean attestation time for varying ratio of random segments and total segments ($n = 2130$) required for hashing to achieve a fixed P_f (≈ 0).

applications. Thus the tremendous security benefits (in terms of the mean attestation time) do not come at a cost to application performance (in terms of the mean inference time), but on the contrary, application performance also improves from choosing carefully when to attest. We point out that the slight increase in the inference latencies are well within our benchmarks' requirements, as seen from the first row in Table 4, where only the applications are run without any attestation mechanism in place.

7 CONCLUSION AND FUTURE WORK

In this paper, we propose PracAttest, a practical OS kernel attestation scheme for edge devices running compute-heavy EdgeML applications. Unlike the conventional software attestation scheme, which provides security at the cost of the application performance, PracAttest brings forth an advantageous security-vs-performance trade-off. We also present three EdgeML benchmarks and verify their data privacy and non-interference requirements with zero false negatives and acceptable false positives. Through these benchmarks, we demonstrate that our attestation tool, PracAttest, gives 50x-80x improved runtime for kernel attestation over state-of-the-art baseline, at negligible overhead on the ML application performance. With edge devices becoming an integral part of our lives, our practical immediately deployable attestation mechanism, PracAttest, can play an important role in securing ML at the edge.

In the future, we plan to explore the dynamic attack scenarios where attesting the static code (as done in this paper) does not suffice [2, 57, 63]. Such attacks could potentially be detected by monitoring the control flow path, when an EdgeML application runs. However, it is impractical to continuously explore and analyze all valid control paths in the secure memory of the edge device. For detecting such attacks, we plan to examine the potential of a PracAttest-like mechanism involving random selection and inspection of control flow paths. The design of such a security mechanism will again focus on minimizing the impact on the EdgeML application's performance. Another promising extension of our work is to investigate the feasibility of further minimizing the computational overhead of the remote attestation for very low-end resource-constrained IoT platforms, e.g., wearables.

REFERENCES

[1] Tigist Abera, N Asokan, Lucas Davi, Jan-Erik Ekberg, Thomas Nyman, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. 2016. C-FLAT: control-flow attestation for embedded systems software. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. 743–754.

[2] Tigist Abera, Raad Bahmani, Ferdinand Brasser, Ahmad Ibrahim, Ahmad-Reza Sadeghi, and Matthias Schunter. 2019. DIAT: Data Integrity Attestation for Resilient Collaboration of Autonomous Systems. In *NDSS*.

[3] Shimaa Ahmed, Amrita Roy Chowdhury, Kassem Fawaz, and Parmesh Ramanathan. 2020. Preech: A system for privacy-preserving speech transcription. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*. 2703–2720.

[4] Muhammad Naveed Aman, Mohamed Haroon Basheer, Siddhant Dash, Jun Wen Wong, Jia Xu, Hoon Wei Lim, and Biplab Sikdar. 2020. HAtt: Hybrid remote attestation for the Internet of Things with high availability. *IEEE Internet of Things Journal* 7, 8 (2020), 7220–7233.

[5] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J Alex Halderman, Luca Invernizzi, Michalis Kallitsis, et al. 2017. Understanding the mirai botnet. In *26th {USENIX} security symposium ({USENIX} Security 17)*. 1093–1110.

[6] Sebastian P Bayerl, Tommaso Frassetto, Patrick Jauernig, Korbinian Riedhammer, Ahmad-Reza Sadeghi, Thomas Schneider, Emmanuel Stapf, and Christian Weinert. 2020. Offline model guard: Secure and private ML on mobile devices. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 460–465.

[7] Bloomberg. 2021. Retrieved Jun 1, 2021 from https://www.bloomberg.com/news/articles/2021-03-09/hackers-expose-tesla-jails-in-breach-of-150-000-security-cams

[8] Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. 2019. SANCTUARY: ARMing TrustZone with User-space Enclaves.. In *NDSS*.

