Secure Deployment of Containerized IoT Systems

Bryan Pearson
College of Engineering and Computer Science
University of Central Florida
Orlando, FL, USA
bpearson@knights.ucf.edu

Daniel Plante
College of Arts and Sciences
Stetson University
DeLand, FL, USA
dplante@stetson.edu

IOT Security (agriculture)
container Orchestration
IOT ContINERIZATION

SoutheastCon 2020 | 978-1-7281-6861-6/20/$31.00 ©2020 IEEE | DOI: 10.1109/SoutheastCon44009.2020.9368276

***Abstract*—Internet of Things (IoT) popularity and complexity has grown at an accelerating rate. As a result, IoT security is more critical than ever before. Many services attempt to simplify the bottlenecks of IoT infrastructure, development and deployment which previously fell to developers. However, these services are often proprietary and insecure. In this paper, we explore an alternative IoT service model that uses Docker to securely bootstrap an IoT network running on 3 Raspberry Pis. This service uses containerization technology with Linux to abstract architectural details, software update mechanisms, and network security away from the developer. Devices authenticate and communicate with a remote server using private key cryptography, while OTA updates are delivered via the Docker registry. We show that this model satisfies modern security standards of lightweight technology and fulfills expectations of a deployment platform.**

***Keywords*—Internet of Things, Security, Docker, Containerization**

I. Introduction

Internet of Things (IoT) has grown substantially in the last decade. IoT connects devices of all types and functions together to produce unique applications of both physical and digital capacity. IoT technology is currently utilized in consumer, healthcare, agriculture, smart city, and various other domains. Forbes predicts that by 2021, the IoT market will reach $520 billion [6].

Current trends in IoT security are worrying. Vendors will frequently expose vulnerable network ports, set up ad-hoc wifi networks which are vulnerable to snooping, and leverage primitive microcontrollers and chips which cannot receive over-the-air (OTA) updates [1] [9] [20]. The authors of the Mirai botnet [5] leveraged vulnerabilities in IoT cameras and routers to launch a massive DDoS attack against Dyn servers. Similar intrusions have been documented for other products such as St. Jude's pacemakers [22], Owlet baby monitors [25], and TRENDnet web cams [26].

IoT complexity and popularity has also given rise to popular service models for device provisioning and enterprise-scale functions. For instance, Balena [4] lets users add, manage, and update IoT devices and applications from a centralized hub. AWS [2], Google Cloud [21], and Microsoft Azure [8] offer services related to MQTT-based communication, certificate-based authentication and authorization policies, and cloud-based data computation. 7SIGNAL [18] offers client-side network performance monitoring and analytics using proprietary sensors. And Laird Connectivity offers many solutions including the Sentrius RG1xx LoRa-Enable Gateway series [7], a series of programmable gateways that support multi-protocol network traffic from IoT devices.

However, prices of these solutions and services are often not cheap, making them financially inaccessible or impractical to many developers and businesses. While Balena's first ten devices are free, "prototype" solutions (20 devices) cost $99 per month, and enterprise-scale solutions can cost up to $2999 per month. Likewise, 7Signal charges thousands of dollars for even modestly sized environments. And Laird's RG1xx series can cost upwards of $700 per gateway [11]. Such prices are unfortunately not uncommon. Developing a cheap and open-source IoT framework that can accomplish the same solutions would result in a significant market shift in favor of IoT developers.

In this paper, we present a framework for seamless and secure IoT provisioning. This framework utilizes Raspberry Pi (denoted hereafter as **RPi**) computers which run Linux and can communicate over the Internet. We install the Docker containerization technology [10] onto each RPi and communicate securely through either SSH or TLS. We show that commercial IoT solutions such as an environmental sensor network can easily benefit from our framework. Docker is a containerization technology that provides isolation and portability for applications, similar to virtual machines. TLS is a cryptographic protocol which provides authentication, encryption, and integrity over a network channel through the use of X.509 certificates. And SSH is an alternative protocol for ensuring network security without the use of certificates.

Previous development workflows on small systems require uploading and testing new code on the device whenever the code is changed. Deploying many devices with the same application usually involves uploading the code to each device manually. These processes can be costly and highly time-consuming. Our framework addresses both of these problems. By utilizing RPi devices with a Linux operating system (OS) and ARM Cortex-A processors, we can develop containers which run our applications, and test them directly on our own machines. We can then cross-compile using GNU Arm Embedded Toolchain [14] for the RPi processor. Finally, we can deploy these applications concurrently by uploading them to a container registry, where our devices will then download from. This workflow pipeline significantly reduces time cost for developers.

Container-based IoT offers several advantages to developers and end users. Linux devices can be easily configured to run multiple applications simultaneously; for instance, one app collects and forwards sensor data to a server, while another app listens and responds to incoming messages from a user controller. Apps can update independently of each other or simultaneously, without any conflicts from the underlying architecture. Finally, Linux-bundled SoCs are often considerably more powerful than their counterparts and can execute much more sophisticated operations; this is particularly important for edge and fog networks which may need to simulate functions of a cloud computing environment. The Docker client simplifies these sorts of scenarios. However, Linux and containers cannot solve all IoT solutions—for example, those which require real-time performance and measurements as offered by real-time operating systems (RTOS). The extent of containerization on RPi will be further discussed in Sections II and III.

