**EDITOR:** Schahram Dustdar, dustdar@dsg.tuwien.ac.at

DEPARTMENT: INTERNET OF THINGS, PEOPLE, AND PROCESSES

# OTA-TinyML: Over the Air Deployment of TinyML Models and Execution on IoT Devices

Bharath Sudharsan and John G. Breslin, *Data Science Institute, NUI Galway, H91 TK33, Galway, Ireland*

Mehreen Tahir, *SFI Center for Research Training, Dublin City University, 9, Dublin, Ireland*

Muhammad Intizar Ali, *School of Electronic Engineering, Dublin City University, 9, Dublin, Ireland*

Omer Rana, *Cardiff University, CF24 3AA, Cardiff, U.K.*

Schahram Dustdar, *TU Wien, 1040, Vienna, Austria*

Rajiv Ranjan, *Newcastle University, NE1 7RU, Newcastle upon Tyne, U.K.*

*This article presents a novel over-the-air (OTA) technique to remotely deploy tiny ML models over Internet of Things (IoT) devices and perform tasks, such as machine learning (ML) model updates, firmware reflashing, reconfiguration, or repurposing. We discuss relevant challenges for OTA ML deployment over IoT both at the scientific and engineering level. We propose OTA-TinyML to enable resource-constrained IoT devices to perform end-to-end fetching, storage, and execution of many TinyML models. OTA-TinyML loads the C source file of ML models from a web server into the embedded IoT devices via HTTPS. OTA-TinyML is tested by performing remote fetching of six types of ML models, storing them on four types of memory units, then loading and executing on seven popular MCU boards.*

Internet of Things (IoT) devices are estimated to reach 75 billion by 2025, with a large percentage of devices being only wirelessly enabled. Often, these devices have limited to no physical access to reprogram and rely on remote techniques for updates. Over-the-air (OTA) updates are one of the famous techniques to remotely, efficiently, and reliably update devices in the field. It not only improves safety and compliance but also reduces operational and maintenance costs and offers the potential for enhanced revenue streams, as the OEMs can offer add-on services without having expensive service technicians perform the updates in person.[1] However, these benefits must be balanced by the risks. A poorly executed OTA update can not only result in bricked devices and significant inconvenience to consumers but can also cost reputational damage to the OEM.[2] Moreover, the majority of IoT devices are embedded systems with a low-cost MCU or a small CPU and a few MBs of memory, which is only sufficient for routine device functionalities. The hardware of such devices has processor chipsets from various vendors, introducing system heterogeneity and thereby complexity. Such challenges impact the OTA update/programming process, increasing cases reporting that tiny devices get into an inconsistent state.[3]

To cope with such challenges, the contributions from industries and practitioners have added essential features to OTA methods, such as update version rollback, integrity verification, update failure management, etc. Such sophisticated OTA methods are not suitable for small CPU- and MCU-based resource-constrained IoT devices. Another interesting fact to note is that these methods are tested only for non-machine-learning (ML) tasks, creating a gap in ML-based edge computing. This gap makes the OEMs and developers practice the conventional approach of flashing the generated C source file (model as a char array) of a trained ML algorithm on their IoT devices using a USBJTAG programming device.[4]

1089-7801 © 2022 IEEE
Digital Object Identifier 10.1109/MIC.2021.3133552
Date of current version 21 June 2022.

This article introduces OTA-TinyML to enable end-to-end remote fetching, storing, and execution of numerous TinyML models on IoT devices. OTA-TinyML is designed considering the constraints in low-cost devices, so it does not impair their routine functionalities and service time. Also, OTA-TinyML provides OEMs and developers the freedom to remotely repurpose their globally deployed devices without the need for physical reflashing. The contribution of this work can be summarized as follows.

1) Initially, the challenges faced by resource-constraint IoT devices in OTA settings are discussed. Then, we present a broad overview of current approaches with several future directions to resolve challenges. Next, in the context of the embedded system of IoT devices, we present the types plus fundamentals of memory units and file systems (FSs), followed by considerations when building a distributable memory medium containing multiple TinyML models.
2) This article introduces OTA-TinyML, an IoT hardware-friendly approach to enable ubiquitous tiny devices to fetch and store TinyML models from the cloud. OTA-TinyML also supports efficient on-device loading and execution of the fetched models from a variety of available memory units and internal or external FS types. OTA-TinyML[a] is released as an open-source repository. Using this work as a base, researchers and engineers can incrementally add features to the OTA-TinyML code to address the challenges covered in this article.

