Journal of Network and Computer Applications 217 (2023) 103699

Contents lists available at ScienceDirect

Journal of Network and Computer Applications

journal homepage: www.elsevier.com/locate/jnca

PARIOT: Anti-repackaging for IoT firmware integrity

Luca Verderame [a,*], Antonio Ruggia [a], Alessio Merlo [b]

[a] DIBRIS - University of Genoa, Via Dodecaneso, 35, I-16146, Genoa, Italy
[b] CASD - Centre for Higher Defence Studies, Piazza della Rovere, 83, I-00165, Rome, Italy

ARTICLE INFO

Dataset link: https://github.com/Mobile-IoT-Security-Lab/PARIOTIC

Keywords:
IoT repackaging
IoT security
IoT firmware update
Firmware
Internet of things

ABSTRACT

IoT repackaging refers to an attack devoted to tampering with a legitimate firmware package by modifying its content (e.g., injecting some malicious code) and re-distributing it in the wild. In such a scenario, the firmware delivery and update processes are central to ensuring firmware integrity.

Unfortunately, several existing solutions lack proper integrity verification, exposing firmware to repackaging attacks. If this is not the case, they still require an external trust anchor (e.g., signing keys or secure storage technologies), which could limit their adoption in resource-constrained environments. In addition, state-of-the-art frameworks do not cope with the entire firmware production and delivery process, thereby failing to protect the content generated by the firmware producers through the whole supply chain.

To mitigate such a problem, in this paper, we introduce PARIOT, a novel self-protecting scheme for IoT that injects integrity checks, called anti-tampering (AT) controls, directly into the firmware. The AT controls enable the runtime detection of repackaging attempts without needing signing keys, internet connection, secure storage technologies, or external trusted parties. PARIOT can be adopted on top of existing state-of-the-art solutions ensuring the widest compatibility with current IoT ecosystems and update frameworks. Also, we have implemented this scheme into PARIOTIC, a prototype to protect C/C++ IoT firmware automatically. The evaluation phase of 50 real-world firmware samples demonstrated the proposed methodology's feasibility and robustness against practical repackaging attacks without altering the firmware behavior or severe overheads.

1. Introduction

The Internet of Things (IoT) paradigm enables the growth of low-cost embedded devices with network connectivity and real-time capabilities that are now used in many verticals, from logistics to precision farming and smart homes. Each IoT device is equipped with firmware, i.e., a bundle that contains all the software needed to ensure the functioning of the device hardware. Typically, the firmware comprises a fully-fledged IoT operating system (like RIOT Baccelli et al., 2013 or Contiki Dunkels et al., 2004) and at least an application that holds the core functionalities of the thing.

During the building phase, the device manufacturer equips the IoT device with the first version of the firmware. However, the functionalities an IoT device requires at deployment time will likely change. To this aim, the firmware will need frequent updates for several reasons: to offer additional functionalities, support new communication protocols, and patch software bugs (including security vulnerabilities).

As firmware has a central role in the life cycle of an IoT device, its security has raised serious concerns from the scientific and industrial community. To this aim, several works were proposed to evaluate the security of the firmware bundle (e.g., David et al., 2018 and Costin et al., 2014 or Costin et al., 2016), enforce the update mechanisms (e.g., Dejon et al., 2019; Cui et al., 2013 and Langiu et al., 2019), or patch existing firmware bundles to cope with end-of-life, vulnerable images. (e.g., Carrillo-Mondéjar et al., 2022; Maroof et al., 2022, and Christensen et al., 2020).

In particular, the integrity of the firmware delivered through an update process represents a major security threat, as witnessed by many real-life examples. For instance, the PsycoB0t (Dronebl, 2008) was the first router botnet that altered the firmware of approximately 85,000 home routers and resulted in large-scale denial of service attacks. Also, the Zigbee Worm (Ronen et al., 2017) triggered a chain reaction of infections, initialized by a single compromised IoT device (light bulb), using a malicious firmware update image.

An attacker can retrieve the firmware differently, such as obtaining it from the vendor's website or community forums, sniffing the OTA update mechanism, or dumping it directly from the device (Gupta, 2019). Once the original firmware is obtained, the attacker can analyze it through reverse engineering techniques to extract sensitive information such as encryption keys, hard-coded credentials, or internal URLs. Thanks to such knowledge, an attacker can craft a modified version

* Corresponding author.
E-mail addresses: luca.verderame@dibris.unige.it (L. Verderame), antonio.ruggia@dibris.unige.it (A. Ruggia), alessio.merlo@casd.difesa.it (A. Merlo).

https://doi.org/10.1016/j.jnca.2023.103699
Received 30 January 2023; Received in revised form 28 April 2023; Accepted 4 July 2023
Available online 8 July 2023

of the firmware and try to re-distribute it in the wild as if it was the original one (Mtetwa et al., 2019).