Our contributions in this paper are as follows. We first describe the requirements of IoT security from a software, network, and hardware standpoint. As previously stated, IoT security is a growing problem in the industry. Smaller developer teams rarely take the time to ensure their devices and applications are secure. In this paper, we focus primarily on network security, since we observe that many recent attacks on IoT occur at the network level. We also address software and hardware requirements. We then implement and describe our framework and a proof-of-concept IoT cluster consisting of three RPi devices which measure temperature, humidity, and CO2 levels. Our framework is written entirely in Python. We connect all devices and our host machine to a network switch and then execute the framework script, which deploys an application onto each device, connects them to WiFi, and bootstraps them for secure communication with the server. Our framework also provisions a PostgreSQL database on our server; when devices collect data, the server connects to devices over SSH to fetch the most recent data. After deployment, we can continue to update our applications by pushing new containers to the registry; our RPis will then automatically fetch and run the new containers.

The rest of this paper is organized as follows. In Section II, we evaluate IoT software, network, and hardware security requirements, and we discuss streamlined IoT development and deployment on the RPi using secure practices. In Section III, we present our framework and environmental sensors, and we methodically discuss the development and deployment process of this environment. Section IV concludes the paper.

II. Linux-Based IoT System Security

In this section, we identify the critical components of IoT security in terms of software development, network-layer communication, OTA updates, and hardware. According to OWASP, the most prevalent IoT vulnerabilities affect the network and software level [23]. We discuss modern methods of securing these features and how to achieve these features on the RPi.

A. *Software Security*

Software security is essential to any computer system. No developer can ever guarantee that their code is bug-free. As a result, features like process isolation and OTA updates are essential for ensuring the longevity of IoT systems. Lack of OTA capabilities has resulted in the recall of many commercial IoT products, since vendors could not deploy updates in a reliable manner.

The complexity of Linux OS presents additional attack surfaces. If an adversary can successfully mount a privilege escalation attack through vulnerabilities in the software or kernel, he can compromise the whole device. This subsumes any OTA patching strategies. In modern cloud systems, attacks like these are mitigated through the use of virtualization, which completely isolates applications from each other. In contemporary microcontrollers such as Arduino, ESP32, and ESP8266, the hardware will commonly only allocate a single, monolithic address space to the application and may either lack sufficient resources (such as memory or an MMU) to virtualize applications, or simply have no available software packages to serve the hypervisor role for these architectures, whereas software hypervisors such as VMware, VirtualBox and QEMU are readily available for x86 and ARM platforms. On these microcontrollers, which may be dual-core, it is possible to run concurrent workloads within the same application by taking advantage of different cores or threads, but importantly, there is no data isolation between these components, since the address space is shared. On the other hand, RPi provides isolation capabilities since Linux can run multiple independent processes simultaneously. Additionally, it is important not to expose the underlying Linux filesystem and processes.

We can isolate applications running on RPi through Docker containers, which provide similar functions as virtual machines but with much less overhead. Figure 1 illustrates the architectural differences between containers and virtual machines. Containers bundle all necessary binaries and dependencies into a single package for portability. In addition, containers running on the same device share the same kernel, removing the need for guest OS provisioning. This greatly reduces the size of containers, which is important for IoT devices. Finally, Docker containers ensure process isolation through namespaces and control groups, which are inherent to Linux.

Docker provides security and simplicity to app deployment and OTA updates for IoT. Docker can be installed on any Linux system, while application-specific dependencies are bundled into the container itself. The app deployment workflow using containers is described below.

1) **Develop**: Developers create, test, and debug their apps on the development machine first.
2) **Build**: Developers create and build a Dockerfile to package the app into a container image. Cross-compilation occurs in the Dockerfile.

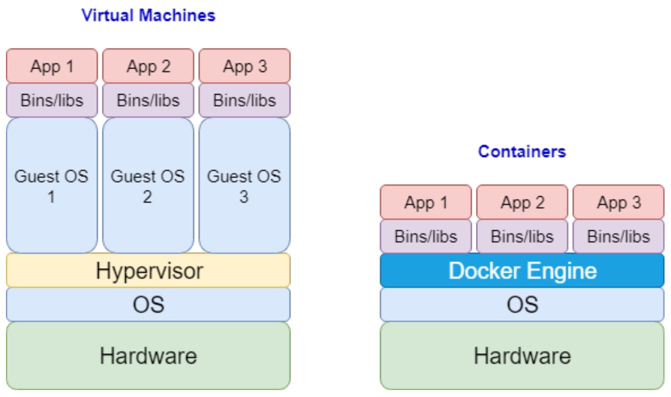

Fig. 1. Comparison between virtual machines and containers.