## Outline

The "Loading TinyML Models via OTA Updates" section covers challenges when loading TinyML models via OTA updates, followed by approaches to tackle challenges. The "TinyML on Memory Units in IoT Boards" section presents memory units and FS rudiments as well as considerations when building a distributable medium holding multiple TinyML models. The "OTA-TinyML Design" section covers the OTA-TinyML design and testing. Finally, the "Conclusion" section concludes this article with a context for future research.

# LOADING TINYML MODELS VIA OTA UPDATES

The OTA update process is vital in any IoT system as it allows OEMs to seamlessly flash new firmware, rapidly respond to bugs and security vulnerabilities, and update on-device models without the need for physical recalls of devices or truck rolls. A successful OTA update requires complex coordination between device hardware, firmware, network connectivity, and the OEM's cloud, making OTA updates sophisticated when executed on tiny devices.

## Challenges

This section explores the research and engineering challenges at various levels of the OTA update process for tiny devices.

### *IoT Hardware Level*

In the standard OTA firmware update process, OEMs initially generate a new firmware file ready for distribution. On the edge device, the flash memory is split into slots allocated to the bootloader and firmware. The newly compiled firmware from the OEM is received and stored on the firmware slot, ready to be executed upon reboot. This is the typical approach widely adopted in practice[5] where the current firmware version is simply replaced with the updated file from the OEM. No matter the approach, the initial step is to equip IoT devices with a resource-friendly program that can accommodate the OTA update process.

Flaws at the chipset level can be exploited to compromise the device by tricking it into downloading substitute malware from a different server. When security vulnerabilities of earlier firmware versions are known, the device can be tricked into downgrading to the flawed software, increasing its vulnerability. Physical attacks can be mounted on devices by reading memory units, extracting cryptographic keys, metadata, or others that can aid in compromising future updates.

The hardware of most IoT devices has processor chipsets from various vendors and is intrinsically heterogeneous in terms of memory capacity, computing power, and security capability.[6] Therefore, the heterogeneous characteristics of these tiny devices should be taken into consideration, and only compatible OTA updates should be rolled out to avoid bricking the devices.

### *Network and Transport Level*

When update files are stored on untrusted repositories in the cloud and unencrypted, reverse engineering attacks can occur. Establishing and maintaining an uninterrupted network connection for the duration of the OTA update is crucial for IoT devices. For instance, in horizontal IoT integrations (networking between installed devices), a fleet of devices is reached via Digi XBee sub-1 GHz networks or via

[a] Code: https://github.com/bharathsudharsan/OTA-TinyML

standard 2.4 GHz Zigbee, LR-WPAN. Here, the bandwidth constraints or wireless interference (common in factory floors) can cause transmission errors or corrupt the OTA files that become unusable when rendered.

For transferring updates over the network, each protocol provides its specifications. For example, the basic device firmware update transport scheme can use low-power bluetooth or bus technologies like USB for transport. Constrained application protocol (CoAP) contains features equivalent to HTTP but constrained edge devices friendly. When the situation of edge device demands update transport over heterogeneous low-power networks in several hops, the network stack can be a combination of CoAP over TCP/transport layer security (TLS) or CoAP over UDP. For networks that cannot directly use IPv6, 6LoWPAN can be leveraged. Overall, for security during transport, DTLS and TLS profiles need to be used in IoT deployments.

The IoT networks are expanding exponentially, and with the expanded deployment, everything must scale accordingly. If an OTA update is rolled out to a large pool of devices, the OTA update requests can easily overwhelm the server and cause denial-of-service (DoS) issues. For this reason, the network should be able to scale and adapt automatically and efficiently handle the network traffic without reaching a breaking point.

### *Cryptography Level*

A typical way to ensure secure updates is utilizing cryptographic algorithms. However, the challenge is, the standard algorithms used on servers and edge gateways cannot be directly deployed on constrained MCU- and small CPU-based devices. Engineering efforts in terms of selection, optimization, cross-platform porting, and memory profiling are required to finalize an algorithm for the given device. For example, for public-key cryptography, compared to RSA, elliptic curve cryptography can be used as the key sizes are smaller. The elliptic curve digital signature algorithm standardized by the National Institute of Standards and Technology (NIST) can be used with the P256r1 curve is widely practiced in the industry.[7] In the context of constrained devices, highly configurable libraries need to be chosen. Also, algorithms need to be considered that provide roughly 128-bit cryptographic strength. TinyCrypt can be an example, as it provides signatures based on the NIST P256r1 curve with emphasis on minimal code size and cryptographic dependencies. Other relevant libraries suitable for constrained budgets are HACL*, Mbed TLS, TweetNaCl, NaCl, WolfSSL, and Monocypher.