[9] Sergey Bratus, Nihal D'Cunha, Evan Sparks, and Sean W Smith. 2008. TOCTOU, traps, and trusted computing. In *International Conference on Trusted Computing*. 14–32.

[10] Xavier Carpent, Karim Eldefrawy, Norrathep Rattanavipanon, Ahmad-Reza Sadeghi, and Gene Tsudik. 2018. Reconciling remote attestation and safety-critical operation on simple IoT devices. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[11] Xavier Carpent, Norrathep Rattanavipanon, and Gene Tsudik. 2018. Remote attestation of IoT devices via SMARM: Shuffled measurements against roving malware. In *2018 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 9–16. https://doi.org/10.1109/HST.2018.8383885

[12] Sachin Chauhan, Kashish Bansal, and Rijurekha Sen. 2020. EcoLight: Intersection Control in Developing Regions Under Extreme Budget and Network Constraints. *Advances in Neural Information Processing Systems* 33 (2020).

[13] Edmund Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004) (Lecture Notes in Computer Science, Vol. 2988)*, Kurt Jensen and Andreas Podelski (Eds.). Springer, 168–176.

[14] Lucas Cordeiro, Pascal Kesseli, Daniel Kroening, Peter Schrammel, and Marek Trtik. 2018. JBMC: A Bounded Model Checking Tool for Verifying Java Bytecode. In *30th International Conference on Computer Aided Verification*.

[15] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptol. ePrint Arch.* 2016, 86 (2016), 1–118.

[16] Nandini Dasgupta. 2015. Tall Blunder. Retrieved Apr 12, 2019 from https://www.downtoearth.org.in/coverage/tall-blunder-22419

[17] Dorothy E Denning. 1976. A lattice model of secure information flow. *Commun. ACM* 19, 5 (1976), 236–243.

[18] Don Kurian Dennis, Yash Gaurkar, Sridhar Gopinath, Chirag Gupta, Moksh Jain, Ashish Kumar, Aditya Kusupati, Chris Lovett, Shishir G Patil, and Harsha Vardhan Simhadri. 2020. EdgeML: Machine Learning for resource-constrained edge devices. *URL https://github.com/Microsoft/EdgeML* (2020).

[19] Ghada Dessouky, Tigist Abera, Ahmad Ibrahim, and Ahmad-Reza Sadeghi. 2018. Litehax: lightweight hardware-assisted attestation of program execution. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–8.

[20] Ghada Dessouky, Shaza Zeitouni, Thomas Nyman, Andrew Paverd, Lucas Davi, Patrick Koeberl, N Asokan, and Ahmad-Reza Sadeghi. 2017. Lo-fat: Low-overhead control flow attestation in hardware. In *Proceedings of the 54th Annual Design Automation Conference*. ACM, 24.

[21] Jian Ding and Ranveer Chandra. 2019. Towards low cost soil sensing using Wi-Fi. In *The 25th Annual International Conference on Mobile Computing and Networking*. 1–16.

[22] Down To Earth. 2018. Crop burning: Haryana farmers to launch a state-wide protest. Retrieved Apr 12, 2019 from https://www.downtoearth.org.in/news/air/crop-burning-haryana-farmers-to-launch-a-state-wide-protest-61889

[23] Down To Earth. 2018. Crop burning: Why are Punjab farmers defying government ban. Retrieved Apr 12, 2019 from https://www.downtoearth.org.in/news/air/crop-burning-why-are-punjab-farmers-defying-government-ban-61869

[24] Ecotech. 2016. Odd-Even Policy, Delhi, Explained. Retrieved Apr 12, 2019 from https://www.ecotech.com/odd-even-policy-delhi-explained

[25] Marco Eilers and Peter Müller. 2018. Nagini: a static verifier for Python. In *International Conference on Computer Aided Verification*. Springer, 596–603.