3) **Push**: Developers upload the image to a central repository.
4) **Deploy**: IoT devices autonomously download and run the newest image from the repository. Old images are removed from devices.

Although apps can always have software-level vulnerabilities such as buffer overflows, SQL injection, and DoS attacks, Docker simplifies the OTA process since patches can be easily tested and verified in the development environment prior to deployment. Additionally, Docker provides a ubiquitous programming interface for all Linux distributions. Traditional OTA deployments in IoT involve unique configurations for different devices; however, containers alleviate hardware or OS-bound dependencies.

Container images also consist of layers as defined by the Dockerfile, so only updated layers will be pulled by devices. This reduces the download size and network bandwidth of devices, resulting in fast concurrent OTA updates on many devices.

B. Network Security

Protection of network traffic is equally important to IoT security. If left unprotected, attackers can sniff, steal, or manipulate data in transit from one device to another. For instance, sensitive data from embedded medical equipment can be altered in a man-in-the-middle (MITM) attack to falsely report a patient's health status. In an environmental monitoring system as presented in this paper, temperature, humidity, and CO2 values can be altered to report dangerously high or low levels. As such, it is necessary to provide adequate security to an IoT network.

Conventional network security is accomplished through SSL/TLS (also known simply as TLS), which is a standardized protocol that relies on public key cryptography, symmetric key cryptography, and X.509 certificates to provide **mutual authentication**, **confidentiality**, and **integrity** to a network.

With **mutual authentication**, two devices (e.g. a RPi and server) authenticate to each other in what is known as the *TLS handshake*. TLS authentication is typically accomplished with X.509 certificates, while the SSH protocol uses public keys. Any Linux platform can generate an X.509 certificate through the OpenSSL library, which is an open-source implementation of the TLS protocol. Other SoCs or microcontrollers may require developers to generate certificates externally and then copy the PEM file over to the device's executable code.

TLS requires a trusted third party, the certificate authority (CA), to sign certificates with its own private key prior to the handshake. If an RPi receives a certificate from a server that is signed by an untrusted CA (or not signed at all), the RPi will reject the certificate and authentication will fail. The same scenario applies when the RPi authenticates to the server.

We choose to generate an ECDH key pair with a 256-bit key size. The cryptographic strength is equal to an RSA key size of 3072 bits. A smaller key size is crucial in reducing time cost of public key operations, which are computationally intensive compared to symmetric key operations. It has been observed that ECC operations perform faster and consume less power than RSA operations of equal strength [3]. This is particularly important in the case of the RPi, which has no hardware acceleration of cryptographic operations.

Mutual authentication is necessary to ensure protection against several attacks. Without device authentication, attackers may spoof the client and request security credentials, or send malicious/falsified data to the server. Without server or user authentication, anyone can collect information from the device.

For **confidentiality**, data packets are encrypted using a shared secret key. This prevents attackers from obtaining sensitive information, since data cannot be decrypted without the secret key. The TLS handshake involves both mutual authentication and secret key establishment. The parameters of the secret key are negotiated at the beginning of the handshake. For the AES algorithm, a 256-bit key in cipher block chaining (CBC) mode is generally considered sufficient.

Finally, **integrity** ensures that data cannot be modified while in transit. To accomplish this, data packets are appended with a hash-based message authentication code (HMAC), which is calculated using the original message, the secret key, and some hash function. Since both devices have the secret key, both can calculate the message digest. If the message receiver finds that the message digest does not match its own calculated digest, this indicates that the message was altered and should be discarded. A common HMAC configuration is AES-256 with SHA-256.

C. Hardware Security

Hardware-enforced security is also important for protecting sensitive data. At a minimum, all IoT technology should implement *flash encryption* and *secure boot* in some form. Currently, external hardware [27] is required for the RPi to implement flash encryption or secure boot, as the base SoC does not support either function.

Flash encryption prevents data from being leaked in the event that a device is compromised. This is done by encrypting data or files with a secret key and decrypting them only when needed by the processor. The secret key should be stored in hardware and restricted from read or write access. Many embedded devices like the ESP32 have one-time programmable memory such as *eFuse* memory, whose data contents cannot be modified more than once. Read access to this data is also restricted by the hardware. Flash encryption implementation on the RPi would require an external chip to provide encryption support. For instance, the ATECC608A [17] and ATAES132A [15] are security chips that support AES-128 hardware key storage and acceleration.

Secure booting prevents attackers from modifying the firmware or uploading their own firmware to a device. Firmware and bootloader images are appended with digital signatures which are easily verifiable using public keys. If the digital signature fails to validate, then the device refuses to boot. The private key is stored in hardware and inaccessible to software. ECDSA is a common algorithm used for producing and validating a digital signature. Again, external chips are needed for implementation on the RPi. As in the case of flash encryption, secure boot keys can either be stored in secure memory or through external chips like the ATECC508A [16] and ATECC608A.