### *Remote Device Management and Operating System Level*

One of the most prominent approaches for remote edge device management is the lightweight machine-to-machine (LwM2M) protocol with data transfer using CoAP and secured with DTLS. The several RESTful interfaces defined by LwM2M can manage various interfaces with devices, perform data reporting, and device actuation. For example, execute a factory reset, read the serial number, and write the current time. Alternative to RESTful design is the usage of remote procedure calls as in CPE WAN management protocol (CWMP), which offers firmware updates for higher end devices like IoT printers.

Not only devices but also device management systems can also be compromised even when using the standard LwM2M, CWMP. Especially in vertical IoT integrations (from sensors to the business level), the linked third-party device-management systems have high-privilege access to massive networks of devices, making it a potential compromised source. When originating large-scale OTA updates using such a system, secured access needs to be enforced.

The top OTA methods are only suited for devices with higher computing resources that can run at least the tiny Linux OS such as Minibian, FreeBSD, or Windows 10 IoT core. But the hardware of interest to this study cannot accommodate such OS. More shredded OS of interest that can incorporate complex protocol stacks, such as IPv6, DTLS, UDP, and CoAP are the popular FreeRTOS, $\mu$C/OS, followed by RIOT, Mbed OS, Zephyr, Tock. Implementing OTA methods on top of such shredded OS is yet to be explored.

## Directions for Handling Challenges

This section presents techniques to ensure secure, reliable, and successful OTA updates for IoT devices.

### *Batching Devices and Update Roll-Out Phases*

There can exist millions of devices designed using parts supplied from one OEM (e.g., using a particular chipset from MediaTek). Also, the same device can serve different purposes based on the edge application flashed on it. So to avoid negative impacts, the device-management system should batch devices considering both hardware specifications and its software status. Then, communicate information on the availability of OTA updates and how to fetch them. Here, only the highly trusted users must be given device-management system access to minimize the chances of malware injection into the system. When the batch contains a high volume of devices, simultaneous advertisement of update availability can overwhelm the server with OTA requests causing DoS issues. So the updates need to be rolled out in incremental phases.

### *Remote and Physical Security*

Typically, IoT devices are connected to a server via a secure telemetry channel that operates using MQTT. Here, to reduce the attack surface, a separate mechanism (e.g., establishing a new connection to the dedicated OTA server) needs to be avoided for OTA updates. Instead, the same MQTT channel can be leveraged for downloads. Tiny IoT devices designed using MCUs, small CPUs that execute both the edge application and networking stack in the same constrained memory unit, making MQTT more memory-efficient as it does not require an additional HTTP client. Invariable of weather using the MQTT channel or a separate HTTPS server, TLS needs to be used for initial secure connection establishment.

Orthogonal to remote attacks, which are the common form of security threat, IoT devices can also get compromised by physical attacks. Outdoor devices like security cameras, video doorbells are physically vulnerable as their casings can be removed for hardware access. Security of such devices can be improved by using tamper-proof screws wired to an alerting system, removing JTAG ports. Also, encrypting sensitive data like code, credentials, and keys makes the information useless after managing to read memory units of devices.

### *Minimizing Unexpected Saturation and Downtimes*

Timely updates can be guaranteed only when edge applications perform checks in the background as frequently as possible. However, in smart infrastructures hosting thousands of IoT devices, networks can get congested (particularly the low bandwidth sub 1 GHz networks) due to an increase in devices-server update communication. A classic example of such a scenario could be the installation of the same model cameras throughout the facility, and there is an update released. Here, a priority-based progressive deployment approach needs to be adopted to avoid network congestion while also considering update of the nature that can be minor (fixing bugs), major (new firmware release), or emergency (security issues). The constrained resources of IoT devices can overflow due to continuous update requests to servers during update server unavailability and/or network congestion. Here, the to/from counts of unsuccessful requests must be limited with time-shifting by setting a 24-h wait period before starting over the process. The criticality of devices in use should also be considered when scheduling updates. For remote sensors that perform periodic sensing, it is acceptable to cease normal operation. But there should be no updates when medical devices are in use, a connected car is driven, a robotic cleaner or a coffee machine is in the midst of a job.