[26] The Indian Express. 2018. 14,000 of 21,000 trees to be axed for redevelopment of south Delhi colonies: Govt. Retrieved Apr 12, 2019 from http://tinyurl.com/ybys6zro

[27] US Food and Drug Administration. 2017. Firmware Update to Address Cybersecurity Vulnerabilities Identified in Abbott's (formerly St. Jude Medical's) Implantable Cardiac Pacemakers: FDA Safety Communication. Retrieved Feb 26, 2021 from https://www.fda.gov/medical-devices/safety-communications/firmware-update-address-cybersecurity-vulnerabilities-identified-abbotts-formerly-st-jude-medicals

[28] Carnegie Council for Ethics in International Affairs. 2004. Workers' Rights and Pollution Control in Delhi. Retrieved Apr 12, 2019 from https://www.carnegiecouncil.org/publications/archive/dialogue/2_11/section_2/4451

[29] Jiahao Gao, Zhiwen Hu, Kaigui Bian, Xinyu Mao, and Lingyang Song. 2020. AQ360: UAV-aided air quality monitoring by 360-degree aerial panoramic images in urban areas. *IEEE Internet of Things Journal* 8, 1 (2020), 428–442.

[30] Dennis Giffhorn. 2012. *Slicing of Concurrent Programs and its Application to Information Flow Control*. Ph.D. Dissertation. Karlsruher Institut für Technologie, Fakultät für Informatik.

[31] Tiago Gomes, Sandro Pinto, Adriano Tavares, and Jorge Cabral. 2015. Towards an FPGA-based edge device for the Internet of Things. In *IEEE Conference on Emerging Technologies & Factory Automation (ETFA)*. 1–4.

[32] Google. 2019. PHA Family Highlights: Triada. Retrieved Feb 26, 2021 from https://security.googleblog.com/2019/06/pha-family-highlights-triada.html

[33] Jürgen Graf, Martin Hecker, and Martin Mohr. 2013. Using JOANA for Information Flow Control in Java Programs - A Practical Guide. In *Proceedings of the 6th Working Conference on Programming Languages (ATPS'13) (Lecture Notes in Informatics (LNI) 215)*. Springer Berlin / Heidelberg, 123–138.

[34] Le Guan, Peng Liu, Xinyu Xing, Xinyang Ge, Shengzhi Zhang, Meng Yu, and Trent Jaeger. 2017. Trustshadow: Secure execution of unmodified applications with arm trustzone. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*. 488–501.

[35] Seunghun Han, Junghwan Kang, Wook Shin, HyoungChun Kim, and Eungki Park. 2018. Shadow-BoxV2: The Practical and Omnipotent Sandbox for ARM. *Blackhat-ASIA* (2018).

[36] Deccan Herald. 2016. Delhi's odd-even scheme has no impact: study. Retrieved Apr 12, 2019 from https://www.deccanherald.com/content/666902/delhis-odd-even-scheme-has.html

[37] Ahmad Ibrahim, Ahmad-Reza Sadeghi, and Gene Tsudik. 2018. Us-aid: Unattended scalable attestation of iot devices. In *2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 21–30.

[38] Aditya Nigam in Revolutionary Democracy. 2001. Industrial Closures in Delhi. Retrieved Apr 12, 2019 from http://www.revolutionarydemocracy.org/rdv7n2/industclos.htm

[39] Trent Jaeger, Reiner Sailer, and Umesh Shankar. 2006. PRIMA: policy-reduced integrity measurement architecture. In *Proceedings of the eleventh ACM symposium on Access control models and technologies*. ACM, 19–28.

[40] Jongmin Jo, Sucheol Jeong, and Pilsung Kang. 2020. Benchmarking GPU-Accelerated Edge Devices. In *IEEE International Conference on Big Data and Smart Computing (BigComp)*. 117–120.

[41] Jair Ferreira Júnior, Eduardo Carvalho, Bruno V Ferreira, Cleidson de Souza, Yoshihiko Suhara, Alex Pentland, and Gustavo Pessin. 2017. Driver behavior profiling: An investigation with different smartphone sensors and machine learning. *PLoS one* 12, 4 (2017), e0174959.