Hardware security for IoT is an ongoing field of research and development. ARM's TrustZone technology [13] splits the system into a normal OS and a secure OS, wherein processors can switch between the two as needed. The secure OS provides confidentiality and integrity to the environment through a hardware root of trust. Trusted Platform Module (TPM) [12] is a chip that uses an immutable key pair to provide a root of trust to a bootloader sequence as well as software encryption. It is widely adopted in Microsoft systems as well as some UEFI implementations. It is anticipated that more IoT-focused SoCs will continue to adopt these technologies as the industry evolves.

D. Streamlined IoT Deployment

Modern software packages for IoT deployment are proprietary and expensive. We show that open-source technologies and standards like Docker, TLS, and Linux provide a feasible solution to this problem; this limits costs only to the hardware and server overhead.

Inter-dependency of developers, devices, and end-users is needed in any IoT system. In a Docker-based system, developers can deliver updates to their devices through a Docker registry, while devices can be configured to pull the newest container images from the registry on a scheduled basis. Devices can then proceed to collect environmental data and send them to a remote server. The server forwards data to end users, stores it in a back-end database, performs big data, machine learning, or some other analysis on the received data in some capacity.

Figure 2 illustrates the inter-dependency of developers, devices, and end-users in a Docker-based IoT system. Developers can deliver updates to their devices through a Docker registry, while devices can be configured to pull the newest container images from the registry on a scheduled basis. Devices can proceed to collect environmental data, then send this data to a remote server. The server forwards this data to end users and, optionally, stores the data into a database or acts on the data in other manners. The details of each step are further elaborated below. Authentication requirements are handled in the initial provisioning stage.

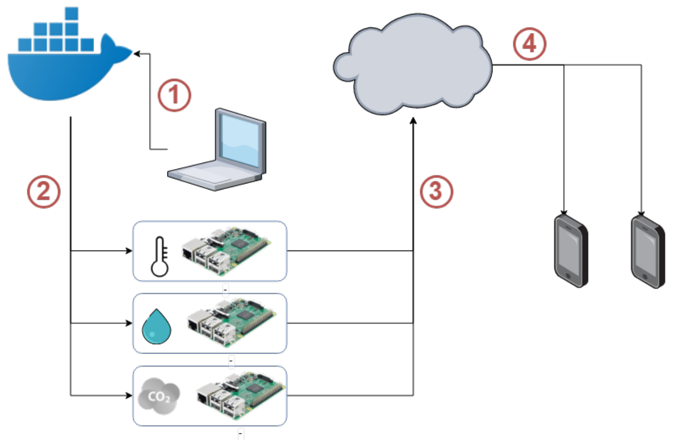

Fig. 2. Flowchart of a standard IoT environment with Docker.

The details of each step are elaborated as follows. In **step 1,** developers build their app and containerize it, then push the image to a registry. In **step 2**, devices pull and execute the new container image from the registry. In **step 3,** devices collect environmental data and send this data to a remote server. In **step 4,** the server delivers the data to end users through a web app or smart phone app.

In **step 1,** developers work in their environment of choice; the only requirement is that Docker be present on the system. Once ready, developers containerize their project and push the image to a Docker registry. Mutual trust is required between the developer system and the registry. Docker Hub, the official registry by Docker, enforces a TLS connection. This connection employs mutual authentication through client-side credentials and a server-side certificate. Developers can optionally deploy their own private registry and configure TLS in a similar fashion.

In **step 2,** devices pull and execute the new container image from the Docker registry. Developers can add daemons to the devices which perform image pulls on a regular basis. Mutual trust is required between the registry and each device. TLS configuration is identical to step 1. The server can provide a X.509 certificate to authenticate to the client, while devices must present credentials for client-side authentication.

In **step 3,** devices collect environmental data using sensors and send this data to a remote server. Message Queuing Telemetry Transport (MQTT) over TLS is an appropriate protocol for this. MQTT is a lightweight messaging protocol based on a publish-subscribe model. Mutual authentication is required between the devices and the server. Both the clients and the server will authenticate through TLS certificates.

Finally, in **step 4,** the server delivers the data to end users, possibly through a smart phone app or a web app. End users must prove ownership of a device when requesting data on that device. There are several solutions to this. One solution is that each device comes programmed with a random, unique ID and key that can be accessed through an ad-hoc web server. Registered users can provide these credentials to the server in order to register the device under their account. Then, when the server receives data from a device, it also receives the ID and key, and it can look up the account which owns the device and store this information in a database.

Mutual authentication and data protection can be established between all involved parties. Developers can authenticate to a Docker registry with their credentials, while the registry provides a X.509 certificate for server-side authentication. RPis can perform mutual authentication with the registry in the same way. The server can authenticate with RPis via SSH or MQTT over TLS, where both protocols also provide message obfuscation and integrity. And user apps authenticate to the server through credentials, while the server can authenticate to users via TLS.

