DB-PAISA: **Discovery-Based Privacy-Agile IoT Sensing+Actuation**

Isita Bagayatkar
ibagayat@uci.edu
University of California, Irvine

Youngil Kim
youngik2@uci.edu
University of California, Irvine

Gene Tsudik
gene.tsudik@uci.edu
University of California, Irvine

Abstract

Internet of Things (IoT) devices are becoming increasingly commonplace in numerous public and semi-private settings. Currently, most such devices lack mechanisms to facilitate their discovery by casual (nearby) users who are not owners or operators. However, these users are potentially being sensed, and/or actuated upon, by these devices, without their knowledge or consent. This naturally triggers privacy, security, and safety issues.

To address this problem, some recent work explored device transparency in the IoT ecosystem. The intuitive approach is for each device to periodically and securely broadcast (announce) its presence and capabilities to all nearby users. While effective, when no new users are present, this *Push*-based approach generates a substantial amount of unnecessary network traffic and needlessly interferes with normal device operation.

In this work, we construct DB-PAISA which addresses these issues via a *Pull*-based method, whereby devices reveal their presence and capabilities only upon explicit user request. Each device guarantees a secure timely response (even if fully compromised by malware) based on a small active Root-of-Trust (RoT). DB-PAISA requires no hardware modifications and is suitable for a range of current IoT devices. To demonstrate its feasibility and practicality, we built a fully functional and publicly available prototype. It is implemented atop a commodity MCU (NXP LCP55S69) and operates in tandem with a smartphone-based app. Using this prototype, we evaluate energy consumption and other performance factors.

1 Introduction

In recent years, Internet of Things (IoT), embedded, and smart devices have become commonplace in many aspects of everyday life. We are often surrounded by cameras, sensors, displays, and robotic appliances in our homes, offices, public spaces, and industrial settings. With rapid advances in AI/ML, 5G, robotics, and automation, the use of IoT devices is becoming more and more prevalent.

IoT devices feature various sensors and/or actuators. Sensors collect information about the environment, while actuators control the environment. Some IoT devices use sensors to collect sensitive information, e.g., cameras, voice assistants, and motion detectors. Whereas, actuators on some IoT devices perform safety-critical tasks, e.g., operate door locks, set off alarms, and control smart appliances. This is not generally viewed as a problem for owners who install, deploy, and operate these devices. After all, owners should be aware of their devices' locations and functionalities. However, the same does not hold for casual users who come within sensing and/or actuation range of IoT devices owned and operated by others. Such users remain unaware unless nearby devices are human-perceivable, e.g., visible or audible.

This issue is partly due to lack, or inadequacy, of security features on commodity IoT devices, which manufacturers often justify with size, energy, and monetary constraints of the devices. Moreover, consumers gravitate towards monocultures, as witnessed by the great popularity of certain devices, such as Ring doorbells, Roomba vacuum cleaners, and Echo voice assistants. This leads to such massively popular IoT devices becoming highly attractive attack targets. Attacks can exploit devices' software vulnerabilities to exfiltrate sensed data, report fake sensed data, perform malicious actuation, or simply zombify devices [3, 8, 10, 21, 23, 73, 89, 104]. To mitigate these risks, a large body of work focused on constructing small Roots-of-Trust (RoT-s) for IoT devices. This includes remote attestation ($\mathcal{R}$A) [28, 33, 42, 87, 98], run-time attestation [2, 24, 36, 39], and sensor data protection [34, 50, 74]. However, all these techniques are research proposals; they are largely ignored by manufacturers who lack compelling incentives to introduce security features to their products.

Furthermore, most prior work on privacy and security for IoT ecosystems [7, 35, 41, 92, 105] and a few government IoT guidelines [30, 37, 38] focus on device owners or operators who are well aware of device presence and capabilities. As mentioned above, IoT devices also impact other users within the sensing and/or actuation range. Ideally, these casual nearby users must be informed of the device's presence and capabilities, which would enable users to make informed decisions.

Another motivation comes from data protection laws, such as the European General Data Protection Regulation (GDPR) [79] and California Consumer Privacy Act (CCPA) [69]. They aim to protect individuals' personal data and grant them control over data collection, processing, storing, and sharing. We observe that many (perhaps most) IoT devices operate in tandem with a cloud-based *digital twin* hosted by device manufacturers, software vendors, or users. In such cases, sensed data is processed and stored on devices as well as in the cloud. Therefore, logic dictates that IoT devices should also be subject to the same data protection laws.

Privacy and safety risks are not only apparent in public spaces, e.g., city streets, event venues, parks, campgrounds, airports, and beaches. They also occur in semi-private settings, e.g., conference rooms, hotels, and private rentals, such as Airbnb. In such places, privacy is expected, yet not guaranteed [5, 94, 106]. Casual visitors (users) are naturally suspicious of unfamiliar surroundings [60, 109] partly due to being unaware of nearby IoT devices.

Proceedings on Privacy Enhancing Technologies 2025(2), 434–449

https://doi.org/10.56553/popets-2025-0070

1.1 Current State of Device Transparency

To address the issues presented above, some recent research explored a privacy-agile IoT ecosystem based on manufacturer compliance. In particular, PAISA [63] allows users to learn about the presence and capabilities of nearby devices by listening to periodic secure announcements by each device using WiFi broadcast. Announcements are guaranteed to be sent in time even if the device is fully malware-compromised. Also, each announcement includes a recent attestation token, allowing a recipient user to check whether the device is trustworthy. PAISA works on commodity devices that have a Trusted Execution Environment (TEE) (e.g., ARM TrustZone [15]). It is unsuitable for low-end IoT devices without TEE support.

Another recent work, DIAL [66], requires each device to have a physically attached NFC tag. The device helps users locate it by either sounding a buzzer or using an ultra-wideband (UWB) interface. Upon physically locating a device, the user simply taps a smartphone on the NFC tag to get device information. Although DIAL does not require a TEE on each device, it imposes other requirements on the manufacturer and user, such as mounting an NFC tag and a buzzer or a UWB interface (rare on commodity IoT devices) for localization. It also requires physical access to a device to tap the NFC tag manually.

Both PAISA and DIAL follow the *Push* model: IoT devices announce their presence at fixed time intervals. This raises two concerns: (1) *unnecessary network traffic*, the volume of which can be high when numerous compliant devices are deployed in a given space, and (2) *interference with normal device operation* (in PAISA only), which is detrimental for devices that perform safety- or time-critical tasks. Both (1) and (2) are especially problematic when no new users are around to scan for these announcements. Also, as discussed above, DIAL's requirements for NFC-s and buzzers or UWB interfaces are currently unrealistic for most settings.

Another common application domain is large IoT deployments in industrial settings. Industrial IoT (*IIoT*) devices play an increasingly important role in automation across various industry sectors. Their numbers surpassed 3.5 billion in 2023, accounting for 25% of total IoT deployments [95, 96]. According to a recent Ericsson mobility report [46], the estimated number of connected devices in a typical smart factory is between 0.5 and 1 per square meter. This implies potentially millions of devices in a large industrial installation, e.g., a warehouse, port, or factory [100].

In such large *IIoT* systems, the owner/operator needs to maintain and keep track of all deployed devices. Unlike public and semi-public settings with casual users, the owner knows the identities and types of deployed *IIoT* devices. However, they might not know which devices are reachable, operational, or malware-compromised. They might also not know device locations since many industrial settings involve moving components. An intuitive approach is to use existing infrastructure to probe devices. For example, specialized software for managing merchandise or IoT devices associated with tagging information (e.g., barcode or NFC) can be used for inventory management [16, 68, 99, 107]. However, these techniques do not provide either reliable *current software state* or *current location* of moving devices.

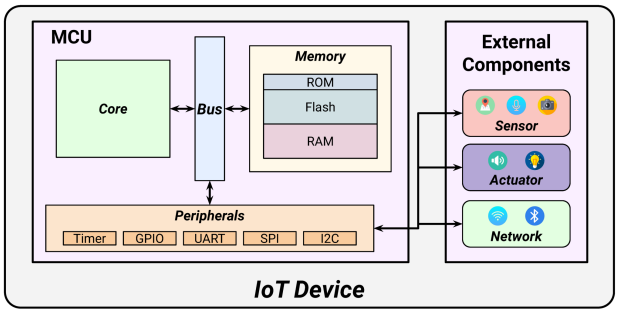


Figure 1: IoT Device Layout

1.2 Overview and Contributions

In this paper, we construct (1) DB-PAISA, an efficient privacy-compliant IoT technique geared for (semi-)public environments, and (2) IM-PAISA: *Inventory Management-PAISA*, geared for *IIoT* settings, described in Appendix A. Both techniques are *Pull*-based, meaning that a user (or an owner in IM-PAISA) issues an explicit request to learn about nearby devices. This obviates the need to generate and broadcast device announcements continuously. As a consequence, it reduces network load and interference with normal device functionality, especially when no new users are nearby. This additional communication step of sending a discovery request changes the adversary model, which results in new security challenges, discussed later in the paper.

DB-PAISA has two primary components: (1) a user device that sends a request and processes responses, and (2) compliant IoT devices that ensure a response containing software status and capabilities upon request. A manufacturer specifies the device information that can be released by its IoT devices.

Unlike relevant prior work [7, 34, 35], DB-PAISA requires no hardware modifications for low-end devices and relies on a popular TEE (ARM TrustZone [13]) to prioritize its tasks over other software. Also, DB-PAISA imposes no special requirements on user devices, except that IoT and user devices should support the same network interface, e.g., WiFi or Bluetooth.

Contributions of this work are three-fold:

- DB-PAISA, a pull-based IoT device discovery framework with low bandwidth overhead and no interference with normal device operation whenever no new users are present.
- A comparison of bandwidth, energy, and runtime overheads between PAISA's *Push* and DB-PAISA's *Pull* models.
- A full prototype of DB-PAISA comprised of: (a) an IoT device with ARM TrustZone-M and Bluetooth extended advertisements, and (b) an Android app that requests, scans, processes, and displays IoT device information. The implementation is publicly available at [67].

2 Background

2.1 Targeted IoT Devices

In general, we consider constrained IoT devices geared for executing simple tasks and deployed on a large scale, e.g., smart bulbs, locks, speakers, plugs, and alarms. Given tight resource budgets, they

typically rely on microcontroller units (MCUs) with no memory virtualization. Many of such MCUs have ARM Cortex-M [14] or RISC-V cores [82]. Our focus is on devices equipped with relatively simple TEEs, such as TrustZone-M. Note that we explicitly rule out low-end MCUs with no hardware security features.

As shown in Figure 1, a typical IoT device contains an MCU and multiple peripherals. The MCU includes a core processor(s), main memory, and memory bus, forming a System-on-a-Chip (SoC). Its primary memory is typically partitioned into: (1) program memory (e.g., flash), where software is installed, (2) data memory (RAM), which the software uses for its stack and heap, and (3) read-only memory (ROM), where the bootloader and tiny immutable software are placed during provisioning. Such an MCU also hosts peripherals, including a timer, General-Purpose Input/Output (GPIO), and Universal Asynchronous Receiver/Transmitter (UART).

The MCU interfaces with various special-purpose sensors and actuators through such peripherals. Common sensors are exemplified by microphones, GPS units, cameras, gyroscopes, as well as touch and motion detectors. Whereas, speakers, door locks, buzzers, sprinklers, as well as light and power switches, are examples of actuators.

Scope of Targeted IoT Device Types: For certain personal IoT devices, there is no need to inform nearby users of their presence since doing so can compromise their owners' privacy. This involves medical devices, e.g., blood pressure monitors, insulin pumps, catheters, or pacemakers. Clearly, neither PAISA nor DB-PAISA is applicable to such devices. Delineating the exact boundaries between devices' presence of which should (or should not) be released (or be discoverable) is beyond the scope of this paper.

2.2 Network

IoT devices are connected to the Internet and/or other devices either directly or through intermediate entities, e.g., hubs or routers. To support this, a typical IoT device features at least one network interface (e.g., WiFi, Bluetooth, Cellular, Ethernet, or Zigbee). WiFi and Cellular provide wireless Internet connectivity at relatively high speeds, while Bluetooth and Zigbee are geared towards lower-speed, shorter-range communication with neighboring devices. In this work, we focus on WiFi and Bluetooth since they are commonplace on both smartphones and IoT devices [11].

WiFi 802.11n (aka WiFi 4) [59] has a range of $\leq$ 75m indoors and $\leq$ 250m outdoors [1]. The latest WiFi standards (e.g., WiFi 6) achieve multi-gigabit speeds, making it ideal for bandwidth-intensive activities.

Bluetooth 5 [19] operates on a relatively shorter range, typically $\leq$ 40 meters indoors [26] and its data transfer speed peaks at $\approx$ 2Mbps. Since Bluetooth 4.0, Bluetooth devices have started supporting Bluetooth Low Energy (BLE). BLE is optimized for low power consumption, making it well-suited for resource-constrained IoT devices. Bluetooth Classic is utilized for connection-oriented data transfer, such as audio streaming. Whereas, BLE is widely used in beaconing applications (e.g., asset tracking, proximity marketing, and indoor navigation) for its power efficiency.

2.3 Trusted Execution Environment (TEE)

A TEE is a secure and isolated environment within a computer system, typically implemented as a hardware-based component. It provides a secure area for the execution of sensitive operations (e.g., cryptographic processing or handling of sensitive data) without interference or compromise from other software, operating systems, and hypervisors.

ARM TrustZone, one of the most prominent commercial TEEs, divides the system into two separate execution environments: Secure world and Normal (non-secure) world. Non-secure applications and interrupts cannot access computing resources, such as memory and peripherals, that are configured as secure. Within the ARM TrustZone framework, TrustZone-A (TZ-A) and TrustZone-M (TZ-M) refer to two specific implementations tailored for different types of processors; TZ-A is designed for high-performance application processors (e.g., smartphones), while TZ-M is tailored for low-power, resource-constrained MCUs commonly used in IoT devices. Although core security goals are the same for both TZ-A and TZ-M, techniques used to achieve these goals differ.