### *End-to-End Integrity Protection*

There is never a guarantee for devices to receive authentic files as the files can be swapped in transit by a man-in-the-middle, attack, or transmission errors can corrupt the image. To ensure the end-to-end integrity of the file, the OEM should sign the image using the code-signing certificate and attach metadata, both of which should be hashed with a private key in the OEM's certificate to generate a signature. In addition to securing servers to avoid malicious firmware swaps, OEMs must secure supply chain software deployments to avoid integrating malicious code during the deployment life-cycle. Also, the exposure of private signing keys should be prevented. Storing firmware's digital signature on a blockchain can also be investigated to improve OTA security in IoT.[8]

At the device end, to evaluate the authenticity of the received files, devices with the OEM's public signing certificate decrypts and compares the hash. Devices can conduct additional checks by using metadata to make sure the received image is the most recent and not the flawed earlier versions; using a chipset that supports secure boot or by making the bootloader keep a check on the sign to avoid booting an unauthorized image.

### *Failure Recovery*

Numerous factors, including power outages, faulty batteries, lost connectivity, and impatient user restarting a seemingly unresponsive device, can cause the OTA updates to fail. To track the OTA outcomes, tests need to be in place that should ideally be atomic with binary outcomes—either failure or success. The update can be considered a success when the device boots the new image and reconnects to the IoT service. If the test fails, devices must be able to revert to the previous image to continue operating. Otherwise, the failed update may result in a bricked device. In case of failures, the crucial aspects to troubleshooting are there should exist ways to remote login to the device for decentralized investigation and extract device logs for central investigation and correlations.

**TABLE 1.** Sample list to evaluate OTA update mechanisms.

| Attack | Description |
|---|---|
| Resource exhaustion | Repeated attempts for fraudulent updates for long periods |
| Offline devices | Cut communication between devices and the OTA server |
| Firmware mismatch | Replay an authentic update but for incompatible devices |
| Firmware tampering | Update devices with intentionally flawed image |
| Wrong memory unit | Flashing new firmware on wrong memory location or unit |
| Reverse engineering | Eavesdropping updates while transmitted over networks |

### *Evaluation Metrics*

A few typical firmware update threats are briefed in Table 1. It is recommended to evaluate and compare the security of any OTA update mechanism based on such a, but more detailed list. We give an example evaluation for offline devices. The IPv6-based mechanism may not mitigate this threat. Using the SUIT specification-based mechanism that uses a best-before timestamp to expire updates provides mitigation provided devices have a real-time clock module. If time information is not available in cases of low-cost devices, SUIT configuration cannot be useful. Whereas, when using the LwM2M mechanism, the device can be protected as LwM2M can provide current time to devices.

## TINYML ON MEMORY UNITS IN IOT BOARDS

IoT devices usually come with different memory units, having distinct specifications and underlying FSs. This section presents the memory unit selection guidelines and rudiments to designing embedded systems for handling TinyML models.

### Memory Unit and FS Rudiments

Following are the distinct memory units and FS types to host TinyML models-powered IoT applications.

### *EEPROM File System*

EEPROM is the most commonly considered nonvolatile memory unit to add to an embedded system. EEPROM file system (EEFS) serves the purpose of FS abstraction for EEPROM, RAM, and PROM memories while being efficient, lightweight, and reliable.[9,10] The EEFS can be used on devices that cannot support a full MSDOS FS but require the ability to patch, dump, and diagnose files. EEFS contains slots of different sizes, where each slot is a fixed size contiguous region of memory allocated for a single file. The size of each slot is based on the file to be stored in the slot and is determined during FS creation. Keeping each file in one contiguous memory area makes it quicker to patch or reload an EEPROM without going through the FS interface. Additional free space can be added to the end of each slot to allow room for the file to grow in size if necessary. Since EEFS is used with EEPROM, files will not change very often—in most cases, it will be used for writing once read many. Ferroelectric random access memory (FRAM) can be used as a direct replacement for serial flash and EEPROM. FRAM units offer several advantages over EEPROM and are available in a range of memory sizes and packages.

### *On-Chip Flash*

Flash is one type of EEPROM. Data can be stored permanently (persisted across resets or power failures) on the flash memory of MCUs and small CPUs, e.g., using `Preferences.h` library when using Arduino IDE. The popular ESP32 board has about 4 MB of internal flash memory (see Table 2) in which data like threshold values, network credentials, and API keys can be reliably stored, which can be read by edge applications upon demand.

Modern MCUs have flash memory self-contained within the same chip as the processing unit. Also, a higher capacity flash chip can be soldered on the board, outside the processor chip, and interfaced via a protocol. Invariable of their placement, in-system programming (ISP) or in-application programming methods can be used for reprogramming flash memory. The worst-case scenario can be a crash or power outage while reprogramming the flash. In a few edge application scenarios, manual recovery by a technician via ISP is acceptable. However, for truly remote systems, such as surveillance set up in the ocean, middle of a forest, a sophisticated intrusion-free self-recovery mechanism is desirable.