This model illustrates that a secure and streamlined IoT deployment solution is quite feasible. Network security is at the forefront, while software security can be effortlessly integrated and improved. As long as standardized protocols are used, then the cryptographic strength in network security is limited only by the entropy of the private keys. OpenSSL uses software-defined random number generators which are considered sufficient for most purposes. While software security is still limited by the developer app, vulnerabilities and bugs can be quickly patched via container-based OTA updates. Furthermore, system-wide software exploits are greatly reduced due to container isolation.

III. Implementation

In this section, we describe our implementation of the framework and features discussed in Section II. Development takes place on Ubuntu 16.04, which has native Docker support. This framework will provision a cluster of three RPi devices that collect temperature, humidity, and sensor data from the environment and send this data to a remote PostgreSQL database. Data is then aggregated for display on a web server.

The framework is categorized into two stages: the *provisioning stage* and the *deployment stage*. In the *provisioning stage*, the RPis and provisioning computer are connected physically via a network switch. The provisioning computer copies configuration files over to each RPi. Communication between RPis and the server is established. In the *deployment stage*, RPis fetch and execute containerized apps as specified in the configurations. RPis collect and forward data to the server. Developers can continue to push app updates to the Docker registry.

In our laboratory, we were restricted to use the development platform, remote server, and end user device as the same machine. In a genuine IoT system, these systems are all different. However, for this framework, we focus primarily on the developer workflow rather than end user specifications and requirements. We plan to address this subject in future research.

TABLE I
An overview of expected costs for the framework.

Component	Cost	Quantity/Rate
RPi 3B	\$35	x3
Netgear router	\$60	x1
Ethernet cable 2-pack	\$5.99	x2
Temp/humidity sensor	\$9.95	x1
Air quality sensor	\$5.49	x1
Server	\$20	Monthly

Furthermore, in the current model, the RPis and the server communicate using SSH rather than MQTT. Like TLS, SSH provides mutual authentication, confidentiality, and integrity to a communication channel, but it lacks the presence of certificates and a CA. Although SSH provides security through public key cryptography, it presents a single point of failure, since compromise of the server would also compromise all devices. We plan to replace SSH completely with MQTT in future research.

We also not use real-world sensors to collect data in our experiment, due to restrictions in our lab environment. Instead, RPi applications generate random data corresponding to temperature, humidity, or CO2, and forward this data to the server. This does not alter the effectiveness of the model.

A. Components

We now provide an overview of the hardware and software components utilized in the framework. We list the cost of each hardware component, including server costs, while all software is currently free. The total estimated costs of our framework are summarized in table I. Note that while genuine sensors are not utilized in this experiment, we still report their costs for clarity to the reader.

1) Hardware: Our setup consists of three RPis with HypriotOS installed, a network switch, and four ethernet cables. Figure 3 shows the usage of these components during the provisioning stage.

For RPi, we use the Raspberry Pi 3 Model B, which retails for \$35 from Adafruit and contains built-in WiFi and Bluetooth connectivity. The RPi boasts a quad-core 1.2 GHz CPU, 1 GB RAM, a 40 pin GPIO set, and microSD storage. The RPi is powerful compared to some other SoCs, especially those focused on real-time computing. This makes it a good candidate for running several consumer-focusedb applications on one device.

We use a Netgear AC1200 dual-band WiFi router to connect devices during the provisioning stage. The router retails for \$60 and can connect up to four devices. The network switch is necessary to guarantee trust between the provisioning computer and each RPi.

Finally, we use four ethernet cables to connect each RPi and the provisioning computer to the network switch, allowing for communication between the four devices. A two-pack of Ethernet cables can be purchased on Amazon for \$5.99.

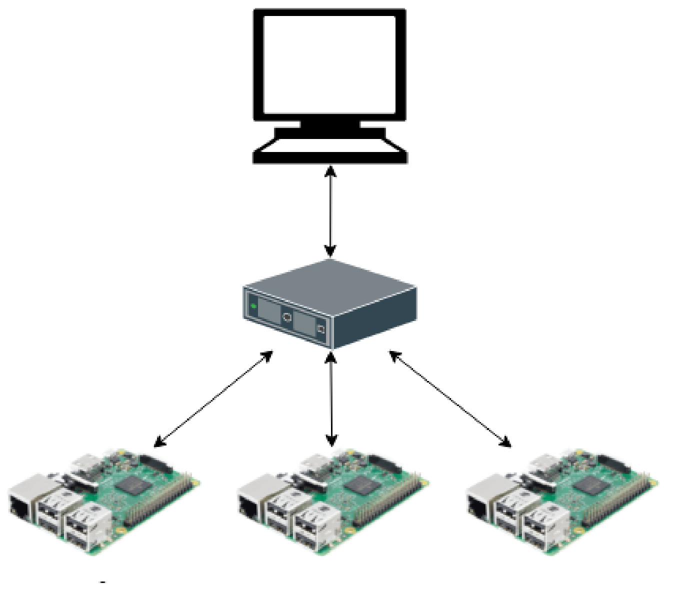

Fig. 3. Initiation schematic between provisioning computer and RPIs.