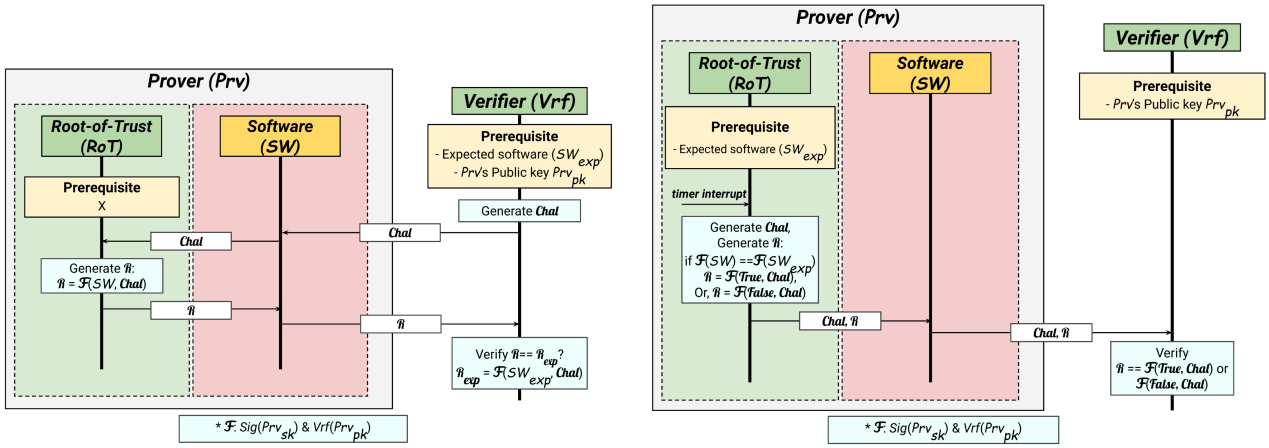
Unlike TZ-A, TZ-M is memory map-based. Memory areas and other critical resources marked as secure can only be accessed when the core is running in a secure state. Peripherals can also be designated as *secure*, which makes them exclusively accessible through and controlled by secure code. Besides two security regions (Secure world/Normal world), TZ-M introduces a non-secure callable (NSC) region. The NSC region allows secure functions to be safely called from non-secure code, exposing certain secure functions to Normal world without compromising overall system security.

2.4 Remote Attestation ($\mathcal{R}$A)

$\mathcal{R}$A is a security technique for verifying integrity and trustworthiness of software state on a remote entity. It allows a trusted server (verifier – $\mathcal{V}$rf) to securely determine whether a remote device (prover – $\mathcal{P}$rv) is running the expected software. As shown in Figure 2(a), $\mathcal{R}$A is typically realized as a challenge-response protocol:

(1) $\mathcal{V}$rf initiates $\mathcal{R}$A by sending a request with a unique challenge ($\mathcal{C}$hal) to $\mathcal{P}$rv.
(2) $\mathcal{P}$rv generates an unforgeable attestation report, which includes an authenticated integrity check over its software and $\mathcal{C}$hal, and returns it to $\mathcal{V}$rf.
(3) $\mathcal{V}$rf verifies the attestation report and determines $\mathcal{P}$rv's current state.

Figure 2(b) shows a variation of the above protocol, referred to as non-interactive $\mathcal{R}$A [9]. In the variant, $\mathcal{P}$rv autonomously decides when to trigger $\mathcal{R}$A and locally generates a unique and timely $\mathcal{R}$A report with $\mathcal{C}$hal. This obviates the need for $\mathcal{V}$rf to initiate the $\mathcal{R}$A process. Also, since $\mathcal{P}$rv knows the (hash of) benign software that it is supposed to execute, it can generate its own $\mathcal{R}$A reports indicating whether or not it is indeed running the expected software. This way, $\mathcal{V}$rf no longer needs to know the expected software configuration for each $\mathcal{P}$rv.



(a) Interactive (Traditional) $\mathcal{R}$A

(b) Non-Interactive and Self-Initiated $\mathcal{R}$A

Figure 2: Different Types of $\mathcal{R}$A

3 Design Overview

3.1 DB-PAISA Protocol Overview

DB-PAISA involves two main entities: an IoT device (I_{dev}) and a user device (U_{dev}). I_{dev} is equipped with a TEE and deployed in public or semi-private settings. U_{dev} can be any personal device with sufficient computing resources, such as a smartphone, smartwatch, or AR device. Also, a device manufacturer (Mfr) serves a minor (yet trusted) role in creating and maintaining accurate and up-to-date information for its I_{dev}-s.

Similar to prior work, all messages exchanged between I_{dev} and U_{dev} are broadcasted without any prior security associations or channel/session establishment. As mentioned earlier, in the *Push* model of PAISA [63] and DIAL [66], I_{dev} periodically announces itself, and U_{dev} (if present) receives and processes such information from nearby devices. Conversely, in DB-PAISA, the user initiates the process via an app on U_{dev}. U_{dev} broadcasts a discovery request and waits. Upon receiving a request, I_{dev} generates a response. Figure 3 depicts an overview of *Push* and *Pull* models. Unlike PAISA, which needs a time synchronization phase, DB-PAISA has only two phases: *Registration* and *Runtime*.

***Registration* Phase** takes place during the manufacturing of I_{dev}. Mfr installs software and provisions secrets for the underlying public key algorithm (i.e., I_{dev} key-pair) as well as some metadata.

***Runtime* Phase** has three steps: Request, Response, and Reception. Once completing the boot sequence, I_{dev} keeps listening for requests while performing normal operations. In Request step, U_{dev} broadcasts a request (Msg_{req}) to solicit device information from nearby I_{dev}-s and waits. Upon receiving Msg_{req}, I_{dev} generates and returns a response (Msg_{resp}) in Response step. Next, U_{dev} processes and verifies Msg_{resp} in Reception step, and displays I_{dev} information to the user. A detailed description of this protocol is in Section 4.

3.2 Adversary Model

An adversary ($\mathcal{A}$dv) can access all memory regions (e.g., flash and RAM), except for the TCB (defined in Section 4.1) and its data within the TEE. Outside the TCB, any I_{dev} components and peripherals, including timers, network interfaces, sensors, and actuators, are subject to compromise. Communication between I_{dev}-s and U_{dev}-s is subject to eavesdropping and manipulation by $\mathcal{A}$dv following the Dolev-Yao model [40]. Additionally, *Registration* phase is considered to be secure. Mfr is trusted to accurately provision IoT devices and safeguard their secrets. Similarly, DB-PAISA app on U_{dev} is considered trustworthy. Though it maintains no secrets, it is assumed to correctly generate formatted requests as well as accurately process and display information to the user.

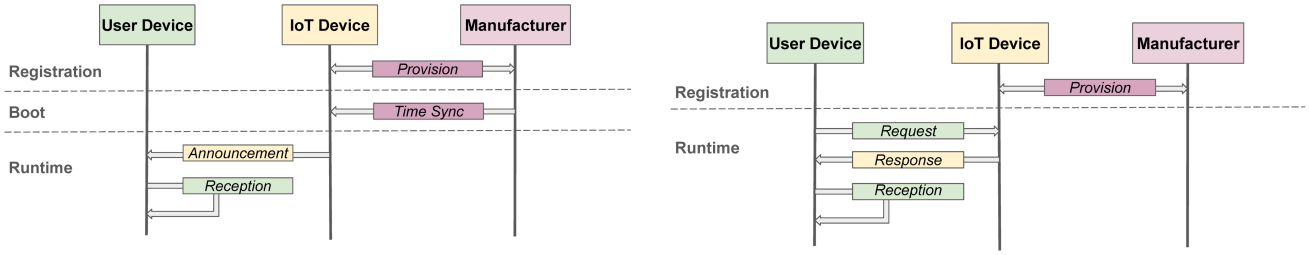
Denial-of-Service (DoS): $\mathcal{A}$dv can exploit vulnerabilities in non-TCB software to introduce malware, and then attempt to exhaust I_{dev}'s computing resources, e.g., cores, memory, and peripherals. Through such malware, it can also occupy (squat on) non-TCB-controlled peripherals from inside the device. Also, it can swamp peripherals from the outside, e.g., via excessive network traffic or fake request flooding, which is possible since requests are not authenticated. Sections 3.4 and 7.2 describe how DB-PAISA fully mitigates malware-based, and partially defends against network-based, attacks.

Replay Attacks: $\mathcal{A}$dv can reply with stale, yet valid, responses to U_{dev}. Replay attacks on I_{dev} are not a serious concern due to lack of authentication for Msg_{req}. Replays of stale Msg_{req}-s are a special case of request flooding mentioned above.

Wormhole Attacks: DB-PAISA does not defend against wormhole attacks [57]. In such attacks, $\mathcal{A}$dv tunnels both Msg_{req} and its corresponding Msg_{resp} to/from a remote[1] I_{dev}, thus fooling a U_{dev} into believing that a far-away I_{dev} is nearby. Well-known prior techniques [6, 22, 62, 65] can address wormhole attacks. However, mounting a wormhole attack in DB-PAISA is strictly harder than in PAISA, since $\mathcal{A}$dv must tunnel both Msg_{req} and Msg_{resp}, while only the latter suffices in PAISA.

Runtime Attacks: Similar to PAISA, DB-PAISA detects software modifications via periodic $\mathcal{R}$A. However, runtime attacks (e.g., control-flow and non-control-data) are out of scope. Prior research [2, 25,

[1] Here, "remote" means: beyond one-hop range of a U_{dev}.



(a) PAISA *Push* Model Overview

(b) DB-PAISA *Pull* Model Overview

Figure 3: *Push* and *Pull* Models Overview

32, 36, 39, 97] has proposed various mitigation techniques, such as control-flow attestation (CFA) and control-flow integrity (CFI). Unfortunately, these techniques are resource-intensive and thus usually impractical for lower-end IoT devices.

Physical Attacks: DB-PAISA protects against non-invasive physical attacks, e.g., reprogramming I_{dev} using a legal interface, such as JTAG. However, DB-PAISA does not protect against physically invasive attacks, such as hardware faults, ROM manipulation, or secret extraction via side-channels [108]. For defenses against these attacks, we refer to [80].

Non-compliant (Hidden) Devices: We do not consider attacks whereby $\mathcal{A}$dv gains physical access to an environment and surreptitiously deploys non-compliant malicious devices. For this purpose, we refer to some recent research on detecting and localizing hidden devices via specialized hardware [70, 81, 85, 93] and network traffic analysis [88, 90].

Co-existence with Other Secure Applications: In single-enclave TEEs (e.g., ARM TrustZone), all secure resources are shared by all secure applications. Consequently, a compromised secure application can access DB-PAISA secrets or mount a DoS attack by squatting on the network interface, which is part of DB-PAISA TCB. As in PAISA, we assume that all secure applications are free of vulnerabilities. Although this issue can be addressed by TEEs that support multiple enclaves (e.g., Intel SGX [61], ARM CCA [12]), such TEEs are generally unavailable on IoT devices targeted by DB-PAISA.

3.3 Requirements

Recall that DB-PAISA aims to facilitate *efficient* and *guaranteed* timely responses on I_{dev}-s upon requests from nearby U_{dev}-s. In terms of performance and functionality, DB-PAISA must satisfy the following properties:

- *Low latency:* Msg_{req} reception and Msg_{resp} generation time on I_{dev} must be minimal. DB-PAISA implementation should have a minimal impact on I_{dev}'s normal operation.
- *Low bandwidth:* DB-PAISA messages (Msg_{req} and Msg_{resp}) must consume minimal bandwidth.
- *Scalability:* I_{dev} should be able to handle multiple Msg_{req}-s and respond in time.
- *Casualness:* As in PAISA, no prior security association between U_{dev}-s and I_{dev}-s should be assumed, and secure sessions/handshakes between them must not be required.

To mitigate attacks defined in Section 3.2, DB-PAISA must provide:

- *Unforgeability:* U_{dev} must verify that Msg_{resp} comes from a genuine DB-PAISA-enabled I_{dev}, i.e., $\mathcal{A}$dv cannot forge Msg_{resp}.
- *Timeliness:* U_{dev} must receive Msg_{resp} containing I_{dev} information in a timely fashion.
- *Freshness:* Msg_{resp} must be fresh and reflect the recent software state of I_{dev}.

3.4 DoS Attacks

We now consider DoS attacks on U_{dev} and I_{dev}.

A network-based $\mathcal{A}$dv can mount DoS attacks by flooding U_{dev} with fake Msg_{resp}-s. This forces U_{dev} to verify numerous (invalid) signatures, clearly wasting resources. DB-PAISA offers no defense against such attacks, due to the casual nature of the I_{dev}-U_{dev} relationship. Also, this may not be a critical issue on higher-end U_{dev}.

As discussed in Section 3.2, there are two types of DoS attacks on I_{dev}: (1) an internal malware-based $\mathcal{A}$dv who keeps the CPU and/or network peripherals busy, and (2) a network-based $\mathcal{A}$dv who floods I_{dev} with frivolous Msg_{req}-s and thereby depletes computing resources with expensive cryptographic operations needed for Msg_{resp} generation. Similar to PAISA, (1) is mitigated by configuring the TEE such that a network peripheral is placed under the exclusive control of the TCB. Consequently, Msg_{req} reception and Msg_{resp} generation tasks have the highest priority.

Dealing with (2) is more challenging. Since Msg_{req}-s are not authenticated, a flood of fake Msg_{req}-s can simply exhaust I_{dev} resources. The same can occur in a non-hostile scenario when a flurry of valid requests from multiple benign users (e.g., in a suddenly crowded space) overwhelms I_{dev}.

To mitigate Msg_{req} flooding (whether hostile or not), DB-PAISA uses a lazy-response technique whereby I_{dev} collects Msg_{req}-s for a certain fixed time and responds to all of them with a single Msg_{resp}. Specifically, I_{dev} maintains a pool of nonces (up to a certain maximum number) from Msg_{req}-s. At the end of the period or when the nonce pool reaches its capacity, I_{dev} signs a collective Msg_{resp} containing all pooled nonces. Each U_{dev} with an outstanding Msg_{req} checks whether the received Msg_{resp} contains its nonce. This approach allows I_{dev} to pace its compute-intensive cryptographic operations. Details are in Section 4.