### *SPI Flash File System*

This is an internal FS that can store files in the NOR flash chip. Instead of the traditional file-by-file upload into the storage of MCUs, an SPI flash file system (SPIFFS) image could be prepared offline and flashed at once.[11] The main benefits of using SPIFFS are: low RAM usage; supports Posix-like API that can accept open, close, read, write, etc. commands; compatible

**TABLE 2.** Popular MCU boards (top) and optimized INT8 models (bottom) used for OTA-TinyML testing.

| Name | Processor | Flash (MB) | SRAM | Clock (MHz) |
|---|---|---|---|---|
| B1: Teensy 4.0 | Cortex-M7 | 2 | 1 MB | 600 |
| B2: STM32 Nucleo H7 | Cortex-M7 | 2 | 1 MB | 480 |
| B3: Arduino Portenta | Cortex-M7+M4 | 16 | 8 MB | 480 |
| B4: Feather M4 Express | Cortex-M4 | 2 | 192 KB | 120 |
| B5: Generic ESP32 | Xtensa LX6 | 4 | 520 KB | 240 |
| B6: Arduino Nano 33 | Cortex-M4 | 1 | 256 KB | 64 |
| B7: Raspberry Pi Pico | Cortex-M0+ | 16 | 264 KB | 133 |

| Task: Model Name | Score | .tflite (KB) | .h (KB) |
|---|---|---|---|
| Recognize Gestures: MagicWand | 0.67 (Acc) | 19 | 118 |
| Visual Wake Words: MicroNet S-L | 0.76-0.82 (Acc) | 273-529 | 1689-3267 |
| Speech Recognition: Wav2letter | 0.0783 (LER) | 22600 | 143421 |
| Keyword Spotting: DNN S-L | 0.82-0.86 (Acc) | 82-491 | 508-3029 |
| Keyword Spotting: CNN S-L | 0.91-0.92 (Acc) | 75-492 | 436-3029 |
| Keyword Spotting: MicroSpeech | 0.62 (Acc) | 18 | 112 |
| Image Classification: MobileNet v2 | 0.69 (Acc) | 3927 | 24215 |
| Anomaly Detection: MicroNet S-L | 0.95-0.96 (AUC) | 246-452 | 1523-2794 |

with any NOR flash, not only SPI flash; multiple SPIFFS configurations can run on the same IoT board, or the interfaced SPI flash module; highly configurable with built-in FS consistency checks.

### *SD Card File System*

Standard SD and SDHC cards can be interfaced with MCU boards via SPI SD card modules. Libraries allow the IoT application to move through directories, create and read/write files. Compared to the above memory units, this contains numerous contributions from the open-source electronic prototyping community, and it can support FAT16 and FAT32 FSs.

### *Embedded Multimedia Card*

Embedded Multimedia Card (eMMC) is a chip packaged with both flash memory and its controller. eMMCs and SD cards are the same from the interface standpoint as they have the common pins that connect to MCUs. eMMCs are comparatively more expensive and come with benefits, such as higher reliability, due to a lower likelihood of physical corruption and faster IoT application interaction with memory.

## Memory Unit Selection Guidelines

Not all memory units are created equal. Hence, when building a distributable medium containing many TinyML models for on-the-fly demand-based IoT devices, repurposing, the following points can be considered.

### *Interplay Among Cost, Space, and Speed*

EEPROMs are cheaper than SD cards. If the MCUs lack an internal EEPROM, they can be purchased for just a few cents, e.g., a 64 KB of storage costs ≈ $1.50. SD cards need adapters, while an EEPROM is one physically small chip ready to be used after soldering. EEPROM bit rate ranges from 100 to 1000 Kbits/s. SD cards can only use SPI, which is faster than I2C but more susceptible to noise than I2C. Although EEPROMs can support both I2C, SPI interfaces and reach bus speeds of 1 MHz+, their write cycles are slower at ≈ 500 ns, and multiple instructions are needed to write to a cell. When there are no-cost restrictions, FRAM can be considered as it is write cycles are ≈ less than 50 ns and more radiation tolerant.

### *Energy and Memory*

FRAM just needs 1.5 V to operate; EEPROMs can operate from 1.8–to 5 V, while SD card needs 3.3 V. Also, the peak current for SD cards is 100 mA and less than 10 mA for EEPROM. Most SD libraries use a huge amount of RAM and program memory compared to EEPROM, which only consumes little to interface. The smallest SD cards have much higher storage than the largest EEPROMs. Developers need to consider the expected erase/write cycles during design as well. Both EEPROMs and SD cards have a limited number of write/erase cycles. For EEPROM, it is typically limited to one million cycles. For SD cards, the number is much less and unknown, making end life calculations less accurate. For FRAM, it is upwards of one trillion and is the highest.