[42] Ashish Kumar, Saurabh Goyal, and Manik Varma. 2017. Resource-efficient machine learning in 2 KB RAM for the Internet of Things. In *International Conference on Machine Learning (ICML)*. 1935–1944.

[43] Ralf Küsters, Tomasz Truderung, Bernhard Beckert, Daniel Bruns, Michael Kirsten, and Martin Mohr. 2015. A hybrid approach for proving noninterference of Java programs. In *2015 IEEE 28th Computer Security Foundations Symposium*. IEEE, 305–319.

[44] Aditya Kusupati, Manish Singh, Kush Bhatia, Ashish Kumar, Prateek Jain, and Manik Varma. 2018. FastGRNN: A fast, accurate, stable and tiny kilobyte sized gated recurrent neural network. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems (NIPS)*. 9031–9042.

[45] Shuvendu K Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. 2012. Symdiff: A language-agnostic semantic diff tool for imperative programs. In *International Conference on Computer Aided Verification*. Springer, 712–717.

[46] Matthew Leon. 2020. The Dark Side of Unikernels for Machine Learning. *arXiv preprint arXiv:2004.13081* (2020).

[47] En Li, Liekang Zeng, Zhi Zhou, and Xu Chen. 2019. Edge AI: On-demand accelerating deep neural network inference via edge computing. *IEEE Transactions on Wireless Communications* 19, 1 (2019), 447–457.

[48] Fu Li, Hai Zhang, Huan Che, and Xiaochen Qiu. 2016. Dangerous driving behavior detection using smartphone sensors. In *2016 IEEE 19th International Conference on Intelligent Transportation Systems (ITSC)*. IEEE, 1902–1907.

[49] Pieter Maene, Johannes Götzfried, Ruan De Clercq, Tilo Müller, Felix Freiling, and Ingrid Verbauwhede. 2017. Hardware-based trusted computing architectures for isolation and attestation. *IEEE Trans. Comput.* 67, 3 (2017), 361–374.

[50] Larry W McVoy, Carl Staelin, et al. 1996. lmbench: Portable Tools for Performance Analysis.. In *USENIX annual technical conference*. San Diego, CA, USA, 279–294.

[51] Francesca Meneghello, Matteo Calore, Daniel Zucchetto, Michele Polese, and Andrea Zanella. 2019. IoT: Internet of threats? A survey of practical security vulnerabilities in real IoT devices. *IEEE Internet of Things Journal* 6, 5 (2019), 8182–8201.

[52] Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. 2006. *Jif 3.0: Java information flow*. http://www.cs.cornell.edu/jif

[53] Ivan De Oliveira Nunes, Sashidhar Jakkamsetti, Norrathep Rattanavipanon, and Gene Tsudik. 2020. On the TOCTOU problem in remote attestation. *arXiv preprint arXiv:2005.03873 (to appear in CCS 2021)* (2020).

[54] Amitangshu Pal and Krishna Kant. 2019. Water flow driven sensor networks for leakage and contamination monitoring in distribution pipelines. *ACM Transactions on Sensor Networks (TOSN)* 15, 4 (2019), 1–43.

[55] Sandro Pinto and Nuno Santos. 2019. Demystifying Arm TrustZone: A comprehensive survey. *ACM Computing Surveys (CSUR)* 51, 6 (2019), 1–36.

[56] The Pioneer. 2017. Farmers protest Punjab Government's orders. Retrieved Apr 12, 2019 from https://www.dailypioneer.com/2017/state-editions/farmers-protest-punjab-governments-orders.html

[57] Davide Quarta, Marcello Pogliani, Mario Polino, Federico Maggi, Andrea Maria Zanchettin, and Stefano Zanero. 2017. An experimental security analysis of an industrial robot controller. In *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 268–286.

[58] Saeed Saadatnejad, Mohammadhosein Oveisi, and Matin Hashemi. 2019. LSTM-based ECG classification for continuous monitoring on personal wearable devices. *IEEE journal of biomedical and health informatics* 24, 2 (2019), 515–523.