We acknowledge that pooling nonces from Msg_{req}-s does not fully address Msg_{req} flooding: $\mathcal{A}$dv can simply flood I_{dev} at a high enough speed, causing I_{dev} to generate signed Msg_{resp}-s at an unsustainable rate. One remedial measure is to adopt random deletion

– a relatively effective countermeasure to the well-known TCP SYN-flooding attack [84]. Furthermore, nonce pooling delays Msg_{resp} generation, making U_{dev}-s wait longer. To address this issue, an alternative approach (albeit with a slightly different setting) is discussed in Section 7.2.

4 DB-PAISA Design

Recall that DB-PAISA has two phases: *Registration* and *Runtime*, as shown in Figure 3(b).

4.1 Registration Phase

In this phase, Mfr provisions each I_{dev} with device software (SW_{dev}), DB-PAISA trusted software ($SW_{DB-PAISA}$), cryptographic keys, and other metadata. Also, the timer and network peripherals are configured as secure via the TEE. Finally, Mfr configures the attestation interval ($T_{\mathcal{A}tt}$) and the lazy-response delay ($T_{\mathcal{G}en}$) according to each I_{dev} use case. Their use is described in Section 4.2.

Software: Normal functionality of I_{dev} is managed by SW_{dev}, which resides in and executes from the non-secure memory region. Meanwhile, $SW_{DB-PAISA}$ is installed in the secure region.

Cryptographic Keys: There is a public/private key-pair for I_{dev} (pk_{IoT}, sk_{IoT}). The latter is used to sign Msg_{resp}. This key pair is generated inside the TEE. pk_{IoT} is shared with Mfr, while sk_{IoT} never leaves the TEE. pk_{Mfr} is assumed to be trusted, e.g., via a public key infrastructure (PKI) dealing with all pk_{Mfr}-s.

Device Manifest ($Manifest_{IoT}$) is the information about I_{dev}, generated and maintained by Mfr. It must include Mfr certificate ($Cert_{Mfr}$) and I_{dev} certificate ($Cert_{IoT}$) signed by $Cert_{Mfr}$. The exact contents of $Manifest_{IoT}$ are up to the individual Mfr. However, some fields are mandatory: pk_{IoT}, certificates, type/model of I_{dev}, sensors/actuators it hosts, etc. Note that $Manifest_{IoT}$ is not placed into I_{dev}; it is hosted by Mfr at a URL, URL_{Man}.

Once U_{dev} obtains $Manifest_{IoT}$, it uses $Cert_{Mfr}$ to authenticate it, and extracts pk_{IoT} from $Cert_{IoT}$. $Manifest_{IoT}$ might contain other device information, e.g., purposes of its sensors/actuators, the specification link, SW_{dev} version, and coarse-degree deployment geographical location, e.g., country, state, or city.[2]

URL (URL_{Man}): $Manifest_{IoT}$ size can be large since it depends on Mfr and type/model of I_{dev}. Thus, we cannot expect it to fit into one Msg_{resp}. Therefore, $Manifest_{IoT}$ is indirectly accessible at a shortened URL, called URL_{Man}, contained in each Msg_{resp}. URL_{Man} is a shortened version of $URL_{Man_{Full}}$, created using a URL shortening service, such as `Bitly`[18] or `TinyURL`[101], for making it short and of constant size.

Metadata: The following parameters are stored in the secure memory region of I_{dev}: (1) URL_{Man}, (2) $H_{SW_{dev}}$– hash of SW_{dev}, (3) $T_{\mathcal{A}tt}$– inter-attestation time parameter, (4) $T_{\mathcal{G}en}$– Msg_{resp} generation interval, and (5) $|Pool_N|_{Max}$– maximum size of the nonce pool.

TCB: Cryptographic keys, $SW_{DB-PAISA}$, and all aforementioned metadata (stored in the secure memory region) are considered to be within the TCB, along with the timer and network interface, which are configured as secure peripherals.

[2]This coarse location information can aid in (partially) mitigating wormhole attacks.

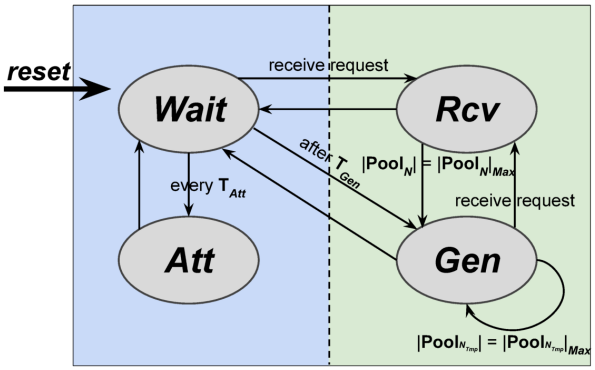


Figure 4: DB-PAISA **State Machine on** I_{dev}

4.2 Runtime Phase

Runtime phase composed of three steps: Request, Response, and Reception.

4.2.1 DB-PAISA *Trusted Software ($SW_{DB-PAISA}$) on* I_{dev}. As shown in Figure 4, I_{dev} has four states in Response step: $\mathcal{W}$ait, $\mathcal{A}$tt, $\mathcal{R}$cv, and $\mathcal{G}$en. $\mathcal{A}$tt is a periodic state independent of others: its periodicity is governed by $T_{\mathcal{A}tt}$.

(a) $\mathcal{W}$ait: After its boot sequence completes, I_{dev} runs SW_{dev} while listening for Msg_{req}-s on the network interface. Depending on the condition below, it transitions to other states:

- *Every* $T_{\mathcal{A}tt}$: I_{dev} periodically attests SW_{dev} via a secure timer set off every $T_{\mathcal{A}tt}$. When the timer expires, I_{dev} proceeds to **$\mathcal{A}$tt**.
- Msg_{req} *received*: Once I_{dev} identifies an incoming packet as a Msg_{req} (via DB-PAISA protocol identifier – $ID_{DB-PAISA}$, in a header field), it transitions to **$\mathcal{R}$cv**.
- *After* $T_{\mathcal{G}en}$: It proceeds to **$\mathcal{G}$en**.

(b) $\mathcal{A}$tt: I_{dev} computes a hash over SW_{dev} and compares it to $H_{SW_{dev}}$. If they do not match, the 1-bit Att_{result} flag is set to `Fail` and to `Success` otherwise. Next, I_{dev} securely records its current timer value ($time_{L_{\mathcal{A}tt}}$) and returns to **$\mathcal{W}$ait**.

An intuitive alternative to timer-based attestation is to perform attestation upon each Msg_{req}. The attestation itself consumes far less time/energy than signing Msg_{resp} However, for a low-end I_{dev} that performs safety-critical tasks, decoupling attestation from discovery can be more appealing.

(c) $\mathcal{R}$cv: As discussed in Section 3.4, DB-PAISA uses a flexible lazy-response strategy to amortize signature costs and mitigate potential DoS attacks via Msg_{req} flooding. In this state, I_{dev} maintains $Pool_N$, composed of nonces extracted from Msg_{req}-s.

- I_{dev} initially transitions to this state when it receives the first Msg_{req}. It extracts the nonce from Msg_{req}, N_{usr}, and places it into a (initially empty) nonce pool $Pool_N$. It then sets a secure timer to $T_{\mathcal{G}en}$ and returns to **$\mathcal{W}$ait**. Note that the timer for $T_{\mathcal{G}en}$ is only set when the first Msg_{req} comes in.
- If $Pool_N$ is not empty, I_{dev} extracts the nonce and adds it to $Pool_N$. Once $Pool_N$ reaches $|Pool_N|_{Max}$, I_{dev} transitions to **$\mathcal{G}$en**. Otherwise, it returns to **$\mathcal{W}$ait**.

- If it has been transitioned from $\mathcal{G}$en (i.e., Msg_{req} arrives while generating Msg_{resp}), I_{dev} extracts the nonce and adds it to a temporary list – $Pool_{N_{Tmp}}$, and returns to $\mathcal{G}$en.

Although $|Pool_N|_{Max}$ is a configurable parameter, it is physically upper-bounded by the amount of space available in Msg_{resp}, which depends on the network interface and the underlying wireless medium. This is further discussed in Section 5.4.

Note that if $|Pool_N|_{Max}$ is set to 1, a separate Msg_{resp} is generated for each Msg_{req}. Also, $|Pool_N|_{Max}$ should not be set over the physical upper bound because it leads to all Msg_{req}-s ignored until the current Msg_{resp} is completely handled. This can result in a false sense of privacy, frustrating users to erroneously think there are no I_{dev}-s around.

(d) $\mathcal{G}$en: I_{dev} computes attestation time ($time_{\mathcal{A}tt}$) as:

$$time_{\mathcal{A}tt} = time_{IoT} - time_{L_{\mathcal{A}tt}}$$

where $time_{IoT}$ is the current timer value. Then, it generates $Att_{report} := [Att_{result}, time_{\mathcal{A}tt}]$, signs the message $Sig_{resp} := \mathcal{S}IG(N_{dev}||Pool_N||URL_{Man}||Att_{report})$, and creates $Msg_{resp} := [N_{dev}, Pool_N, URL_{Man}, Att_{report}, Sig_{resp}]$. $\mathcal{S}IG$ is a signing operation and N_{dev}, generated by I_{dev}, is added to randomize Msg_{resp}. Next, I_{dev} assigns $Pool_{N_{Tmp}}$ to $Pool_N$, resets $Pool_{N_{Tmp}}$ to be empty, and broadcasts Msg_{resp}. Note that if an influx of Msg_{req}-s (i.e., # of Msg_{req}-s > $2*|Pool_N|_{Max}$) is received within $T_{\mathcal{G}en}$, $|Pool_{N_{Tmp}}|$ exceeds $|Pool_N|_{Max}$. Then, the first block (of length $|Pool_N|_{Max}$) of N_{usr}-s from $Pool_{N_{Tmp}}$ is moved to $Pool_N$ after the current Msg_{resp} is sent out. The remaining N_{usr}-s in $Pool_{N_{Tmp}}$ are in turn handled similarly when $\mathcal{G}$en is triggered. Finally, it goes to the next state with the below conditions:

- If $|Pool_{N_{Tmp}}| < |Pool_N|_{Max}$, it transitions to $\mathcal{W}$**ait**.
- Otherwise, it re-enters $\mathcal{G}$**en** to process a new Msg_{resp} with next N_{usr}-s.

If any Msg_{req}-s are received in this state, it goes to $\mathcal{R}$cv to handle them.

4.2.2 DB-PAISA *App on* U_{dev}. There are two steps: Request & Reception.

Request: U_{dev} initiates device discovery by broadcasting a Msg_{req} that contains a unique N_{usr} and $ID_{DB-PAISA}$. It then waits for Msg_{resp}-s for a time period set as part of the DB-PAISA app configuration.

Reception: Upon reception of Msg_{resp}, U_{dev} parses it and checks for the presence of N_{usr}. If not found, Msg_{resp} is discarded. Next, U_{dev} fetches $Manifest_{IoT}$ from URL_{Man}. It then retrieves pk_{Mfr} from $Cert_{Mfr}$ and verifies the signature of $Manifest_{IoT}$ using pk_{Mfr}. Successful verification implies that both $Manifest_{IoT}$ and pk_{IoT} are trustworthy. Finally, U_{dev} verifies Sig_{resp} with pk_{IoT} and displays the details to the user.

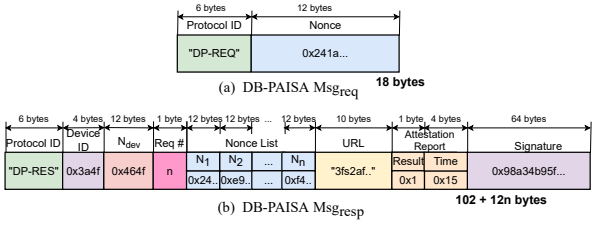
5 Implementation

This section describes DB-PAISA implementation details. All source code is available at [67].

5.1 Implementation Setup

I_{dev} is implemented on an NXP LPC55S69-EVK [76] board. It features an ARM Cortex-M33 processor with ARM TrustZone-M (TZ-M), running at 150 MHz with 640 KB flash and 320 KB SRAM. Due to lack of a built-in network module, we use an ESP32-C3-DevKitC-02 board [47], connected to the NXP board via UART. As U_{dev}, we use a Google Pixel 6 Pro Android smartphone [51] with eight heterogeneous cores running at ≤ 2.8 GHz. Figure 5 shows the overall experimental setup.

Figure 5: Implementation Setup



(a) DB-PAISA Msg_{req}

(b) DB-PAISA Msg_{resp}

Figure 6: Msg_{req} **and** Msg_{resp} **Formats (n: # of requests within** $T_{\mathcal{G}en}$**, and n** ≥ 1)

Secure Peripherals: We configure UART4 and CTIMER2 as secure, granting $SW_{DB-PAISA}$ exclusive control over the network and timer peripheral. UART4 and CTIMER2 are configured with the highest priority values (0 and 1, respectively) to ensure timely and guaranteed reception of Msg_{req}-s and generation of Msg_{resp}-s. If SW_{dev} needs to access the network peripheral to communicate with an external entity, it can use UART4 for its normal operation via $SW_{DB-PAISA}$. $T_{\mathcal{A}tt}$ is set to 300s and $T_{\mathcal{G}en}$ – to 1s on CTIMER2. Also, CASPER and HASH-AES Crypto Engine peripherals facilitate cryptographic operations, i.e., signing Msg_{resp}-s and hashing (non-secure) memory during attestation.

5.2 Network

Device Discovery: We focus on popular wireless media types: WiFi and Bluetooth. Each has a network discovery procedure. To initiate discovery, one typically broadcasts a short (specially formatted) network discovery message. Most fields in this message are reserved and of fixed size. Thus, using network discovery messages as Msg_{req}-s and Msg_{resp}-s is not trivial.