### *Reliability*

For devices using SD cards, there is a risk of sudden failure and loss of all data without any warning—voltage transients, elevated temperatures, shocks, dust, and continuous use can be the reason. Also, if devices are installed in a high-vibration environment, an SD card can shake out of its slot, making the soldered EEPROM or eMMC better options. Subsystems designed for vehicles require memory units to be certified to a higher standard than flash, SD cards, making an auto-certified eMMC more suitable.

## OTA-TINYML DESIGN

The operational flow of OTA-TinyML is shown in Figure 1, comprising two parts. The first part circled,[1] explained in the "Part 1: Fetching TinyML Models via HTTPS" section, contains a method, that upon demand, fetches ML model files from the cloud server on the edge devices. The second part circled,[2] explained in the "Part 2: Store, Execute TinyML Models from Memory Units" section, contains a method to enable storage of fetched files in internal memory or external FSs, then the loading and model execution.

### Part 1: Fetching TinyML Models Via HTTPS

This part of OTA-TinyML enables IoT devices to download ML models from the internet. Initially, the TinyML models need to be stored in the web server in the formats `.bin` (model as a compressed binary file) or as `.h` (model as a C array), both of which can be generated from the trained model using API `TFLiteConverter.from_keras_model()`. After setting up the web server, these models can be downloaded upon request from the edge devices using the target server address along with the directory/path of the ML models. On devices, OTA-TinyML initially establishes a connection to the server via Ethernet or WiFi. Then, it downloads the target from the HTTPS URL using `http.get()` method of HTTPClient object and passes the file to OTA-TinyML Part 2 for storage on the available memory unit of the edge device.

### Part 2: Store, Execute TinyML Models From Memory Units

This part of OTA-TinyML enables storing of multiple ML models (fetched from a web server) on any memory unit of choice. It is compatible with internal memory units like on-chip flash and SPIFFS as well as external FSs like EEFS and SD card file system (SDFS). This part also is responsible for loading and executing models demanded by the IoT applications.

In conventional TinyML approaches, after the ML model training phase, the output model is converted to an array and exported as a C header file. This file is imported into the code of the IoT application using `#include model_name.h`, on which the TF Lite Micro interpreter is run to obtain predictions. Orthogonal to these conventional TinyML approaches, we show that, on the MCU boards, it is possible to load an ML model from a file instead of a C array. While executing various TinyML models on MCU boards, we found out and also report here that

> *Interpreters works identical in both cases - whether the model is declared as an array from the beginning or loaded as an array from somewhere else*

Using this finding/concept, OTA-TinyML initially reads the ML model in `model_name.bin` format stored in any memory unit into a byte array. Then, it allocates memory for the read model using the `malloc()` function and copies the model content byte-by-byte, from the `.bin` file to the MCU SRAM memory, using which the interpreter produces predictions.

### OTA-TinyML Testing

The models and MCU boards used for OTA-TinyML testing are given in Table 2. For Part 1 testing, to ensure extensiveness, the .h file size of models we fetched from cloud to edge MCUs varies from 112 KB (MicroSpeech) to 143421 KB (Wav2letter). For Part 2, we used four types of memory units to extensively test the onboard model storing and loading performance of OTA-TinyML. So SDFS is interfaced to B3, B7; EEFS to B6, B4; Internal SPIFFS of B5; Internal flash memory of B1, B2. As shown in Figure 1, we first upload the .bin files of 16 pretrained ML models into an HTTPS web server. Then, the C++ implementation of the OTA-TinyML approach is provided as a .ino file