[59] Andrei Sabelfeld and Andrew C Myers. 2003. Language-based information-flow security. *IEEE Journal on selected areas in communications* 21, 1 (2003), 5–19.

[60] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert Van Doorn. 2004. Design and Implementation of a TCG-based Integrity Measurement Architecture.. In *USENIX Security symposium*, Vol. 13. 223–238.

[61] Khaled Saleh, Mohammed Hossny, and Saeid Nahavandi. 2017. Driving behavior classification based on sensor data fusion using LSTM recurrent neural networks. In *2017 IEEE 20th International Conference on Intelligent Transportation Systems (ITSC)*. IEEE, 1–6.

[62] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. 2016. Edge computing: Vision and challenges. *IEEE Internet of Things Journal* 3, 5 (2016), 637–646.

[63] Zhichuang Sun, Bo Feng, Long Lu, and Somesh Jha. 2020. OAT: Attesting operation integrity of embedded devices. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1433–1449.

[64] Hindusthan Times. 2016. Air cleaner this April than last year, says body studying odd-even. Retrieved Apr 12, 2019 from https://tinyurl.com/y4uk9u47

[65] Hindustan Times. 2018. 16,500 trees: A huge price for south Delhi's redevelopment projects. Retrieved Apr 12, 2019 from https://tinyurl.com/y73te44m

[66] Hindustan Times. 2018. One tree cut every hour over last 13 years, says Delhi govt data. Retrieved Apr 12, 2019 from https://www.hindustantimes.com/delhi-news/one-tree-cut-every-hour-over-last-13-years-says-delhi-govt-data/story-uJBiGcLemQIOCvIfP7rwpN.html

[67] TrustedFirmware.org. 2020. Retrieved Sep 21 ,2020 from https://optee.readthedocs.io/_/downloads/en/3.9.0/pdf/

[68] Rohit Verma, Gyanesha Prajjwal, Bivas Mitra, and Sandip Chakraborty. 2018. Mining spatio-temporal data for computing driver stress and observing its effects on driving behavior. In *Proceedings of the 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM, 452–455.

[69] Xiaofei Wang, Yiwen Han, Victor CM Leung, Dusit Niyato, Xueqiang Yan, and Xu Chen. 2020. Convergence of edge computing and deep learning: A comprehensive survey. *IEEE Communications Surveys & Tutorials* 22, 2 (2020), 869–904.

[70] Jianliang Wu, Yuhong Nan, Vireshwar Kumar, Dave Jing Tian, Antonio Bianchi, Mathias Payer, and Dongyan Xu. 2020. {BLESA}: Spoofing Attacks against Reconnections in Bluetooth Low Energy. In *14th {USENIX} Workshop on Offensive Technologies ({WOOT} 20)*.

[71] Tien-Ju Yang, Yu-Hsin Chen, and Vivienne Sze. 2017. Designing energy-efficient convolutional neural networks using energy-aware pruning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 5687–5695.

[72] Yuzhe Yang, Zhiwen Hu, Kaigui Bian, and Lingyang Song. 2019. ImgSensingNet: UAV vision guided aerial-ground air quality sensing system. In *IEEE Conference on Computer Communications (INFOCOM)*. 1207–1215.

[73] Shuochao Yao, Yiran Zhao, Aston Zhang, Lu Su, and Tarek Abdelzaher. 2017. DeepIoT: Compressing deep neural network structures for sensing systems with a compressor-critic framework. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*. 1–14.

A PolIoT VERIFICATION RESULTS

Since PolIoT uses concurrent threads as shown in Table 3, we run JOANA's object sensitive, simple may-happen-in-parallel analysis [33]. It allows the Program Dependence Graph (PDG) to have a small number of the spurious interference dependence. Table 5 presents the different statistics of the generated PDG and the running time of IFC.

Table 5: Statistics of the PolIoT software verification.