This type of attack called *repackaging*, is well-known in the mobile ecosystem, where attackers alter and re-distribute thousands of Android and iOS applications (Merlo et al., 2021). Unfortunately, such a security threat is barely considered in the IoT ecosystem, especially in low-end devices where resource constraints limit the applicability of state-of-the-art mitigation techniques such as remote attestation or signature verification.

Furthermore, many of the existing solutions for low-end IoT devices focus only on some parts of the delivery process (e.g., from the update server to the device) or do not perform a proper verification of the downloaded firmware and hence cannot ensure its integrity. For instance, Sparrow (SICS, 2018) (used by Contiki) only verifies the CRC of the image to detect errors during transmissions.

On the other hand, recent firmware update solutions like SUIT (Internet Engineering Task Force (IETF), 2018) or UpKit (Langiu et al., 2019) need to have an additional trust anchor (e.g., a signing certificate on the IoT device) or dedicated hardware, like a Trusted Execution Environment (Asokan et al., 2018), to allow verifying the integrity of the image. Nevertheless, the impairment of the supply chain or the delivery mechanism (e.g., the update server or the companion mobile app) could allow an attacker to inject a crafted firmware into the delivery pipeline, such as the firmware modification attack reported on commercial fitness trackers (Shim et al., 2017).

In this work, we investigate the impact of repackaging attacks on IoT firmware, thereby discussing security threats that harm its integrity. Also, we systematically review the integrity protection mechanisms used by state-of-the-art solutions for IoT firmware updates, unveiling their need to rely on signing keys, internet connection, or trusted external entities to cope with the integrity of firmware bundles.

Then, we present PARIOT, a novel self-integrity protection mechanism for IoT firmware that automatically spots altered firmware images without any prerequisite to ensure the integrity of the firmware image or modification to existing firmware delivery infrastructure.

Briefly, PARIOT focuses on inserting encrypted detection nodes (called *Cryptographically Obfuscated Logic Bombs* Zeng et al., 2018) that embed integrity checks on the content of the firmware. These checks are known as *anti-tampering* (AT) controls. The detection nodes are triggered during the execution of the firmware, and if some tampering is detected, the firmware is usually forced to crash. The rationale is to discourage the attacker from repackaging if the likelihood of building a working repackaged firmware is low.

PARIOT aims at applying the protection scheme on the source code during the building pipeline to ensure the widest compatibility with state-of-the-art software update methods (e.g., SUIT) and at minimizing the use of invasive procedures (e.g., binary rewriting Wenzl et al., 2019) that may harm/brick the firmware image and, ultimately, the IoT device.

To experimentally evaluate the feasibility of PARIOT, we implemented it in a tool for C/C++ firmware (i.e., PARIOTIC) that is compatible with existing firmware update solutions. We tested PARIOTIC in a RIOT-based IoT ecosystem with the SUIT update framework and 50 real-world firmware samples. The tool – publicly available on GitHub (Computer Security Laboratory, 2022) – achieved a 90% of success rate and required – on average – only 64.2 *s* per firmware to introduce the protections. We performed the runtime evaluation on an *iotlab-m3 board* hosted by the FIT IoT-LAB testbed (Adjih et al., 2015) to verify the resource consumption overhead (in terms of current, voltage, and power) introduced by the protection on real IoT devices. Moreover, we assessed the reliability of PARIOT by testing the repackaging detection capabilities of the protected firmware at runtime. The preliminary results showed that our solution ensures high compatibility with existing firmware generation and delivery processes, a low resource usage overhead, and a high detection rate of repackaging attacks.

Structure of the paper. In the rest of the paper, we first introduce the firmware production and delivery process (Section 2). Then, we focus on the integrity threats concerning the previous steps (Section 3). Section 4 presents the concept of firmware repackaging in the IoT ecosystem and discusses state-of-the-art anti-repackaging techniques. In contrast, Section 5 highlights the limitations of state-of-the-art approaches.

In Section 6, we describe the PARIOT protection scheme, its distinguishing features, and its runtime behavior. Moreover, we provide an implementation of the methodology (PARIOTIC) in Section 7. Section 8 analyzes the results obtained by applying PARIOTIC on 50 real-world firmware in a RIOT-based IoT ecosystem. Finally, in Section 9, we conclude the paper by summing up the main takeaways and putting forward some considerations for future works.

2. Firmware production and delivery process

Fig. 1 summarizes a typical firmware production and delivery process. The first steps are devoted to the production of the firmware (*generation phase*), i.e., the building of the software bundle containing all the software that ensures the functioning of the IoT device.

The firmware supply chain – even for a relatively simple, single-processor device – consists of many software providers, including chip and tool vendors and companies that provide different software components. In the case of Fig. 1, the software supply chain comprises three actors providing the OS, the device drivers, and the application, which delivers device core functionalities, respectively.

The different pieces of software are then composed by the *Firmware Manufacturer* (FM), which generates the firmware image and some metadata information, like a digitally-signed manifest file, used to evaluate the success of the delivery phase. At the end of the generation phase, the firmware is released on a centralized repository (e.g., an IoT Firmware Update Server).