Msg_{req} & Msg_{resp} Length: As shown in Figure 6, Msg_{req} size is constant – 18 bytes. The minimum size of a Msg_{resp} is 114 bytes for a single nonce. Since one Msg_{resp} can reply to multiple Msg_{resp}-s

(each with its own nonce), we need to encode the total number of nonces in front of the nonce list. We use one extra leading byte for this purpose. In general, a network discovery packet must allow for at least 114 bytes of DB-PAISA-specific data.

Bluetooth *vs* WiFi: Both WiFi and Bluetooth are common on many types of IoT devices as well as smartphones, smartwatches and tablets. To support lightweight connection-less communication between I_{dev}-s and U_{dev}-s, our DB-PAISA prototype uses Bluetooth extended advertisements. The rationale for this choice is three-fold: (1) Bluetooth 5 supports broadcast messages of $\leq$ 1,650 bytes, while IEEE 802.11 WiFi can only carry $\leq$ 255 bytes in a vendor-specific field of a beacon frame, (2) it is more energy-efficient than WiFi [11], and (3) its indoor range of $\approx$ 40m is more appropriate for DB-PAISA, since I_{dev}-s discovered via WiFi may be irrelevant to a U_{dev} due to being too far.

5.3 Normal Operation on I_{dev} (SW_{dev})

We implemented a temperature sensor application as I_{dev}'s normal functionality – SW_{dev}. The same application was used to motivate and evaluate PAISA. It obtains temperature sensor readings on I_{dev} using LPADC (Low-Power Analog-to-Digital Converter) driver every 5s, and sends the data to a remote server. Due to UART4 being set as secure, SW_{dev} cannot use it directly; it first goes through $SW_{\text{DB-PAISA}}$, running in Secure world, to send packets. This is implemented using a Non-Secure Callable (NSC) function, which is the only valid entry point to transition from Normal world to Secure world in TZ-M, except for secure interrupts. SW_{dev} is implemented as a simple task on freeRTOS [86], reading the temperature sensor with 5s delay in a loop.

5.4 DB-PAISA Trusted Software

DB-PAISA includes three trusted applications: (1) secure application ($SW_{\text{DB-PAISA}}$) running in Secure world on I_{dev}, (2) network stack connected to I_{dev} via secure UART4, and (3) Android app on U_{dev}.

***$SW_{\text{DB-PAISA}}$* on I_{dev}:** DB-PAISA uses a secure timer in three cases: (1) triggering an interrupt at $T_{\mathcal{A}tt}$ (300s) to perform attestation, (2) triggering an interrupt at $T_{\mathcal{G}en}$ (1s) for the lazy-response mechanism, and (3) computing estimated attestation time. The first two require the timer interrupt to be triggered at different intervals. Fortunately, most commercial timers support multiple conditions triggering the interrupt. Hence, $SW_{\text{DB-PAISA}}$ requires only one exclusive secure timer.

$SW_{\text{DB-PAISA}}$ has four software components: UART interrupt service routine (ISR), timer ISR, attestation, and Msg_{resp} generation. Once it boots, I_{dev} continuously listens for Msg_{req}-s. UART ISR is triggered whenever I_{dev} receives a packet from the network module. It identifies Msg_{req}-s by $ID_{\text{DB-PAISA}}$ (`"DP-REQ"`) and prioritizes handling them. If $Pool_N$ is empty, it places N_{usr} into $Pool_N$ and sets the secure timer to expire after $T_{\mathcal{G}en}$. Otherwise, it adds (appends) N_{usr} to $Pool_N$.

As mentioned earlier, two conditions can trigger timer ISR: $T_{\mathcal{A}tt}$ and $T_{\mathcal{G}en}$. This timer ISR checks which timer expired. If it is $T_{\mathcal{A}tt}$, ISR computes `SHA256` over non-secure memory (including SW_{dev}) and generates Att_{result}. Otherwise, if $T_{\mathcal{G}en}$ expired, timer ISR initiates Msg_{resp} generation process, which (1) computes $time_{\mathcal{A}tt}$, (2) signs Msg_{resp} contents (N_{dev}, $Pool_N$, URL_{Man}, and Att_{report}) using `ECDSA` (`Pri- me256v1` curve), (3) composes Msg_{resp}, and (4) hands it over to the network module via `UART_WriteBlocking()`. Note that, in order to prioritize Msg_{resp} generation, the timer interrupt from $T_{\mathcal{A}tt}$ is inactivated when $T_{\mathcal{G}en}$ is reached. Also, when $Pool_N$ is empty, the timer interrupt from $T_{\mathcal{G}en}$ is **not** in use, i.e., not set.

When $Pool_N$ reaches its capacity ($|Pool_N|_{Max}$), Msg_{resp} generation is triggered. Msg_{resp} generation can also be triggered by $T_{\mathcal{G}en}$ expiring. UART ISR has the highest priority to receive Msg_{req}-s and process N_{usr}-s, even during Msg_{resp} generation. If UART ISR receives a new Msg_{req} while Msg_{resp} is being generated, N_{usr} from this new Msg_{req} is stored in a separate temporary list, $Pool_{N_{Tmp}}$. It is fed into $Pool_N$ after handling the current Msg_{resp}.

The current minimum nonce length recommended by NIST for lightweight cryptography [29] is 12 bytes. In Bluetooth 5, a single Msg_{resp} can contain $\leq$ 129 nonces, meaning that it can collectively respond to as many Msg_{req}-s. This upper bound is based on the maximum capacity of Bluetooth extended advertisements (1,650 bytes) and other Msg_{resp} fields, e.g., 64-byte signature, and 5-byte Att_{report}.

Network Module: ESP32-C3-DevKitC-02 board features Bluetooth 5, which supports Bluetooth extended advertisements. To send/receive such broadcast messages (Msg_{req}, Msg_{resp}), we use the `NimBLE` library. Once the boot sequence completes, the network module initializes and begins scanning for Msg_{req}-s using `ble_gap_disc()`. Upon receiving Msg_{req}, the module forwards it to the main board (NXP board), using `uart_write_bytes()`. When the NXP board replies with Msg_{resp}, the network module broadcasts out Msg_{resp} to U_{dev}, using `ble_gap_ext_ adv_start()`. The network module (re)transmits Msg_{resp} every 30ms for 300ms to ensure reliable delivery. These timing parameters are configurable.

DB – PAISA App on U_{dev}: DB-PAISA app is implemented as an Android app on U_{dev}, using Android Studio Electric Eel with API level 33. For a Bluetooth scan and broadcast, the app requires a few permissions: `BLUETOOTH_SCAN`, `BLUETOOTH_ADVERTISE`, and `ACCESS_FINE_LOCATION`, which are reflected in 'AndroidManifest.xml' file.

The app uses `android.bluetooth.le` libraries for Bluetooth extended advertisements. When the user clicks the "Scan Devices" button, the app broadcasts Msg_{req} using `startAdvertisingSet()` and waits with a scan for Msg_{resp}-s using `startScan()`. Msg_{resp} is identified by containing $ID_{\text{DB-PAISA}}$ (`"DP-RES"`). The app parses Msg_{resp} and fetches $Manifest_{IoT}$ via URL_{Man} using `getInputStream()` in the `java.net` library. Then, the app verifies signatures in $Manifest_{IoT}$ and Msg_{resp} with pk_{Mfr} and pk_{IoT}, using `verify()` from the `java.security` library. Finally, it displays device information details (from $Manifest_{IoT}$) and Att_{report} (from Msg_{resp}) on U_{dev}, as shown in Figure 5.

6 Evaluation

DB-PAISA security is addressed in Section 6.1. For the sake of a fair comparison, since PAISA was originally implemented using WiFi, we adapt it to Bluetooth. We measure runtime overhead on I_{dev} and U_{dev} across 50 iterations. Response on I_{dev} takes 233ms, consistent with the time PAISA announcement takes. We use this value to discuss performance overhead and energy consumption in

Sections 6.2 and 6.4, respectively. We also empirically measure the DB-PAISA-imposed performance penalty of SW_{dev} with varying parameters in Section 6.2. Runtime overhead on U_{dev} and overall network traffic overhead are evaluated in Sections 6.3 and 6.5.

6.1 Security Analysis

We now consider security guarantees of DB-PAISA against $\mathcal{A}$dv defined in Section 3.2.

I_{dev} Compromise: TZ-M guarantees that the TCB cannot be manipulated by $\mathcal{A}$dv. Also, due to the highest priority of UART and timer interrupts, I_{dev} is guaranteed to perform DB-PAISA tasks (i.e., receiving, processing, and replying to Msg_{req}-s) even with malware presence in TZ-M Normal world. Furthermore, Att_{report} contained in Msg_{resp} reflects the recent state of SW_{dev} which allows U_{dev}-s to detect compromised I_{dev}-s.

Forged Msg_{resp}: $\mathcal{A}$dv cannot generate a valid Msg_{resp} unless sk_{IoT} is leaked or the public key algorithm used for digital signature computation is broken.

DoS Attacks on I_{dev} can be launched by (1) malware on I_{dev} or (2) other malicious devices over the network. Since timer ISR and UART ISR are configured as secure, the former can be addressed with $SW_{DB-PAISA}$'s exclusive control over the network interface. The latter requires the physical presence of $\mathcal{A}$dv within Bluetooth range of I_{dev}, making this type of attack harder. This is partially addressed with the lazy-response approach as discussed in Sections 3.4 and 4.2. Other mitigation techniques are outlined in Section 7.2.

Replay Attacks on U_{dev}: U_{dev} can readily detect replay attacks by checking whether N_{usr} from an outstanding Msg_{req} is included in corresponding Msg_{resp}. Such Msg_{resp}-s can be simply discarded.

Physical Attacks on I_{dev}: TZ-M offers secure storage to store secrets and secure boot to thwart non-invasive physical attacks.

6.2 I_{dev} Runtime Overhead

To measure runtime overhead on I_{dev}, we use U_{Busy} to denote CPU usage for DB-PAISA trusted software ($SW_{DB-PAISA}$), computed as: $U_{Busy} := \frac{time_{DB-PAISA}}{time_{Normal}+time_{DB-PAISA}}$, where $time_{Normal}$ is time used for normal device operation (SW_{dev}) and $time_{DB-PAISA}$ is time used by $SW_{DB-PAISA}$. For the sake of simplicity, despite the presence of periodic attestation every $T_{\mathcal{A}tt}$, only Msg_{resp} generation overhead is considered in the experiment because attestation time is almost negligible ($\approx$ 1ms) compared to its interval ($T_{\mathcal{A}tt}$), which can be quite long, e.g., 1 hour. Also, the signing operation (231ms out of 233ms) dominates Msg_{resp} generation. Therefore, Msg_{resp} generation takes (almost) constant time even with a barrage of Msg_{req}-s within one $T_{\mathcal{G}en}$ in our lazy-response design.

In PAISA *Push* model, U_{Busy} can be represented as: $U_{Busy} := \frac{time_{Ann}}{T_{\mathcal{A}nn}}$, where $time_{Ann}$ is time to generate an announcement and $T_{\mathcal{A}nn}$ is inter-announcement interval. Since $time_{Ann}$ is constant ($\approx$ 232ms), U_{Busy} is a function of configurable $T_{\mathcal{A}nn}$.

Similarly, in DB-PAISA, $U_{Busy} := \frac{time_{Res}}{T_{\mathcal{R}eq}}$, where $time_{Res}$ is time to perform Response step and $T_{\mathcal{R}eq}$ is average time between two consecutive Msg_{req}-s. Also, $time_{Res}$ remains constant, identical to $time_{Ann}$. Thus, U_{Busy} is a function of $T_{\mathcal{R}eq}$. In the worst case, when I_{dev} continuously receives Msg_{req}-s, $T_{\mathcal{R}eq} = T_{\mathcal{G}en}$.

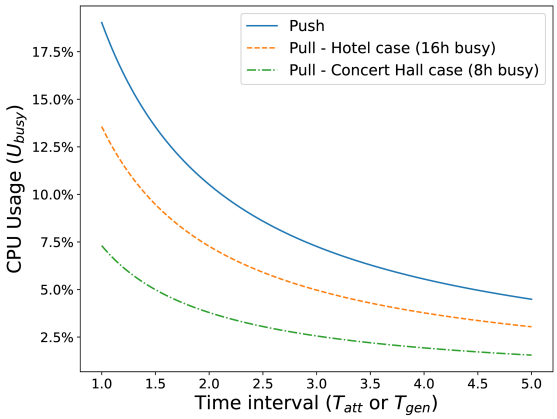


Figure 7: CPU Usage (U_{Busy}) in (DB-)PAISA *Push* and *Pull* models (Worst Case: Continuous Msg_{req}-s in *Pull*)

Time (s) ($T_{\mathcal{A}nn}/T_{\mathcal{G}en}$)	*Push*	*Pull* (with $T_{\mathcal{R}eq}$)			
		1 s	5 s	10 s	30 s
1	19.03%	5.58%	1.95%	1.09%	0.41%
2	10.51%	3.80%	1.68%	1.00%	0.40%
3	7.26%	2.88%	1.48%	0.93%	0.39%
4	5.55%	2.33%	1.32%	0.86%	0.38%
5	4.49%	1.95%	1.19%	0.81%	0.37%

Table 1: CPU Usage (U_{Busy}) of *Push* and *Pull* in Hotel Scenario (No Msg_{req} for $T_{\mathcal{R}eq}$ in *Pull*)

We consider two types of deployment settings for DB-PAISA-enabled I_{dev}-s: (1) *a concert hall*, which is crowded (and expected to have a lot of Msg_{req}-s) for 8 hours a day, and (2) *a hotel*, which is crowded for 16 hours a day. For a fair evaluation, we simply assume 10 Msg_{req}-s per hour to be received by I_{dev} when the location is not crowded. Figure 7 shows the runtime overhead comparison between *Push* and *Pull* in the worst case, with different scenarios. In the concert hall scenario, DB-PAISA reduces U_{Busy} by $\approx$ 30%, and in the hotel scenario, it decreases by $\approx$ 65%.