To collect data, a variety of temperature and air quality sensors are available for low prices. For example, the Adafruit DHT22 [19] is a single-wire temperature and humidity sensor available for \$9.95. And the HiLetgo MQ-135 Air Quality Sensor [24] is a hazardous gas sensor available from Amazon for \$5.49.

In total, our setup of three nodes costs approximately \$186. Meanwhile, server costs may vary anywhere from \$5 per month and up. We conservatively assume that server upkeep will cost developers \$20 per month. This estimate compares favorably to similar IoT deployment solutions. In addition, costs may be further reduced by using cheaper RPi models; for instance, the RPi Zero retails for \$5 and can be paired with an external WiFi dongle to save on costs.

2) Software: We use a variety of software services to provision and deploy our environmental cluster, including Docker, Python, OpenSSH, and HypriotOS. Our development platform runs on Ubuntu 16.04.

We use Docker version 18.04 in our setup. This must be installed on the developer computer and the RPIs to allow for OTA delivery. Dockerfiles must be carefully crafted to cross-compile an app for the ARM architecture delivering them to the Docker registry. Furthermore, in this setup, the Docker registry is assumed to be Docker Hub.

We use the Debian-based HypriotOS 1.7.1 for the RPi OS. This requires flashing each Micro SD with the OS prior to the provisioning stage. HypriotOS is desirable since it is more lightweight than Raspbian and comes prepackaged with Docker. HypriotOS is also relatively lightweight and has an open SSH port with default credentials; this is important for the provisioning stage, since the provisioning device will communicate with RPIs over SSH.

The provisioning script is written in Python 3.6. We use three external Python packages, Fabric, Psycopg2, and Py-Serial, in assistance with our setup. Fabric implements SSH functionality into Python, allowing for remote device access and file transfers. Furthermore, standard output can be saved as variables inside Python. Psycopg2 is a PostgreSQL adaptor for Python, allowing our server to communicate with the database using Python. Finally, PySerial allows communication with the system serial ports.

Finally, we use OpenSSL to generate an ECC NIST P-256 key pair. The private key remains on the server indefinitely, while the public key is copied to each RPi for remote access. As discussed previously, devices communicate with the server over SSH in this demo. The server can use its private key when attempting to authenticate to a RPi, while RPIs can authenticate by providing a fingerprint of the public key to the server, as part of the standard SSH protocol. We require that only the server can authenticate into clients, as client-side authentication would pose many security risks for the server.

B. Framework Provisioning

Our framework initially requires that at least one Docker image has been defined and uploaded to a registry and, moreover, that RPis can access this image either publicly or through credentials. This means the app must be at least partially developed and that cross-compilation is done correctly. Moreover, provisioners are required to define three configuration files, namely a 'public' file, a 'private' file, and an 'app' file. Tables II shows what options are required for each file. Both requirements are detailed below.

The first step is to compile and upload at least one Docker image to the Docker registry. Consequently, this requires that the app itself has been at least partially developed and cross-compiled to work on a RPi. By default, containers restrict access to all external interfaces, including serial ports. These interfaces can be accessed by containers by passing the "–device" argument to the Docker run command, which is needed to access sensor data.

Next, we define the three aforementioned configuration files. These text files are parsed line-by-line, where each line corresponds to a different parameter. The public file defines WiFi preferences, other file locations, and update preferences. Developers can identify the IP addresses of their RPis to avoid potentially malicious devices. Developers can link an app with a RPi by associating an app configuration file with a RPi's IP address. The private configuration file contains sensitive information such as WiFi credentials and Docker registry credentials. Therefore, only privileged users should execute the provisioning script. An administrator can enforce this by restricting configuration access to sudoers, group members, etc. The app configuration file labels the data produced by the RPi and defines Docker runtime behavior. For instance, temperature sensor data may be labeled 'TEMPERATURE'. This metadata is sent to the PostgreSQL database when the server fetches the RPi's data.

In addition, we define three watchdog scripts which run as cronjobs on the server and RPi. The first watchdog lives on the

TABLE II
A SUMMARY OF OPTIONS FOR EACH CONFIGURATION FILE.

Tag	File	Description
SECRET-CONFIG	Public	Points to secret config file
WIFI	Public	If yes, RPIs connect to WiFi
INTERFACE	Public	Ethernet interface SSID
DEVICES	Public	Whitelist RPi IPs
DEVICE-CONFIG	Public	Links an RPi to an app config file
WATCH-SERVER-DIR	Public	Points to server watchdog directory
WATCH-RPI-DIR	Public	Points to RPis watchdog directory
DOCKER-PULL	Public	Frequency of container updates
DATA-PULL	Public	Frequency of data fetching from RPIs
PACKAGE-UPDATE	Public	Frequency of RPi software updates
WIFI-SSID	Private	The WiFi SSID
WIFI-PSK	Private	The WiFi password
SERVER-KEY	Private	Points to the server's private SSH key
DOCKER-USERNAME	Private	Docker registry username
DOCKER-PASSWORD	Private	Docker registry password
DOCKER-IMAGES	App	Lists Docker images for this RPi
DOCKER-RUN	App	Defines 'Docker run' commands
DATA	App	Label the data produced by the RPI