Program Dependence Graph (PDG)	
Number of nodes	9783
Number of edges	58014
Build time	12701 ms
IFC analysis time	20 ms

Are violations correctly caught? Here, we evaluate the soundness of our JOANA-based verification implementation, by explicitly adding the code which violates the non-interference and privacy policy.
Privacy Violation: To guarantee the privacy requirement of the partner agency, let us analyze the sample code shown in Listing 2. Here, we annotate the partner agency's server URL as the *high* source (line 2-3), the attacker's URL as the *low* sink (line 4-5), and the function call to open the URL connection to be the *high* sink (line 10-11). This leaks the sensitive information to the unauthorized sink. Removing the line 7 and then constructing the URL as "`new URL(requestUrl)`" result in a secure flow.

Listing 2: Privacy requirement annotation sample.

```
1 public class SERVER{
2 @source(LEVEL.HIGH)
3 static String requestUrl=<Server URL>;
4 @sink(LEVEL.LOW)
5 static String atckerUrl= <Attacker URL>;
6     public void file_send(){
7         atckUrl=atckerUrl+"?"+requestUrl;
8         Url url = new URL(atckUrl);
9         HttpURLConnection httpURLCon = (HttpURLCon);
10        @sink(LEVEL.HIGH)
11        url.openConnection();
12        ...
13    }
14 }
```

Non-Interference Violation: We consider the sample code shown in Listing 3 for analyzing the non-interference violation. Here, the GPS is annotated as the *GPS* source (line 2-3) and the PM is annotated as the *PM* sink (line 4-5). There is a deliberate control flow between the GPS and PM (line 11-12). It is added in the code to test if JOANA can catch this violation.

Listing 3: Code snippet showing implicit flow.

```
1 public class PGB{
2     @source(LEVEL.GPS)
3     static String gps;
4     @sink(LEVEL.PM)
5     static String pm;//@Sink
6     public void bme_gps_pm_process(){
7         ...
8         gps=getgpsval();//@Source
9         pm=getpmval();//@Sink
10        ...
11        if(gps.equals("..")){
12            pm=...;
13        }
14        ...
15    }
16 }
```

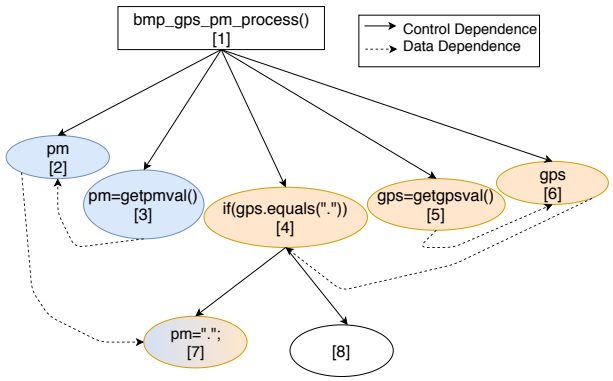


Figure 10: Simplified program dependence graph corresponding to Listing 3.

Figure 10 shows a simplified PDG for the Listing 3. We observe that there is a path from *gps* node 6 (yellow color) to node 7 and also a path from *pm* node 2 (blue color) to node 7. The node 7 should be of type *GPS*. But due to the propagation of security levels along the path 6-4-7 and 2-7, there is a conflict of security levels at node 7 (mixed color). This illegal flow is flagged by JOANA. Removing the line 11-13 from the Listing 3 will remove the generated violation. All the policies specified as the lattice shown in Figure 5 can be similarly verified in JOANA, as tested by us, by creating all possible violation examples. JOANA gives no false negatives.

Are false alarms easy to analyze? As JOANA is based on PDG, it *conservatively* finds explicit and implicit flows. Thus, it gives false alarms/positives [30, 43], which need to be manually analyzed to see if there is any real cause of concern.

Table 6: Results demonstrating that the number of false positives start decreasing once we identify the cause.