Runtime overhead can be further reduced if I_{dev} does not consistently receive Msg_{req}-s during the crowded time block ($T_{\mathcal{R}eq} > T_{\mathcal{G}en}$). This can occur if users are already informed about nearby devices (no new users are around) or if Msg_{req}-s are clustered at specific times. As shown in Table 1, in the hotel setting, U_{Busy} in DB-PAISA is significantly reduced compared to the *Push* model. If I_{dev} receives Msg_{req} every 10s, DB-PAISA U_{Busy} decreases by at least 70% compared to *Push*.

NOTE1: On single-core IoT devices (with TZ-M), only Secure world or Normal world can run at any given time. Therefore, DB-PAISA execution blocks normal functionality of SW_{dev} running in Normal world. While this is not an issue on multi-core devices, DB-PAISA execution still incurs certain runtime overhead, including the relatively expensive signing operation.

NOTE2: Listening for Msg_{req}-s does not interfere with normal device functionality, since the network modem is typically separate from the primary CPU core(s). However, if SW_{dev} attempts to send any packets while DB-PAISA is running, SW_{dev} will hang until DB-PAISA execution completes. Similarly, if an external entity (e.g., a server or a digital twin) attempts to communicate with SW_{dev} while

DB-PAISA is running, its packets might be dropped and would need to be retransmitted.

Empirical Evaluation: Runtime overhead incurred by DB-PAISA execution (and interruption of SW_{dev}) is measured by comparing the runtime of a task with and without DB-PAISA. This experiment is conducted with the sample application, temperature sensor software. Furthermore, DB-PAISA invocation (and therefore normal operation interruption) depends on variables, such as $\mathsf{T}_{\mathcal{R}\mathsf{eq}}$ and $\mathsf{T}_{\mathcal{G}\mathsf{en}}$. We also measure how changing these parameters, and thus the frequency of DB-PAISA invocation, impacts the runtime of SW_{dev}.

For a fair comparison with PAISA, we first evaluate runtime overhead without the lazy-response mechanism (i.e., $\mathsf{T}_{\mathcal{G}\mathsf{en}} = 0$). Recall that SW_{dev}, a temperature sensor software, reads temperature data and sends it to the server every 5s. We measure DB-PAISA overhead added to this task. Table 2 details the runtime overhead for varying $\mathsf{T}_{\mathcal{R}\mathsf{eq}}$-s.

In DB-PAISA, if $\mathsf{I}_{\mathsf{dev}}$ receives a $\mathsf{Msg}_{\mathsf{req}}$ every 1s (i.e., $\mathsf{T}_{\mathcal{R}\mathsf{eq}} = 1$s), SW_{dev} would suffer from a significant delay of 2.13s (42.55%), on average. This is mainly because $SW_{\mathsf{DB-PAISA}}$ executes 5 times, on average, during SW_{dev}'s interval of 5s. Each $SW_{\mathsf{DB-PAISA}}$ execution consumes $\approx$ 233ms while stopping `SysTick`, which is used as the timer in freeRTOS. In other words, tasks in Normal world are unaware of $SW_{\mathsf{DB-PAISA}}$ execution in Secure world, leading to delayed execution of Normal world tasks. Unsurprisingly, as $\mathsf{T}_{\mathcal{R}\mathsf{eq}}$ grows (i.e., infrequent $\mathsf{Msg}_{\mathsf{req}}$-s from users), the overhead on SW_{dev} sharply decreases. For example, SW_{dev} overhead is 0.56s (11.13%) when $\mathsf{T}_{\mathcal{R}\mathsf{eq}} = 3$s, and 0.22s (4.43%) when $\mathsf{T}_{\mathcal{R}\mathsf{eq}} = 7$s.

Note that $\mathsf{T}_{\mathcal{R}\mathsf{eq}}$ in DB-PAISA is equivalent to $\mathsf{T}_{\mathcal{A}\mathsf{nn}}$ in PAISA when $\mathsf{T}_{\mathcal{G}\mathsf{en}} = 0$. For both, PAISA's Annoucnement and DB-PAISA's $\mathcal{G}$en procedures are executed at those intervals. Both sign and generate the message containing $\mathsf{I}_{\mathsf{dev}}$ device information. In PAISA, $\mathsf{T}_{\mathcal{A}\mathsf{nn}}$ is fixed and configured at provisioning time. In contrast, $\mathsf{T}_{\mathcal{R}\mathsf{eq}}$ in DB-PAISA depends on the environment. When there are no new users sending $\mathsf{Msg}_{\mathsf{req}}$-s, SW_{dev} runs with no interference. Therefore, DB-PAISA incurs lower runtime overhead than PAISA, making it better suited for $\mathsf{I}_{\mathsf{dev}}$-s in less crowded settings.

Table 3 shows DB-PAISA overhead with varied $\mathsf{T}_{\mathcal{R}\mathsf{eq}}$-s and $\mathsf{T}_{\mathcal{G}\mathsf{en}}$-s. As mentioned in Section 5, if there are multiple $\mathsf{Msg}_{\mathsf{req}}$-s in a given $\mathsf{T}_{\mathcal{G}\mathsf{en}}$, $SW_{\mathsf{DB-PAISA}}$ receives them, appends their nonces to $\mathsf{Pool}_{\mathsf{N}}$, and generates a single $\mathsf{Msg}_{\mathsf{resp}}$ to respond to all at the end of $\mathsf{T}_{\mathcal{G}\mathsf{en}}$. For example, when $\mathsf{T}_{\mathcal{G}\mathsf{en}} = 1$s, 13-15 $\mathsf{Msg}_{\mathsf{req}}$-s are collectively handled by one $\mathsf{Msg}_{\mathsf{resp}}$ on average with 0.1 $\mathsf{T}_{\mathcal{R}\mathsf{eq}}$. This incurs SW_{dev} delay of 1.43s (28.59%). Moreover, with the same $\mathsf{T}_{\mathcal{R}\mathsf{eq}} = 1$s, $\mathsf{T}_{\mathcal{G}\mathsf{en}}$ significantly impacts the overhead: 2.13s (42.55%) for $\mathsf{T}_{\mathcal{G}\mathsf{en}} = 0$s, 0.88s (17.64%) for $\mathsf{T}_{\mathcal{G}\mathsf{en}} = 1$s, and 0.26s (5.15%) for $\mathsf{T}_{\mathcal{G}\mathsf{en}} = 5$s.

As shown in Table 5, a user obtains device information in 3.57s, on average. Thus, the result can be shown to the user within 10s when $\mathsf{T}_{\mathcal{G}\mathsf{en}} = 5$s. A too-long $\mathsf{T}_{\mathcal{G}\mathsf{en}}$ would lead to users having a false sense of privacy. It is because they may leave the place before getting $\mathsf{Msg}_{\mathsf{resp}}$-s. Then, they would think that there are no nearby $\mathsf{I}_{\mathsf{dev}}$-s. To avoid such issues, $\mathsf{T}_{\mathcal{G}\mathsf{en}}$ must be configured reasonably by compliant manufacturers. This is further discussed in Section 7.

Recall that the network peripheral is configured as secure. SW_{dev} invokes an NSC function to send data to the remote server. The overhead of the NSC function call primarily stems from context-switching between Secure and Normal worlds. To measure this overhead, the number of cycles was measured before and after calling the NSC function. We create a mock function that executes the same task as the NSC function in Normal world, and measure the number of cycles to execute the mock function. The NSC function call requires 519 cycles, while the mock function takes 392 cycles. Thus, the overhead of calling an NSC function is only 127 cycles, corresponding to $< 1\mu$s on $\mathsf{I}_{\mathsf{dev}}$ running at 150MHz.

6.3 $\mathsf{U}_{\mathsf{dev}}$ Runtime Overhead

Table 5 shows the latency of each step in the DB-PAISA app when $\mathsf{T}_{\mathcal{G}\mathsf{en}} = 0$s. After sending $\mathsf{Msg}_{\mathsf{req}}$, it takes $\approx$ 2.2s, on average, to receive $\mathsf{Msg}_{\mathsf{resp}}$. In the worst case, the DB-PAISA app would wait $\approx$ 4.95s. This overhead mostly stems from network communication delay between $\mathsf{U}_{\mathsf{dev}}$ and $\mathsf{I}_{\mathsf{dev}}$. In both PAISA and DB-PAISA, processing $\mathsf{Msg}_{\mathsf{resp}}$ takes $\approx$ 1.3s, i.e., the time between receiving $\mathsf{Msg}_{\mathsf{resp}}$ and displaying device information on $\mathsf{U}_{\mathsf{dev}}$ screen. This latency is primarily due to the fetching of $\mathsf{Manifest}_{\mathsf{IoT}}$ from $\mathsf{URL}_{\mathsf{Man}}$.

In DB-PAISA, the app needs to wait for $\approx$ 3.5s to receive $\mathsf{Msg}_{\mathsf{resp}}$. In contrast, in *Push* model, the app simply listens for device announcements. It displays $\mathsf{I}_{\mathsf{dev}}$ details much faster, after $\approx$ 1.3s.

6.4 $\mathsf{I}_{\mathsf{dev}}$ Energy Consumption

On the NXP board representing $\mathsf{I}_{\mathsf{dev}}$, we measure the current by observing the voltage drop over a 2.43ohm resistor via Pinout 12. However, the network module (ESP) does not support power consumption measurements. Hence, our ESP board's current measurement relies on the energy estimation specified in the official documentation [48]. Transmission duration is assumed to be 300ms with 30ms intervals for reliable reception on $\mathsf{U}_{\mathsf{dev}}$, i.e., 10 transmissions per $\mathsf{Msg}_{\mathsf{req}}$.

Note that the current on the ESP board is substantially higher than on the NXP board because BLE on the ESP board drains quite a lot of energy: 97.5mA for receiving and 130mA for transmitting, on average. Also, the ESP board consumes more energy than mainstream BLE technology (Nordic BLE [75]) because it is a standalone board running a real-time operating system. BLE energy consumption can be lowered by integrating BLE into the device.

Table 4 shows energy consumption on NXP and ESP boards with *Push* and *Pull* models. Power consumption on the NXP board remains almost the same in both. Since the *Pull* model continuously listens for $\mathsf{Msg}_{\mathsf{req}}$-s on the ESP board, it results in constant high power consumption. However, energy consumed in the *Push* model goes down significantly when $\mathsf{T}_{\mathcal{A}\mathsf{nn}}$ increases while making users wait longer to receive $\mathsf{Msg}_{\mathsf{resp}}$-s. Consequently, as expected, the *Push* model is more energy efficient. Nevertheless, the gap in power consumption of *Push* and *Pull* models can be reduced when (1) BLE is integrated into $\mathsf{I}_{\mathsf{dev}}$ as mentioned above, and (2) $\mathsf{T}_{\mathcal{A}\mathsf{nn}}$ is small to minimize the latency to get device information from the user perspective.

The preceding analysis shows a clear trade-off between performance and energy overheads. See details in Section 7.1.

$T_{\mathcal{A}nn}/T_{\mathcal{R}eq}$	1s	2s	3s	4s	5s	6s	7s	8s	9s	10s
Mean (s)	2.13 (42.55%)	0.89 (17.70%)	0.56 (11.13%)	0.40 (8.08%)	0.32 (6.39%)	0.26 (5.30%)	0.22 (4.43%)	0.20 (3.95%)	0.17 (3.44%)	0.15 (3.08%)
Std (s)	0.10	0.06	0.11	0.14	0.07	0.10	0.14	0.15	0.15	0.15
Min (s)	2.06	0.59	0.29	0.29	0.29	0.00	0.00	0.00	0.00	0.00
Max (s)	2.40	0.92	0.63	0.63	0.61	0.36	0.33	0.33	0.33	0.32
Median (s)	2.09	0.90	0.59	0.30	0.30	0.30	0.29	0.29	0.29	0.29

Table 2: I_{dev} Runtime Overhead with Varying $T_{\mathcal{A}nn}$ (in PAISA) and $T_{\mathcal{R}eq}$ (in DB-PAISA)

$T_{\mathcal{G}en}$	1 s							
$T_{\mathcal{R}eq}$	**0.1s**	**0.3s**	**0.5s**	**1s**	**2s**	**3s**	**4s**	**5s**
Mean (s)	1.43 (28.59%)	1.25 (25.04%)	1.25 (24.99%)	0.88 (17.64%)	0.87 (17.47%)	0.56 (11.17%)	0.40 (7.95%)	0.32 (6.40%)
Std (s)	0.13	0.12	0.11	0.08	0.08	0.11	0.14	0.07
$T_{\mathcal{G}en}$	2 s							
$T_{\mathcal{R}eq}$	**0.1s**	**0.3s**	**0.5s**	**1s**	**2s**	**3s**	**4s**	**5s**
Mean (s)	0.73 (14.64%)	0.71 (14.12%)	0.68 (13.53%)	0.55 (10.97%)	0.40 (8.04%)	0.55 (10.95%)	0.40 (8.04%)	0.32 (6.34%)
Std (s)	0.13	0.12	0.11	0.08	0.08	0.11	0.14	0.07
$T_{\mathcal{G}en}$	3 s							
$T_{\mathcal{R}eq}$	**0.1s**	**0.3s**	**0.5s**	**1s**	**2s**	**3s**	**4s**	**5s**
Mean (s)	0.48 (9.65%)	0.47 (9.39%)	0.46 (9.29%)	0.40 (8.06%)	0.40 (8.10%)	0.26 (5.24%)	0.41 (8.20%)	0.32 (6.37%)
Std (s)	0.15	0.15	0.15	0.14	0.14	0.10	0.15	0.07
$T_{\mathcal{G}en}$	4 s							
$T_{\mathcal{R}eq}$	**0.1s**	**0.3s**	**0.5s**	**1s**	**2s**	**3s**	**4s**	**5s**
Mean (s)	0.36 (7.30%)	0.35 (7.06%)	0.35 (7.04%)	0.32 (6.33%)	0.26 (5.22%)	0.26 (5.11%)	0.20 (3.93%)	0.31 (6.24%)
Std (s)	0.13	0.12	0.12	0.07	0.10	0.11	0.15	0.09
$T_{\mathcal{G}en}$	5 s							
$T_{\mathcal{R}eq}$	**0.1s**	**0.3s**	**0.5s**	**1s**	**2s**	**3s**	**4s**	**5s**
Mean (s)	0.29 (5.86%)	0.29 (5.88%)	0.29 (5.89%)	0.26 (5.15%)	0.26 (5.24%)	0.26 (5.26%)	0.20 (3.91%)	0.16 (3.19%)
Std (s)	0.05	0.04	0.04	0.11	0.10	0.10	0.14	0.15

Table 3: I_{dev} Runtime Overhead with Varying $T_{\mathcal{G}en}$ in DB-PAISA

Time (s) ($T_{\mathcal{A}nn}/T_{\mathcal{G}en}$)	**NXP (mA)**		**ESP (mA)**		I_{dev} **(mA) (NXP+ESP)**	
	Push	*Pull*	*Push*	*Pull*	*Push*	*Pull*
1	8.46	9.12	60.35	100.75	68.81	109.87
2	8.46	8.78	45.43	99.13	53.88	107.91
3	8.45	8.67	40.45	98.59	48.90	107.26
4	8.45	8.62	37.96	98.31	46.42	106.93
5	8.45	8.58	36.47	98.15	44.92	106.74

Table 4: I_{dev} Energy Consumption of *Push* and *Pull*

6.5 Network Overhead

In PAISA *Push* model, announcement size is 128 bytes, while in the *Pull* model, Msg_{req} and Msg_{resp} are 18 and 100 bytes, respectively Since both *Push* and *Pull* models use the same Bluetooth packet header, we compare only the payload size. Figure 8 shows bits per second (bps) of *Push* and *Pull* I_{dev}-s, respectively, with varying intervals. Bandwidth overhead is reduced by 40.1% in the hotel scenario and 70.5% in the concert hall scenario. However, the *Pull* model raises another concern; packet size scales with the number of $U_{dev}$$Msg_{req}$-s within one $T_{\mathcal{G}en}$, leading to larger Msg_{resp}.