Algorithm 1 Device Provisioning

```
procedure PROVISION($public\_config\_file$)
    if $user \neq privileged$ then exit()
    end if
    parse($public\_config\_file$)
    parse($private\_config\_file$)
    install($python$, $openssh$, $pip$)
    $devices \leftarrow scan(interface)$
    $(pub\_key, priv\_key) \leftarrow generate\_keypair()$
    server.store($priv\_key$, $server\_watchdog$)
    for $i \leftarrow 1, devices.size$ do
        $d \leftarrow devices[i]$
        d.store($pub\_key$)
        d.store($docker\_watchdog$)
        d.store($host\_watchdog$)
        d.store($app\_config$)
        d.disable_password_ssh()
        d.connect_to_wifi($wifi\_credentials$)
        d.login_to_docker_registry($registry\_credentials$)
        d.docker_pull_and_run($app\_config$)
    end for
end procedure
```

server. When active, it fetches data from the RPis and forwards them to the database. The second and third watchdogs are copied to each RPi. The second watchdog updates Docker containers with the app configuration file that lives on each RPi, while the third watchdog updates software packages on the host itself. The execution frequency and location of the watchdogs are defined in the public configuration file.

The final step is to execute the provisioning script. Algorithm 1 provides pseudocode for the script. We pass in the public configuration file as a parameter, since this file can locate the other configuration files and the watchdog scripts. The configuration options are saved as variables in the program. Then, the software dependencies are installed onto the provisioning device; Python and OpenSSH are installed through apt-get, while Python dependencies are installed through Pip. Next, the Ethernet interface is scanned for device IP addresses, which are cross-referenced with the whitelisted IPs given by the public configuration file. The approved IPs are saved. Next, an ECC NIST P-256 key pair is generated using OpenSSH, and the private key is stored on the server. Recall that the server and provisioning device are the same computer; in a production environment, the key pair would instead be generated on the server directly, and the public key would be transported to the provisioning computer. Afterwards, the server watchdog is placed on the server, and the script configures a cronjob to automatically execute the watchdog on regular intervals.

The last step is to configure each RPi to run its app and communicate with the server. We start by copying the public key over to the RPi, allowing the server to SSH into the device. The device watchdogs and app configuration file are also copied over, and cronjobs are created to execute the watchdogs. Password-based SSH is disabled in the SSH daemon settings so that only key-based SSH access is allowed. The WiFi credentials are then used to login to the WiFi network. Finally, the RPi uses the Docker registry credentials to login to Docker, before using the app configuration settings to download and execute the app.

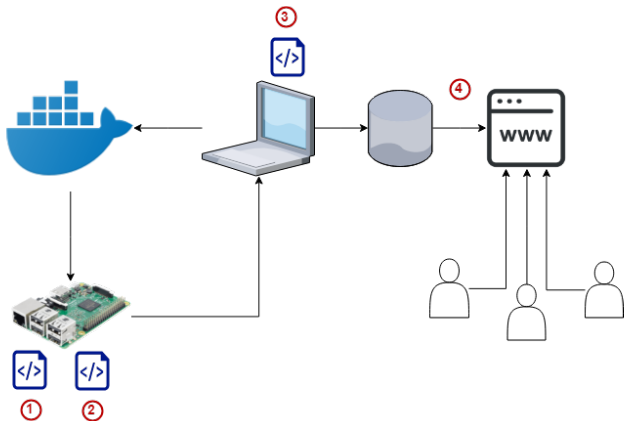

Fig. 4. Flowchart of the IoT environment prepared for this paper.

The script terminates once all devices have been provisioned. Note that the database and end-user application are not discussed in the provisioning stage. It is expected that the server will be configured separately to forward sensor data to the database and serve end-user content. The purpose of the framework we offer is to streamline the app provisioning and development cycle of IoT environments; end-user requirements are a topic for future research.

C. Development Workflow

To demonstrate the feasibility of our framework, we have developed a proof-of-concept web server that displays data from our PostgreSQL database. The development workflow is illustrated once again; developers push images to the Docker registry, which are then pulled and executed by the RPis through their watchdog and configuration files. The server connects to each RPI and collects their sensor data. Also shown is the role in which the database and web server play in the framework, which are also hosted on the server.

Figure 4 depicts a flowchart of our environmental setup. We designate four points of interest. **Points one and two** refer to the two watchdogs that belong to each RPi. **Point three** refers to the server watchdog. Finally, **point four** shows the data pipeline. End-users can login to the publicly exposed web interface to view sensor data in near-real-time. Figure 5 shows the web interface displaying temperature data from the sensors.