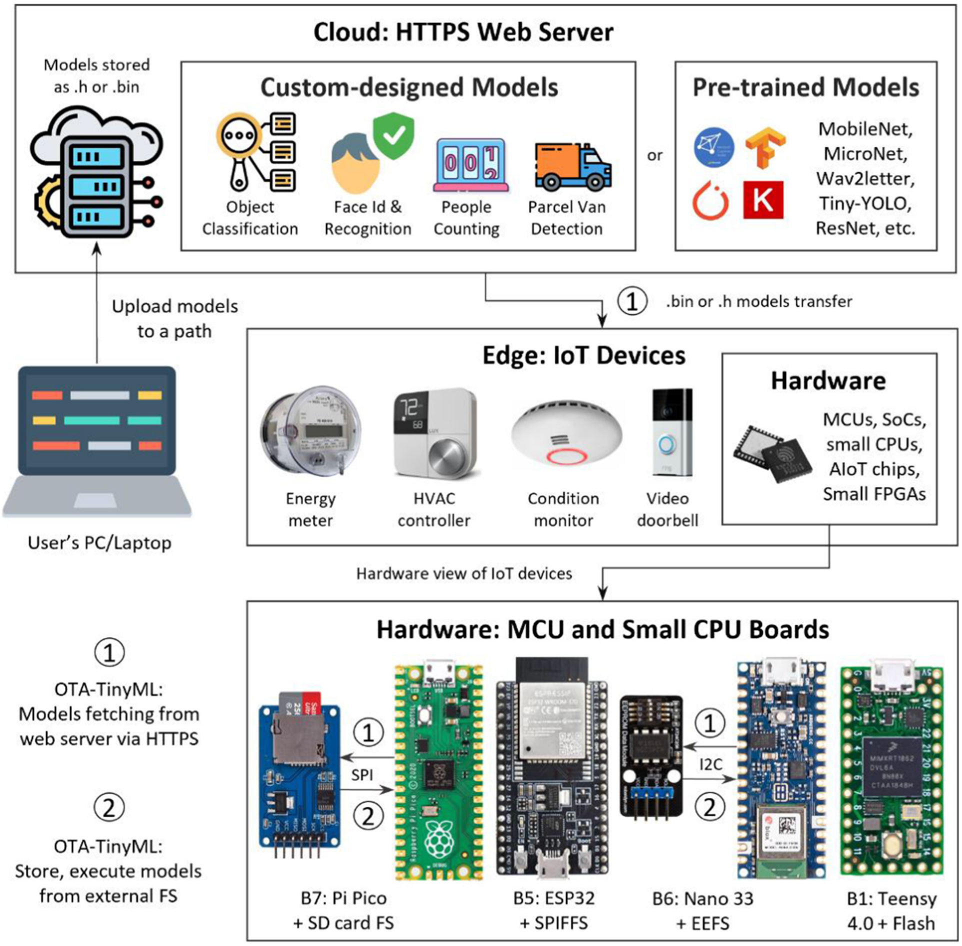

**FIGURE 1.** OTA-TinyML for models fetching from a web server, storing in internal memory or external FS, and execution on IoT boards upon demand.

containing the server details is flashed on 7 MCU boards B1 to B7 using Arduino IDE. At this stage, both the server and edge devices are ready for OTA-TinyML testing.

Starting from B1 to B7, we instructed devices to initialize the model fetching process. The boards B3, B7 with SDFS have the highest storage capacity, so they downloaded all 16 models (≈188 MB). Similarly, the other boards downloaded models according to their memory limits. Next, the fetched model files get stored on the FS interface to the boards. Then, based on the device SRAM capacity (see Table 2), models are loaded from FS and executed to produce inference results. For example, B4 with the least SRAM of 192 KB used OTA-TinyML only on MagicWand and Micro-Speech models. Whereas boards B1–B3, with a better SRAM capacity, used the OTA-TinyML to load and execute more model varieties. In summary, despite the diversity in MCU hardware specification or manufacturer, OTA-TinyML part 1 successfully fetched different size models from the cloud without stalling the devices. Part 2 successfully stored, loaded, and executed models from internal memory (Flash, SPIFFS) and also from external FS (SDFS, EEFS).

## CONCLUSION

This article investigated the research and engineering challenges when practicing OTA machine learning involving heterogeneous IoT devices that are constrained in multiple aspects. OTA-TinyML was introduced as an IoT hardware friendly mechanism, whose open-sourced implementation allows OEMs and developers to keep their deployed devices evergreen while also being able to remotely repurpose (load and run the model on demand) their devices on the fly.

Initial OTA-TinyML testing was performed involving popular models and IoT development boards among the TinyML community that demonstrated the end-to-end fetching, storage, and execution of many ML models on a single memory-constrained device. For example, with OTA-TinyML, even the low-cost US$3 ESP32 chip with only 4 MB flash can dynamically fetch n models from web server, such as keyword spotting (3 MB), anomaly detection (2.7 MB), visual wake words (3.2 MB), etc., store in internal memory or external FSs, then execute any model upon demand.

Future work plans to use OTA-TinyML as a base, upon which features can be added to address the covered challenges for bringing it closer to the standard OTA mechanisms of Android, edge gateways, and other better hardware-specification devices existing around us.

## ACKNOWLEDGMENT

This work was supported in part by Science Foundation Ireland (SFI) under Grant SFI/16/RC/3918 (Confirm) and Grant SFI/12/RC/2289_P2 (Insight), with both grants cofunded by the European Regional Development Fund.