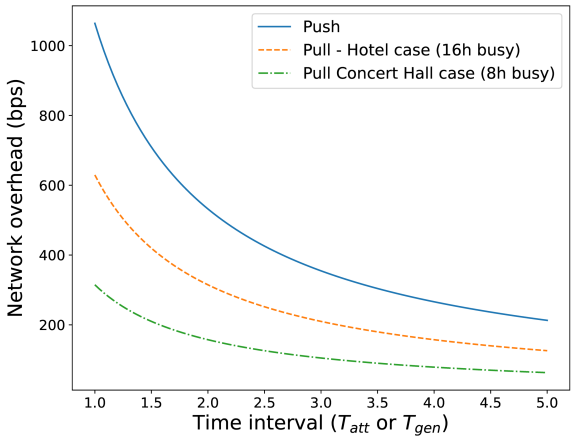


Figure 8: Network Traffic Overhead

To alleviate this problem, *Push* and *Pull* models can be blended to dynamically switch between *Pull* and *Push* protocols depending on the rate of Msg_{req}-s. For instance, given a high influx of Msg_{req}-s, *Pull* model would switch to *Push* model. This mitigation is further discussed in Section 7.1.

7 Discussion & Limitations

7.1 *Push vs Pull* Tradeoffs & Blending

Push and *Pull* models have trade-offs in terms of energy consumption as well as performance overheads. Also, network bandwidth utilization in both models depends on the number of I_{dev}-s and U_{dev}-s in an area. The better-suited model is contingent on the deployment use case. However, it is quite straightforward to blend them and support on-the-fly switching between them.

For example, suppose that I_{dev} is using *Pull* mode. Once it starts receiving an influx of Msg_{req}-s (say, over a certain threshold) in rapid succession, it switches to *Push* mode. I_{dev} stays in *Push* mode for a certain period (configurable by Mfr) and then switches back to *Pull* mode. In other words, when the setting experiences an influx of new users, *Push* mode works better, while *Pull* is better when there are fewer and/or infrequent new users. Note that U_{dev} need not be aware of whether I_{dev} is running in *Push* model or *Pull* model. It can collect both announcement messages (from *Push* devices) and Msg_{resp}-s (from *Pull* devices) once it sends a single Msg_{req}. This heuristic is quite easy to implement, in part because

Model	Event	Time (ms)	Std Dev (ms)	Min (ms)	Max (ms)	Median (ms)
Push	Scan - Display	1324.56	405.81	794	3085	1279
Pull	Request - Reception	2237.18	526.17	993	3721	2215.5
	Reception - Display	1328.60	205.99	993	2244	1305.0
	Total	3565.62	519.04	2611	4952	3486.5

Table 5: U_{dev} Runtime Overhead of *Push* and *Pull*

announcement messages in *Push* and Msg_{resp}-s in *Pull* are almost identical format-wise.

7.2 Msg_{req} DoS Mitigation

As discussed in Section 3.4, the lazy-response approach does not fully mitigate DoS attacks: once $Pool_N$ size reaches $|Pool_N|_{Max}$, I_{dev} composes and signs Msg_{resp} without waiting for $T_{\mathcal{G}en}$ to elapse. Thus, if it is subject to a high volume of incoming Msg_{req}-s, I_{dev} can be essentially choked.

Intuitively, public key-based signing can be replaced with highly efficient symmetric MACs. However, this would require the existence and involvement of a trusted third party (e.g., Mfr) to verify such MACs for U_{dev}. (Recall that I_{dev}-s and U_{dev}-s have no prior security context.) When I_{dev} sends Msg_{resp} with an HMAC, U_{dev} can ask Mfr, via a secure channel, to verify the HMAC. We consider this approach undesirable as it involves an additional (and trusted) third party, which can become a bottleneck and would represent a highly attractive attack target.

7.3 Localization

Localizing I_{dev}-s can help considerably improve user awareness and IoT device transparency. Although PAISA and DB-PAISA discover nearby devices, none of them truly and reliably localize them.

Indoor localization using WiFi or Bluetooth poses a challenge. Localization typically relies on signal strength (e.g., RSSI), timing information (e.g., time-of-flight, time-of-arrival, and round-trip time), or directional information (e.g., angle-of-arrival). Signal strength-based localization is relatively imprecise and vulnerable to spoofing attacks [17, 58, 102]. Meanwhile, timing-based localization is vulnerable to distance enlargement attacks, whereby the device falsely claims to be farther than it really is [91].

High-accuracy localization relies on external infrastructure or multiple antennas to triangulate location. Recently, Bluetooth 5.1 [20] introduced a new feature called Direction Finding, which supports highly accurate localization. This requires devices to have multiple antennas to triangulate the locations of other devices. The emerging Ultra-Wideband (UWB) technology [4] is promising for indoor localization due to its high precision and resistance to interference. Nonetheless, these advanced localization technologies are not yet widely available on commodity IoT devices.

7.4 TZ-M Alternatives

DB-PAISA relies on TZ-M to provide guaranteed execution. It can also be deployed on devices with a Root-of-Trust (RoT) that supports a secure timer, secure network peripherals, secure storage, and task prioritization. As one of the examples of low-end, GAROTA [7], an active RoT solution for small embedded devices, supports a secure timer, UART, and GPIO. It customizes hardware to offer these security features. Alternatively, RISC-V devices can execute DB-PAISA using MultiZone [54]. It supports Physical Memory Protection (PMP) to isolate secure execution from non-secure tasks. In addition, I/O PMP (IOPMP) allows configuring peripherals as secure.

7.5 False Sense of Privacy with Large $T_{\mathcal{G}en}$

Fast response time of Msg_{resp} through small $T_{\mathcal{G}en}$ is relevant to users who linger long enough to receive I_{dev} Msg_{resp}-s. Users who run, walk, or cycle by I_{dev}'s location (i.e., move out of I_{dev}'s WiFi/BT broadcast range too quickly) may miss I_{dev} Msg_{resp}-s. We expect privacy-concerned users to remain in a given location long enough to receive Msg_{resp}-s; $T_{\mathcal{G}en}$ should be configured by manufacturers to be at most a few seconds to retain users' attention.

8 Related Work

Device discovery: In addition to recent work described in Section 1.1, some research [31, 72, 78] introduces a registration-based approach to enhance the transparency of device presence and capabilities. The technique provides a scalable registry-based privacy infrastructure, where IoT device owners publish information about their IoT devices and their capabilities in online registries accessible to other users. This setup assists users in identifying nearby IoT resources and selectively informs users about the data privacy practices of these resources. Furthermore, IoT-PPA [31] facilitates the discovery of user-configurable settings for IoT resources (e.g., opt-in, opt-out, data erasure), enabling privacy assistants to help users align their IoT experience with their privacy preferences.

Hidden IoT device detection: There is also a large body of research focused on discovering hidden devices through either emitting/measuring signals with specialized hardware or analyzing network traffic.

[52, 70, 81, 83, 93] employ specialized hardware to detect hidden IoT devices using various technologies, including NLJD sensors [81], millimeter waves [70], software-defined radio (SDR) signals [52], and time-of-flight sensor data [83]. While these methods are effective in detecting the presence of IoT devices, they are incapable of identifying them. Moreover, this technique requires users to possess specialized tools to discover IoT devices.

Network traffic analysis involves examining patterns and packets to deduce the presence of devices. Prior research used this approach to identify devices [77, 88], detect active sharing of sensed information [53], and localize devices [88]. In particular, [90] achieves these objectives by establishing causality between patterns in observable

wireless traffic. However, network analysis is less effective when IoT devices communicate infrequently. Also, malicious $\mathcal{A}$dv can evade detection by not communicating when users are detected nearby or manipulating its network traffic [56].

IoT privacy: Another line of IoT privacy research involves suggesting privacy labels designed for IoT devices that inform users about security and privacy concerns. This entails understanding user concerns, incorporating expert recommendations, and offering the privacy labels on devices to aid purchasing decisions [43, 49]. Another related research direction explored users' perceptions of risk and their willingness to pay for security/privacy features [44, 45].

Automated privacy assistants and consent platforms [27, 55, 64, 71, 103] have been proposed to assist users in managing privacy settings for IoT devices that they encounter, especially, in scenarios where the volume of notifications might overwhelm users.

9 Conclusion

This work presents DB-PAISA, a novel approach to enhance privacy and security in IoT ecosystems through a *Pull*-based discovery mechanism. Unlike previous *Push*-based models, DB-PAISA minimizes unnecessary network traffic and interference with device operations by enabling IoT devices to respond to explicit user requests. Our implementation and evaluation demonstrate DB-PAISA's practicality and efficiency, comparing runtime overheads and energy consumption with the *Push* model.

Acknowledgments

We thank PETS 2025 reviewers for their constructive feedback. This work was supported in part by funding from NSF Award SATC-1956393, NSA Awards H98230-20-1-0345 and H98230-22-1-0308, as well as a DARPA subcontract from Peraton Labs.

References

[1] Ramia Babiker Mohammed Abdelrahman, Amin Babiker A Mustafa, and Ashraf A Osman. 2015. A Comparison between IEEE 802.11 a, b, g, n and ac Standards. *IOSR Journal of Computer Engineering (IOSR-JEC)* 17, 5 (2015), 26–29.

[2] 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 2016 ACM CCS*. ACM, 743–754.

[3] Abbas Acar, Hossein Fereidooni, Tigist Abera, Amit Kumar Sikder, Markus Miettinen, Hidayet Aksu, Mauro Conti, Ahmad-Reza Sadeghi, and Selcuk Uluagac. 2020. Peek-a-boo: i see your smart home activities, even encrypted!. In *Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks* (Linz, Austria) *(WiSec '20)*. Association for Computing Machinery, New York, NY, USA, 207–218. https://doi.org/10.1145/3395351.3399421

[4] G Roberto Aiello and Gerald D Rogerson. 2003. Ultra-wideband wireless systems. *IEEE microwave magazine* 4, 2 (2003), 36–47.

[5] Airbnb. 2024. Airbnb update their policy on security cameras. https://news.airbnb.com/an-update-on-our-policy-on-security-cameras/.

[6] Rubayyi Alghamdi and Martine Bellaïche. 2023. A cascaded federated deep learning based framework for detecting wormhole attacks in IoT networks. *Comput. Secur.* 125 (2023), 103014.

[7] Esmerald Aliaj, Ivan De Oliveira Nunes, and Gene Tsudik. 2022. {GAROTA}: generalized active {Root-Of-Trust} architecture (for tiny embedded devices). In *31st USENIX Security Symposium (USENIX Security 22)*. 2243–2260.

[8] Omar Alrawi, Chaz Lever, Manos Antonakakis, and Fabian Monrose. 2019. SoK: Security Evaluation of Home-Based IoT Deployments. In *2019 IEEE Symposium on Security and Privacy (SP)*. 1362–1380. https://doi.org/10.1109/SP.2019.00013

[9] Moreno Ambrosin, Mauro Conti, Riccardo Lazzeretti, Md Masoom Rabbani, and Silvio Ranise. 2020. Collective remote attestation at the Internet of Things scale: State-of-the-art and future challenges. *IEEE Communications Surveys & Tutorials* 22, 4 (2020), 2447–2461.

[10] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, Chaz Lever, Zane Ma, Joshua Mason, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, and Yi Zhou. 2017. Understanding the Mirai Botnet. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 1093–1110. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/antonakakis

[11] Jacob Arellano. 2019. Bluetooth vs. Wi-Fi for IoT: Which is Better? https://www.verytechnology.com/iot-insights/bluetooth-vs-wifi-for-iot-which-is-better.

[12] ARM. 2021. ARM Confidential Compute Aarchitecture (ARM CCS). https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture.

[13] Arm Ltd. 2009. ARM TrustZone for Cortex-M. https://www.arm.com/technologies/trustzone-for-cortex-m.

[14] Arm Ltd. 2017. Cortex-M Prototyping System. https://developer.arm.com/documentation/100112/0100.

[15] Arm Ltd. 2018. Arm TrustZone. https://www.arm.com/products/security-on-arm/trustzone.