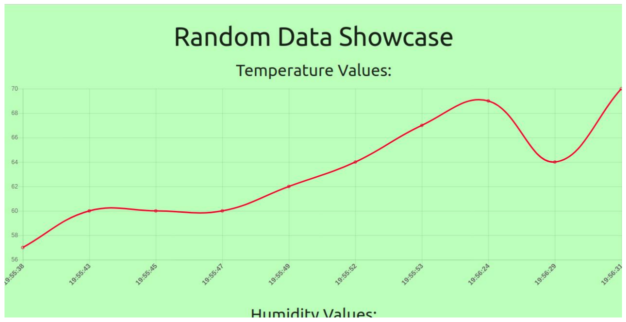

Fig. 5. Web server displaying recently collected temperature values.

IV. Conclusion

In this paper, we investigate the problematic trends in IoT security and deployment solutions, and we develop a secure IoT deployment model which uses Docker, key-based authentication, and RPis running open-source software to create a proof-of-concept environmental monitoring system which models real-world products. This project demonstrates that network and software vulnerabilities in the IoT market can be easily addressed with standardized, open-source protocols and technologies. Furthermore, our model shows that the developer workflow can be fully streamlined through containerized OTA updates. We have shown that a scalable, affordable IoT deployment framework is achievable. Our future work will include end user authentication, NoSQL support, MQTT/TLS communication, incorporation of hardware security, and integration with cloud services such as AWS.

Acknowledgement

This work was supported in part by the National Science Foundation under Grant No. 1915780. Any opinions, findings, conclusions, and recommendations in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] R. Alharbi and D. Aspinall. An iot analysis framework: An investigation of iot smart cameras' vulnerabilites. In *Living in the Internet of Things: Cybersecurity of the IoT - 2018.*, Mar. 2018.
[2] I. Amazon.com. What is aws? https://aws.amazon.com, 2019.
[3] M. Bafandehkar, S. M. Yasin, R. Mahmod, and Z. M. Hanapi. Comparison of ecc and rsa algorithm in resource constrained devices. In *Proceedings of the 3rd International Conference on IT Convergence and Security (ICITCS)*, Dec. 2013.
[4] Balena. Build your iot project with balena. https://www.balena.io/, 2018.
[5] J. Biggs. Hackers release source code for a powerful ddos app called mirai. https://techcrunch.com/, 2016.
[6] L. Columbus. Iot market predicted to double by 2021, reaching $520b. https://www.forbes.com/sites/louiscolumbus/2018/08/16/iot-market-predicted-to-double-by-2021-reaching-520b/, Aug 2018.
[7] L. Connectivity. Wireless gateways. https://connectivity.lairdtech.com/, 2019.
[8] M. Corporation. Microsoft azure cloud computing platform & services. https://azure.microsft.com/en-us/, 2019.
[9] J. Deogirikar and A. Vidhate. Security attacks in iot: A survey. In *2017 Internation Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC)*, Feb. 2017.
[10] I. Docker. Enterprise container platform. https://www.docker.com/, 2019.
[11] D.-K. Electronics. Laird - wireless & thermal systems 450-0190. https://www.digikey.com/, 2019.
[12] I. O. for Standardization. Information technology—trusted platform module library—part 1: Architecture (iso/iec standard no. 11889-1. https://www.iso.org/standard/66510.html, Aug. 2015.
[13] A. Holdings. Trustzone. https://developer.arm.com/technologies/trustzone, 2017.
[14] A. Holdings. Gnu arm embedded toolchain. https://developer.arm.com/, 2019.
[15] M. T. Inc. Ataes132a. microchip.com/wwwproducts/en/ATAES132A, 2016.
[16] M. T. Inc. Atecc508a. https://www.microchip.com/wwwproducts/en/ATECC508A, 2017.
[17] M. T. Inc. Atecc608a. https://www.microchip.com/wwwproducts/en/ATECC608A, 2017.
[18] S. S. Inc. 7signal - the leader in enterprise wireless network monitoring. https://7signal.com/, 2019.
[19] A. Industries. Dht22. https://www.adafruit.com/, 2012.
[20] A. Kamble and S. Bhutad. Survey on internet of things (iot) security issues & solutions. In *2018 2nd International Conference on Inventive Systems and Control (ICISC)*, Jan. 2018.
[21] G. LLC. Google cloud platform. https://cloud.google.com.
[22] C. Osborne. Fda issues recall of 465,000 st. jude pacemakers to patch security holes. https://www.zdnet.com/, 2017.
[23] T. O. W. A. S. Project. Owasp internet of things project. https://www.owasp.org/, 2019.
[24] L. Shenzhen HiLetgo Technology Co. Mq-135 mq135 air quality sensor hazardous gas detection module for arduino avr. http://www.hiletgo.com, 2018.
[25] I. Thompson. Wi-fi baby heart monitor may have the worst iot security of 2016. https://www.theregtister.co.uk/, Oct. 2016.
[26] K. Zetter. Flaw in home security cameras exposes live feeds to hackers. https://www.wired.com/, Feb. 2012.
[27] Zymbit. Security module for raspberry pi. https://www.zymbit.com/zymkey/, 2018.