Sources↓	Number of file writes														
	A (all file writes enabled)					B (BME file writes removed)					C (BME, GPS file writes removed)				
Sinks→	PM	GPS	BME	IMU	Camera	PM	GPS	BME	IMU	Camera	PM	GPS	BME	IMU	Camera
PM	x	96	96	136	160	x	94	92	136	160	x	90	90	136	160
GPS	96	x	96	136	160	94	x	92	136	160	0	x	0	0	0
BME	96	96	x	136	160	0	0	x	0	0	0	0	x	0	0
IMU	144	144	144	x	240	147	147	144	x	240	138	135	135	x	240
Camera	96	96	96	136	x	94	94	92	136	x	92	90	90	136	x
Sources↓	D (BME, GPS, PM file writes removed)					E (all file writes removed)					F (all filewrites and two timestamps removed)				
Sinks→	PM	GPS	BME	IMU	Camera	PM	GPS	BME	IMU	Camera	PM	GPS	BME	IMU	Camera
PM	x	0	0	0	0	x	0	0	0	0	x	0	0	0	0
GPS	0	x	0	0	0	0	x	0	0	0	0	x	0	0	0
BME	0	0	x	0	0	0	0	x	0	0	0	0	x	0	0
IMU	132	132	132	x	240	129	129	129	x	117	0	0	0	x	0
Camera	88	88	88	136	x	86	86	86	92	x	0	0	0	0	x

Listing 4: Code snippet showing a false positive.

```
1 public class PGB{
2     @source(Level.GPS)
3     static String gps;
4     @sink(Level.BME)
5     static String bme;
6     public void bme_gps_pm_process(){
7         ...
8         while(true){
9             gps=getgpsval();//@Source
10            bme=getbmeval();//@Sink
11            ...
12            fr.write(gps);
13            ...
14        }
15    }
16 }
```

Listing 4 shows an example where JOANA raises a false alarm. A violation is flagged, citing the information flow from the *GPS* label to the *BME* (while there is no such implicit or explicit flow in the code). We inspected the PDG to find the reason for this false alarm. We observed that this leak occurs due to possible exceptions from the file write operation. These exceptions are added when the Java byte code is converted to PDG for the analysis by JOANA. The execution of the line 10 in Listing 4 depends on whether the file write is executed successfully or not. If it throws an exception, the program will terminate. Therefore, JOANA shows a flow from GPS to BME, i.e., no BME data is obtained if the file write of GPS is unsuccessful. Similar false positives have also been reported in the existing literature [30]. This violation is not dangerous for our non-interference policy as sensors are still not affecting each others' values and, therefore, safe to ignore.

Table 6 shows the number of violations detected by JOANA at the PDG level for the entire PolIoT software in six scenarios labeled A-F. We find that although the number of violations are high but they emanated from just a few lines in the code. It can be seen from Table 6 that once we start decreasing the number of file writes, the number of violation also decreases. This validates that the file write exceptions are the primary cause of the encountered false alarms.

There is a second kind of false alarm raised by JOANA for PolIoT. As shown in the scenario E in Table 6, the number of violation does not become zero in spite of having no file writes. Our Java program has three classes: IMU, Camera and PGB (for PM, GPS and BME) running concurrently. They invoke the same class for getting the timestamps of the sensor readings. The timestamp is returned in a local variable which is not shared between the threads. However, JOANA still reports an illicit flow from the source in the IMU class to the sinks in the PGB and Camera classes. Similarly, the source in Camera class shows illicit flows to the sinks in the other two classes. JOANA merges the three invocations of the timestamp class in different threads. We observe that if the timestamp is present in just one of the threads, we get the last case (scenario F) in Table 6 with zero violations. Therefore, it is safe to ignore this false alarm.

In essence, IoT developers have to iteratively go through the alarms raised by JOANA while fixing the true alarms and analyzing the false alarms to see if they can be safely ignored. As JOANA catches all violations and has few, easy to analyze false positives, we believe that this one-time manual process brings forth a significant security impact at an acceptable overhead.