[16] Anas M Atieh, Hazem Kaylani, Yousef Al-Abdallat, Abeer Qaderi, Luma Ghoul, Lina Jaradat, and Iman Hdairis. 2016. Performance improvement of inventory management system processes by an automated warehouse management system. *Procedia Cirp* (2016).

[17] Kevin Bauer, Damon McCoy, Eric Anderson, Markus Breitenbach, Greg Grudic, Dirk Grunwald, and Douglas Sicker. 2009. The Directional Attack on Wireless Localization -or- How to Spoof Your Location with a Tin Can. In *GLOBECOM 2009 - 2009 IEEE Global Telecommunications Conference*. 1–6. https://doi.org/10.1109/GLOCOM.2009.5425737

[18] Bitly. 2008. Bitly. https://bitly.com/.

[19] Bluetooth Special Interest Group. 2016. Bluetooth 5.0 Specification. https://www.bluetooth.com/specifications/specs/core-specification-5-0.

[20] Bluetooth Special Interest Group. 2019. Bluetooth 5.1 Specification. https://www.bluetooth.com/specifications/specs/core-specification-5-1.

[21] Mirai Botnet. 2016. Website. https://www.incapsula.com/blog/malware-analysis-mirai-ddos-botnet.html.

[22] Agnès Brelurut, David Gérault, and Pascal Lafourcade. 2015. Survey of Distance Bounding Protocols and Threats. In *Foundations and Practice of Security - 8th International Symposium, FPS*.

[23] Ismail Butun, Patrik Österberg, and Houbing Song. 2019. Security of the Internet of Things: Vulnerabilities, attacks, and countermeasures. *IEEE Communications Surveys & Tutorials* (2019).

[24] Adam Caulfield, Norrathep Rattanavipanon, and Ivan De Oliveira Nunes. 2023. {ACFA}: Secure Runtime Auditing & Guaranteed Device Healing via Active Control Flow Attestation. In *32nd USENIX Security Symposium (USENIX Security 23)*.

[25] Adam Caulfield, Norrathep Rattanavipanon, and Ivan De Oliveira Nunes. 2023. {ACFA}: Secure Runtime Auditing & Guaranteed Device Healing via Active Control Flow Attestation. In *32nd USENIX Security Symposium (USENIX Security 23)*.

[26] Mario Collotta, Giovanni Pau, Timothy Talty, and Ozan K Tonguz. 2018. Bluetooth 5: A concrete step forward toward the IoT. *IEEE Communications Magazine* 56, 7 (2018), 125–131.

[27] Jessica Colnago, Yuanyuan Feng, Tharangini Palanivel, Sarah Pearman, Megan Ung, Alessandro Acquisti, Lorrie Faith Cranor, and Norman Sadeh. 2020. Informing the design of a personalized privacy assistant for the internet of things. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. 1–13.

[28] Victor Costan, Ilia Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal hardware extensions for strong software isolation. In *25th USENIX Security Symposium (USENIX Security 16)*.

[29] NIST Cybersecurity. 2019. Submission Requirements and Evaluation Criteria for the Lightweight Cryptography Standardization Process. https://csrc.nist.gov/csrc/media/Projects/lightweight-cryptography/documents/final-lwc-submission-requirements-august2018.pdf.

[30] NIST Cybersecurity. 2022. Recommended Criteria for Cybersecurity Labeling for Consumer Internet of Things (IoT) Products. https://nvlpubs.nist.gov/nistpubs/CSWP/NIST.CSWP.02042022-2.pdf.

[31] Anupam Das, Martin Degeling, Daniel Smullen, and Norman Sadeh. 2018. Personalized privacy assistants for the internet of things: Providing users with notice and choice. *IEEE Pervasive Computing* 17, 3 (2018), 35–46.

[32] Lucas Davi, Patrick Koeberl, and Ahmad-Reza Sadeghi. 2014. Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation. In *Design Automation Conference*.

[33] Ivan De Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, Michael Steiner, and Gene Tsudik. 2019. VRASED: A Verified Hardware/Software Co-Design for Remote Attestation. In *USENIX Security*.

[34] Ivan De Oliveira Nunes, Seoyeon Hwang, Sashidhar Jakkamsetti, and Gene Tsudik. 2022. Privacy-from-Birth: Protecting Sensed Data from Malicious Sensors with VERSA. In *43rd IEEE Symposium on Security and Privacy, SP 2022*.

[35] Ivan De Oliveira Nunes, Sashidhar Jakkamsetti, Youngil Kim, and Gene Tsudik. 2022. CASU: Compromise Avoidance via Secure Update for Low-End Embedded Systems. In *41st IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*.

[36] Ivan De Oliveira Nunes, Sashidhar Jakkamsetti, and Gene Tsudik. 2021. Tiny-CFA: Minimalistic Control-Flow Attestation Using Verified Proofs of Execution. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*.

[37] Departmemt for Digital, Culture, Media, and Sport, The UK. 2018. Code of Practice for Consumer IoT Security. https://assets.publishing.service.gov.uk/government/uploads/system/uploads/attachment_data/file/971440/Code_of_Practice_for_Consumer_IoT_Security_October_2018_V2.pdf.

[38] Department of Home Affairs, Australia. 2020. Securing the Internet of Things for Consumers. https://www.homeaffairs.gov.au/reports-and-pubs/files/code-of-practice.pdf.

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

[40] D. Dolev and A. Yao. 1983. On the security of public key protocols. *IEEE Transactions on Information Theory* (1983).

[41] Ashutosh Dhar Dwivedi, Gautam Srivastava, Shalini Dhar, and Rajani Singh. 2019. A decentralized privacy-preserving healthcare blockchain for IoT. *Sensors* (2019).

[42] Karim Eldefrawy, Gene Tsudik, Aurélien Francillon, and Daniele Perito. 2012. SMART: Secure and Minimal Architecture for (Establishing Dynamic) Root of Trust. In *NDSS*.

[43] Pardis Emami-Naeini, Yuvraj Agarwal, Lorrie Faith Cranor, and Hanan Hibshi. 2020. Ask the Experts: What Should Be on an IoT Privacy and Security Label?. In *2020 IEEE Symposium on Security and Privacy (SP)*. 447–464. https://doi.org/10.1109/SP40000.2020.00043

[44] Pardis Emami-Naeini, Janarth Dheenadhayalan, Yuvraj Agarwal, and Lorrie Faith Cranor. 2021. Which privacy and security attributes most impact consumers' risk perception and willingness to purchase IoT devices?. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 519–536.

[45] Pardis Emami-Naeini, Janarth Dheenadhayalan, Yuvraj Agarwal, and Lorrie Faith Cranor. 2023. Are Consumers Willing to Pay for Security and Privacy of IoT Devices?. In *In Proceedings of the 32nd USENIX Security Symposium*.

[46] Ericsson. 2024. Realizing smart manufacturing through IoT. https://www.ericsson.com/en/reports-and-papers/mobility-report/articles/realizing-smart-manufact-iot.

[47] Espressif Systems. 2016. ESP32-C3-DevKitC-02. https://docs.espressif.com/projects/esp-idf/en/latest/esp32c3/hw-reference/esp32c3/user-guide-devkitc-02.html.

[48] Espressif Systems. 2024. ESP32 Series Datasheet. https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf.

[49] Yuanyuan Feng, Yaxing Yao, and Norman Sadeh. 2021. A design space for privacy choices: Towards meaningful privacy control in the internet of things. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–16.

[50] Earlence Fernandes, Justin Paupore, Amir Rahmati, Daniel Simionato, Mauro Conti, and Atul Prakash. 2016. {FlowFence}: Practical data protection for emerging {IoT} application frameworks. In *25th USENIX security symposium (USENIX Security 16)*.

[51] Google. 2021. Google Pixel 6 Pro Specifications. https://www.gsmarena.com/google_pixel_6_pro-10918.php.

[52] Stefan Gvozdenovic, Johannes K Becker, John Mikulskis, and David Starobinski. 2022. Multi-Protocol IoT Network Reconnaissance. In *2022 IEEE Conference on Communications and Network Security (CNS)*. 118–126.

[53] Jeongyoon Heo, Sangwon Gil, Youngman Jung, Jinmok Kim, Donguk Kim, Woojin Park, Yongdae Kim, Kang G. Shin, and Choong-Hoon Lee. 2022. Are There Wireless Hidden Cameras Spying on Me?. In *Annual Computer Security Applications Conference, ACSAC 2022, Austin, TX, USA, December 5-9, 2022*. ACM, 714–726.

[54] HexFive. 2019. HexFive Multizone Security. https://hex-five.com/.

[55] Jason I Hong and James A Landay. 2004. An architecture for privacy-sensitive ubiquitous computing. In *Proceedings of the 2nd international conference on Mobile systems, applications, and services*. 177–189.

[56] Tao Hou, Tao Wang, Zhuo Lu, Yao Liu, and Yalin Sagduyu. 2021. IoTGAN: GAN Powered Camouflage Against Machine Learning Based IoT Device Identification. In *2021 IEEE International Symposium on Dynamic Spectrum Access Networks (DySPAN)*. 280–287. https://doi.org/10.1109/DySPAN53946.2021.9677264

[57] Yih-Chun Hu, A. Perrig, and D.B. Johnson. 2006. Wormhole attacks in wireless networks. *IEEE Journal on Selected Areas in Communications* (2006).

[58] Todd Humphreys, B.M. Ledvina, Mark Psiaki, B.W. O'Hanlon, and Kintner Jr. 2008. Assessing the spoofing threat: Development of a portable GPS civilian spoofer," in. *Proc. of the ION GNSS international technical meeting of the satellite division* 55 (01 2008).

[59] IEEE Standard Association. 2009. IEEE 802.11n-2009 Standard. https://standards.ieee.org/ieee/802.11n/3952/.

[60] Inman. 2019. More than 1 in 10 airbnb guests have found hidden cameras: Survey. https://www.inman.com/2019/06/07/more-than-1-in-10-airbnb-guest-have-found-cameras-in-rentals-survey.

[61] Intel. 2015. Intel Software Guard Extensions (Intel SGX). https://software.intel.com/en-us/sgx.

[62] Shalabh Jain, Tuan Ta, and John S. Baras. 2012. Wormhole detection using channel characteristics. In *Proceedings of IEEE International Conference on Communications, ICC 2012, Ottawa, ON, Canada, June 10-15, 2012*. IEEE, 6699–6704.

[63] Sashidhar Jakkamsetti, Youngil Kim, and Gene Tsudik. 2023. Caveat (IoT) Emptor: Towards Transparency of IoT Device Presence. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 1347–1361.

[64] Hongxia Jin, Gokay Saldamli, Richard Chow, and Bart P. Knijnenburg. 2013. Recommendations-based location privacy control. In *2013 IEEE International Conference on Pervasive Computing and Communications Workshops*.

[65] Pallavi Kaliyar, Wafa Ben Jaballah, Mauro Conti, and Chhagan Lal. 2020. LiDL: Localization with early detection of sybil and wormhole attacks in IoT Networks. *Comput. Secur.* 94 (2020), 101849.

[66] Berkay Kaplan, Israel J Lopez-Toledo, Carl Gunter, and Jingyu Qian. 2023. A Tagging Solution to Discover IoT Devices in Apartments. In *Proceedings of the 39th Annual Computer Security Applications Conference*.

[67] Youngil Kim, Isita Bagayatkar, and Gene Tsudik. 2024. DB-PAISA Source Code. https://github.com/sprout-uci/DB-PAISA.

[68] Khairy AH Kobbacy and Yansong Liang. 1999. Towards the development of an intelligent inventory management system. *Integrated Manufacturing Systems* (1999).

[69] California Legislature. 2020. California Consumer Privacy Act of 2018 (as amended by the California Privacy Rights Act of 2020). https://www.oag.ca.gov/privacy/ccpa.

[70] Zhengxiong Li, Zhuolin Yang, Chen Song, Changzhi Li, Zhengyu Peng, and Wenyao Xu. 2018. E-Eye: Hidden Electronics Recognition through mmWave Nonlinear Effects. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, Gowri Sankar Ramachandran and Bhaskar Krishnamachari (Eds.).

[71] Xuying Meng, Suhang Wang, Kai Shu, Jundong Li, Bo Chen, Huan Liu, and Yujun Zhang. 2019. Towards privacy preserving social recommendation under personalized privacy settings. *World Wide Web* 22 (2019), 2853–2881.

[72] Chenglin Miao, Wenjun Jiang, Lu Su, Yaliang Li, Suxin Guo, Zhan Qin, Houping Xiao, Jing Gao, and Kui Ren. 2015. Cloud-enabled privacy-preserving truth discovery in crowd sensing systems. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*. 183–196.

[73] Charlie Miller and Chris Valasek. 2015. Remote exploitation of an unaltered passenger vehicle. *Black Hat USA* (2015).

[74] Job Noorman, Jo Van Bulck, Jan Tobias Mühlberg, Frank Piessens, Pieter Maene, Bart Preneel, Ingrid Verbauwhede, Johannes Götzfried, Tilo Müller, and Felix C. Freiling. 2017. Sancus 2.0: A Low-Cost Security Architecture for IoT Devices. *ACM Trans. Priv. Secur.* (2017).

[75] Nordic Semiconductor. 2012. nRF8000 Series. https://www.nordicsemi.com/Products/nRF8000-series/.

[76] NXP Semiconductors. 2020. NXP LPC55S69-EVK. https://www.nxp.com/design/software/development-software/mcuxpresso-software-and-tools-/lpcxpresso-boards/lpcxpresso55s69-development-board:LPC55S69-EVK.

[77] Jorge Ortiz, Catherine H. Crawford, and Franck Le. 2019. DeviceMien: network device behavior modeling for identifying unknown IoT devices. In *Proceedings of the International Conference on Internet of Things Design and Implementation, IoTDI 2019, Montreal, QC, Canada, April 15-18, 2019*. ACM.