## REFERENCES

1. M. M. Villegas, C. Orellana, and H. Astudillo, “A study of over-theair (OTA) update systems for CPS and IoT operating systems,” in *Proc. 13th Eur. Conf. Softw. Archit.*, 2019, pp. 269–272.
2. K. Zandberg, K. Schleiser, F. Acosta, H. Tschofenig, and E. Baccelli, “Secure firmware updates for constrained IoT devices using open standards: A reality check,” *IEEE Access*, vol. 7, pp. 71 907–71 920, 2019.
3. N. Lethaby, “A more secure and reliable OTA update architecture for IoT devices,” Texas Instruments, 2018.
4. B. Sudharsan, P. Patel, J. G. Breslin, and M. I. Ali, “Enabling machine learning on the edge using SRAM conserving efficient neural networks execution approach,” in *Proc. Joint Eur. Conf. Mach. Learn. Knowl. Discov. Databases*, 2021, pp. 20–35.
5. D. Frisch, S. Reißmann, and C. Pape, “An over the air update mechanism for ESP8266 microcontrollers,” in *Proc. 12th Int. Conf. Syst. Netw. Commun.*, 2017, pp. 1–6.
6. B. Sudharsan, P. Yadav, J. G. Breslin, and M. I. Ali, “An SRAM optimized approach for constant memory consumption and ultra-fast execution of ML classifiers on TinyML hardware,” in *Proc. IEEE Int. Conf. Serv. Comput.*, 2021, pp. 319–328.
7. N. I. O. S. A. Technology, “Digital signature standard (DSS),” 2013. [Online]. Available: https://csrc.nist.gov/publications/detail/fips/186/4/final
8. “Using blockchain technology to secure the Internet of Things,” Accessed: Jul. 03, 2021. [Online]. Available: https://cloudsecurityalliance.org/artifacts/using-blockchain-technology-to-secure-the-internet-of-things/
9. “Jesfs.” Accessed: Jul. 04, 2021. [Online]. Available: https://github.com/joembedded/JesFs
10. “NASA EEPROM file system.” Accessed: Jul. 04, 2021. [Online]. Available: https://github.com/nasa/eefs
11. “Internal filesystem whitepaper,” NodeMCU. Accessed: Jul. 17, 2021. [Online]. Available: https://nodemcu.readthedocs.io/en/dev/spiffs/

**BHARATH SUDHARSAN** is currently working toward the Ph.D. degree with the CONFIRM SFI Centre for Smart Manufacturing, Data Science Institute, National University of Ireland Galway, Galway, Ireland. His research interests include TinyML, optimization methods, ML systems, edge computing, and collaborative learning. Contact him at bharath.sudharsan@insight-centre.org.

**JOHN G. BRESLIN** is a personal professor (personal chair) in electronic engineering with the College of Science and Engineering, National University of Ireland Galway, Galway, Ireland, where he is the Director of the TechInnovate/AgInnovate programmes. Contact him at john.breslin@insight-centre.org.

**MEHREEN TAHIR** is currently working toward the Ph.D. degree with an SFI Center for Research Training, Dublin City University, Dublin, Ireland. Her research interests include cloud computing, federated learning, big data analytics, and data science. Contact her at mehreen.tahir2@mail.dcu.ie.

**MUHAMMAD INTIZAR ALI** is an assistant professor with the School of Electronic Engineering, Dublin City University, Dublin, Ireland. His research interests include data analytics, Internet of Things, stream query processing, data integration, distributed and federated machine learning, and knowledge graphs. Contact him at ali.intizar@dcu.ie.

**OMER RANA** is a currently a professor of performance engineering and previously led the Complex Systems Research Group, School of Computer Science and Informatics, Cardiff University, Cardiff, U.K. His research interests include overlap between intelligent systems and high-performance distributed computing, particularly in understanding how intelligent

techniques could be used to support resource management in distributed systems, and the use of these techniques in various application areas. Contact him at ranaof@cardiff.ac.uk.

**SCHAHRAM DUSTDAR** is currently a professor of computer science and the Head of the Distributed Systems Group, TU Wien, Vienna, Austria. He was named Fellow of IEEE in 2016 for contributions to elastic computing for cloud applications. He is the corresponding author of this article. Contact him at dustdar@dsg.tuwien.ac.at.

**RAJIV RANJAN** is the University Chair Professor for the Internet of Things research with the School of Computing, Newcastle University, Newcastle upon Tyne, U.K. His research interests include distributed systems (cloud computing, Big Data, and the Internet of Things). Contact him at raj.ranjan@ncl.ac.uk.