[78] Primal Pappachan, Martin Degeling, Roberto Yus, Anupam Das, Sruti Bhagavatula, William Melicher, Pardis Emami Naeini, Shikun Zhang, Lujo Bauer, Alfred Kobsa, Sharad Mehrotra, Norman Sadeh, and Nalini Venkatasubramanian. 2017. Towards Privacy-Aware Smart Buildings: Capturing, Communicating, and Enforcing Privacy Policies and Preferences. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. 193–198. https://doi.org/10.1109/ICDCSW.2017.52

[79] European Parliament and Council. 2016. General Data Protection Regulation, Regulation (EU) 2016/679 (as amended). https://eur-lex.europa.eu/eli/reg/2016/679/2016-05-04.

[80] Srivaths Ravi, Anand Raghunathan, and Srimat Chakradhar. 2004. Tamper resistance mechanisms for secure embedded systems. In *17th International Conference on VLSI Design. Proceedings*. 605–611. https://doi.org/10.1109/ICVD.2004.1260985

[81] REI. 2015. Orion HX Deluxe Non-Linear Junction Detector. https://reiusa.net/nljd/orion-hx-deluxe-nljd/.

[82] RISC-V International. 2015. RISC-V. https://riscv.org/about/.

[83] Sriram Sami, Sean Rui Xiang Tan, Bangjie Sun, and Jun Han. 2021. LAPD: Hidden Spy Camera Detection Using Smartphone Time-of-Flight Sensors. In *Proceedings of the 19th ACM Conference on Embedded Networked Sensor Systems*

(Coimbra, Portugal) *(SenSys '21)*. Association for Computing Machinery, New York, NY, USA, 288–301. https://doi.org/10.1145/3485730.3485941

[84] Christoph L Schuba, Ivan V Krsul, Markus G Kuhn, Eugene H Spafford, Aurobindo Sundaram, and Diego Zamboni. 1997. Analysis of a denial of service attack on TCP. In *Proceedings. 1997 IEEE Symposium on Security and Privacy (Cat. No. 97CB36097)*. IEEE, 208–223.

[85] Arindam Sengupta, Feng Jin, Renyuan Zhang, and Siyang Cao. 2020. mm-Pose: Real-time human skeletal posture estimation using mmWave radars and CNNs. *IEEE Sensors Journal* 20, 17 (2020), 10032–10044.

[86] Amazon Web Services. 2017. FreeRTOS. https://www.freertos.org.

[87] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. 2005. Pioneer: Verifying Code Integrity and Enforcing Untampered Code Execution on Legacy Systems. *ACM SIGOPS Operating Systems Review* (December 2005).

[88] Rahul Anand Sharma, Elahe Soltanaghaei, Anthony Rowe, and Vyas Sekar. 2022. Lumos: Identifying and Localizing Diverse Hidden IoT Devices in an Unfamiliar Environment. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association.

[89] Amit Kumar Sikder, Giuseppe Petracca, Hidayet Aksu, Trent Jaeger, and A. Selcuk Uluagac. 2021. A Survey on Sensor-Based Threats and Attacks to Smart Devices and Applications. *IEEE Communications Surveys & Tutorials* 23, 2 (2021), 1125–1159. https://doi.org/10.1109/COMST.2021.3064507

[90] Akash Deep Singh, Luis Garcia, Joseph Noor, and Mani Srivastava. 2021. I Always Feel Like Somebody's Sensing Me! A Framework to Detect, Identify, and Localize Clandestine Wireless Sensors. In *30th USENIX Security Symposium*.

[91] Mridula Singh, Patrick Leu, AbdelRahman Abdou, and Srdjan Capkun. 2019. UWB-ED: Distance Enlargement Attack Detection in Ultra-Wideband. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 73–88. https://www.usenix.org/conference/usenixsecurity19/presentation/singh

[92] Tianyi Song, Ruinian Li, Bo Mei, Jiguo Yu, Xiaoshuang Xing, and Xiuzhen Cheng. 2017. A privacy preserving communication protocol for IoT applications in smart homes. *IEEE Internet of Things Journal* (2017).

[93] Spygadgets. 2015. Bug detector and hidden camera finder. https://www.spygadgets.com/collections/counter-surveillance.

[94] KENS 5 Staff and Zack Briggs. 2023. Landlord accused of recording female tenant with hidden camera. https://www.kens5.com/article/news/local/landlord-accused-of-recording-female-tenant-with-hidden-camera/273-ad6cd4ab-9f6c-4f64-975a-0a868c92b7b1.

[95] Statista. 2022. Number of Internet of Things (IoT) connected devices worldwide from 2019 to 2021, with forecasts from 2022 to 2030. https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/.

[96] Statista. 2024. Industrial IoT. https://www.statista.com/outlook/tmo/internet-of-things/industrial-iot/worldwide/.

[97] 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.

[98] Hailun Tan, Wen Hu, and Sanjay Jha. 2011. A TPM-enabled remote attestation protocol (TRAP) in wireless sensor networks. In *Proceedings of the 6th ACM workshop on Performance monitoring and measurement of heterogeneous wireless and wired networks*. ACM, 9–16.

[99] B Sai Subrahmanya Tejesh and SJAEJ Neeraja. 2018. Warehouse inventory management system using IoT and open source framework. *Alexandria engineering journal* (2018).

[100] Tim Newcomb. 2017. 7 of the World's Largest Manufacturing Plants. https://www.popularmechanics.com/technology/infrastructure/g2904/7-of-the-worlds-largest-manufacturing-plants/.

[101] TinyURL LLC. 2002. TinyURL. https://tinyurl.com/app.

[102] Nils Ole Tippenhauer, Kasper Bonne Rasmussen, Christina Pöpper, and Srdjan Čapkun. 2009. Attacks on public WLAN-based positioning systems. In *Proceedings of the 7th International Conference on Mobile Systems, Applications, and Services* (Kraków, Poland) *(MobiSys '09)*. Association for Computing Machinery, New York, NY, USA, 29–40. https://doi.org/10.1145/1555816.1555820

[103] Christine Utz, Matthias Michels, Martin Degeling, Ninja Marnau, and Ben Stock. 2023. Comparing large-scale privacy and security notifications. *Proceedings on Privacy Enhancing Technologies* (2023).

[104] Jaikumar Vijayan. 2010. Stuxnet renews power grid security concerns. https://www.computerworld.com/article/2754164/stuxnet-renews-power-grid-security-concerns.html.

[105] Meng Xu, Manuel Huber, Zhichuang Sun, Paul England, Marcus Peinado, Sangho Lee, Andrey Marochko, Dennis Mattoon, Rob Spiger, and Stefan Thom. 2019. Dominance as a New Trusted Computing Primitive for the Internet of Things. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 1415–1430.

[106] Yaxing Yao, Justin Reed Basdeo, Smirity Kaushik, and Yang Wang. 2019. Defending my castle: A co-design study of privacy mechanisms for smart homes. In *Proceedings of the 2019 chi conference on human factors in computing systems*.

[107] Bassam Zahran, Adamu Hussaini, and Aisha Ali-Gombe. 2021. IIoT-ARAS: IIoT/ICS Automated risk assessment system for prediction and prevention. In *Proceedings of the Eleventh ACM Conference on Data and Application Security and Privacy*. 305–307.

[108] Andreas Zankl, Hermann Seuschek, Gorka Irazoqui, and Berk Gulmezoglu. 2021. Side-channel attacks in the Internet of Things: threats and challenges. In *Research Anthology on Artificial Intelligence Applications in Security*. IGI Global, 2058–2090.

[109] Eric Zeng, Shrirang Mare, and Franziska Roesner. 2017. End User Security and Privacy Concerns with Smart Homes. In *Thirteenth Symposium on Usable Privacy and Security (SOUPS 2017)*. USENIX Association.

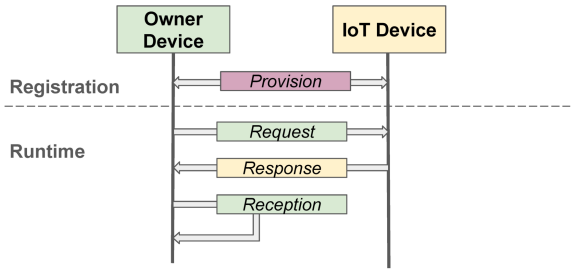


Figure 9: IM-PAISA **Overview**

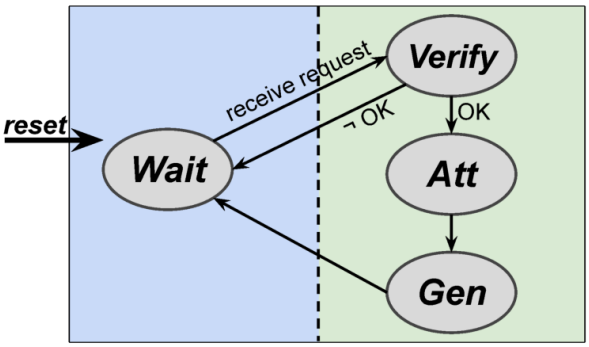


Figure 10: IM-PAISA **State Machine on** I_{dev}

A DB-PAISA **Variant** – IM-PAISA

IM-PAISA is a variant of DB-PAISA, designed for inventory management in large IIoT settings. IM-PAISA has two components: I_{dev} and O_{dev}– the owner's device authorized to solicit information from a multitude of I_{dev}-s. As shown in Figure 9, IM-PAISA also has two phases: *Registration* and *Runtime*.

***Registration* phase** occurs prior to device deployment. Each I_{dev} is securely provisioned with (1) unique secret key ($\mathcal{K}$) shared with O_{dev}, (2) its own device information, and (3) O_{dev}'s public key (pk_{owner}).

***Runtime* phase** has three steps: Request, Response, and Reception. There are three significant differences from DB-PAISA: (1) Msg_{req} is authenticated with O_{dev}'s private key (sk_{owner}), (2) Msg_{resp} is encrypted with $\mathcal{K}$, and (3) $\mathcal{A}$tt takes place upon every Msg_{req}.

A.1 IM-PAISA **Adversary model**

DoS attacks on I_{dev}-s: Similar to DB-PAISA, malware $\mathcal{A}$dv can try to deplete I_{dev} resources via software vulnerabilities. However, DoS attacks from a network $\mathcal{A}$dv are more challenging to address because of costly verification of O_{dev}'s signatures in Msg_{req}-s.

Replay attacks: A network-based $\mathcal{A}$dv can replay arbitrary messages to both O_{dev} and I_{dev}-s.

Eavesdropping: A network-based $\mathcal{A}$dv can eavesdrop on all IM-PAISA protocol messages to learn individual I_{dev} information and the number and types of deployed I_{dev}-s. $\mathcal{A}$dv can also attempt to link occurrences of one or more I_{dev}-s.

A.2 IM-PAISA **Requirements**

System requirements are the same as in DB-PAISA, except that scaling to multiple users is no longer a concern in IM-PAISA. Although IM-PAISA operates in settings with large numbers of I_{dev}-s, only an authorized O_{dev} can solicit information from them.

Security requirements (beyond those of DB-PAISA**):**

- *Request authentication:* I_{dev} must validate each Msg_{req}.
- *Response confidentiality:* Msg_{resp} must not leak information about I_{dev}.
- *Unlinkability:* Given any two valid Msg_{resp}-s, the probability of determining if they were produced by the same I_{dev} should be negligibly close to 50%, for any party except O_{dev}.

A.3 IM-PAISA **Protocol Details**

In this section, we only describe the aspects of IM-PAISA that differ from DB-PAISA.

A.3.1 Registration. pk_{owner}, $\mathcal{K}$, and some metadata are securely installed for each I_{dev}. However, information, that is not deployment-dependent (e.g., $SW_{IM-PAISA}$, $H_{SW_{dev}}$, and peripheral configuration), is assumed to be securely provisioned earlier by Mfr. Note that, since the owner is aware of all identities and types of its I_{dev}-s, there is no longer any need for device information to be provisioned on I_{dev} or maintained by Mfr. As a result, Mfr does not play any active role in IM-PAISA. The **TCB** of IM-PAISA is identical to that of DB-PAISA.

A.3.2 Runtime. Similar to DB-PAISA, the IM-PAISA *Runtime* phase has three steps: Request, Response, and Reception.

***$SW_{IM-PAISA}$* on I_{dev}:** As discussed in Section A.1, DoS attacks by a network-based $\mathcal{A}$dv are out-of-scope. Also, scalability is not an issue since only one O_{dev} is assumed. Furthermore, because O_{dev}'s requests are expected to be much less frequent than U_{dev}'s requests in DB-PAISA, $\mathcal{A}$tt is performed on the fly upon each O_{dev}'s request.

Figure 10 shows four states of I_{dev}:

(a) $\mathcal{W}$ait: The only transition is switching to **$\mathcal{V}$erify** when Msg_{req} is received.

(b) $\mathcal{V}$erify: I_{dev} verifies Msg_{req} with pk_{owner}. If verification succeeds, it transitions to **$\mathcal{A}$tt**. Otherwise, it discards Msg_{req} and returns to **$\mathcal{W}$ait**.

(c) $\mathcal{A}$tt: I_{dev} computes Att_{result} and transitions to **$\mathcal{G}$en**. Note that I_{dev} need not compute the attestation time or include it in Msg_{resp} because $\mathcal{A}$tt occurs with every Msg_{req}.

(d) $\mathcal{G}$en: I_{dev} encrypts N_{owner} (O_{dev}'s nonce in Msg_{req}), device information, and Att_{report} with $\mathcal{K}$. It composes Msg_{resp} that contains an authentication tag. Encryption offers Msg_{resp} confidentiality and unlinkability.

IM – PAISA app on O_{dev}: There are two steps on O_{dev}: Request and Reception.

Request**:** O_{dev} signs N_{owner} with sk_{owner} and broadcasts Msg_{req}, containing N_{owner} and the signature (Sig_{req}). After sending Msg_{req}, it starts a scan to receive Msg_{resp}-s from potential nearby I_{dev}-s.

Reception**:** Upon receiving Msg_{resp}, O_{dev} retrieves the corresponding $\mathcal{K}$ in brute-force attempts. After retrieving $\mathcal{K}$, it decrypts Msg_{resp} with $\mathcal{K}$, and finally, device details are displayed on O_{dev}.