The firmware delivery process (*distribution phase*) can occur either *manually* or by employing an *automated firmware update* process. In the first case, users get the firmware from the firmware repository and distribute it to the IoT devices by using over-the-air (OTA) technologies (e.g., Bluetooth LE or Wi-Fi) or physical interfaces (e.g., UART or USB ports). For such a task, the user may also rely on a *Mobile Update Agent* (MUA), i.e., a companion mobile app (e.g., Samsung SmartThing[1]) that acts as a gateway between the update server and the IoT device. After receiving an update notification, the MUA downloads the software update and verifies its integrity, and – once the verification succeeds – it sends the update to the *Firmware Consumer* using low-power radio technologies. As an example of integrity control, the MUA can verify the consistency between the digitally-signed manifest file and the software package.

In the automated distribution phase, instead, the firmware is distributed to the IoT devices by using client–server architectures (e.g., Software Updates for Internet of Things — SUIT) or distributed solutions (e.g., Choi and Lee, 2020). If this is the case, the IoT Firmware update server interacts through an access point with the device.

The last step in the firmware update is the *loading* phase. An agent placed on the device (i.e., the *Firmware Consumer* - FC) receives the software bundle and the metadata and copies the updated image in the correct memory address to proceed with the installation. Such a step may involve a further verification of the correctness of the received data, e.g., through a hash check or signature verification.

3. Threats to firmware integrity

Security threats involving the integrity of firmware bundles can occur in all three stages of the production and delivery process. This section aims to provide information about the threats targeting the integrity of the firmware bundle, the phase and the entities that are

[1] https://www.samsung.com/it/apps/smartthings/.

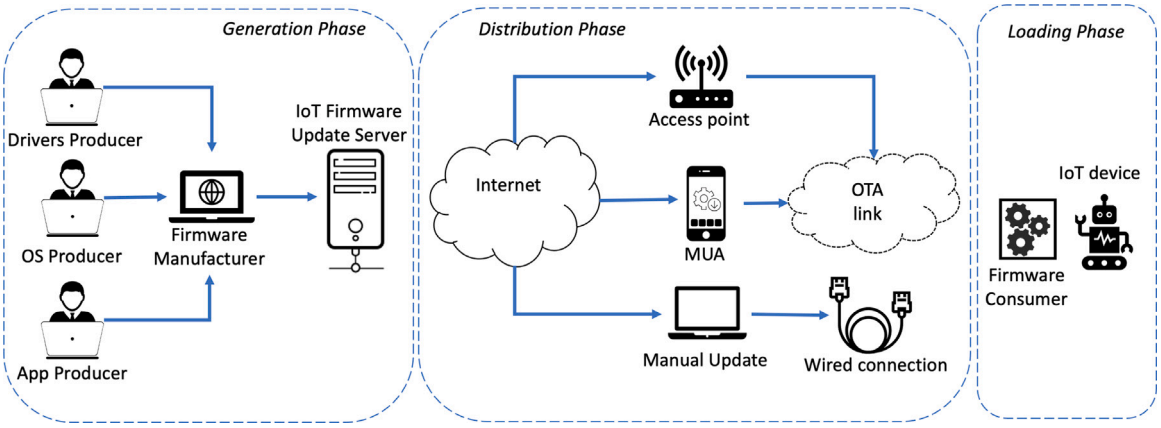


Fig. 1. Firmware production and delivery process.

affected, and the mitigation techniques and the requirements that enable to cope with those threats.

From our analysis, we identified eight distinct security threats harming the integrity of the firmware updates, with some affecting more than a phase of the firmware production and delivery process. The rest of this section provides a brief description of each of them. Table 1 reports the list of all security threats, which can be uniquely identified by a *Thread ID* (first column). To do so, we exploited the Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, and Elevation of Privilege (STRIDE) approach (Khan et al., 2017) and integrated the contributions of state-of-the-art research in the field (David et al., 2018; Gupta, 2019; Internet Engineering Task Force (IETF), 2018; Lanigan et al., 2006).

Modification of firmware before signing. If an attacker can alter the firmware bundle before it is signed (e.g., by modifying the code of one of its components) during the generation phase, she can perform all the same actions as the firmware manufacturer. This allows the attacker to deploy firmware updates to devices that trust the FM. For example, the attacker that deploys malware in the building environment of one of the Producers or the FM can inject code into any binary referenced by the bundle, or she can replace the referenced binary (digest) and URI with the attacker's ones. Possible mitigation techniques are a validation process of the firmware bundle (e.g., by enforcing vulnerability assessment and penetration testing activities) and using air-gapped building environments that are protected from external interference.

Overriding critical metadata elements. An authorized actor – but not the FM – uses an override mechanism during the generation phase to change an information element in the metadata signed by the FM. For example, if the authorized actor overrides the digest and URI of the payload in the manifest file, she can replace the entire payload with another - properly crafted -one. To mitigate this threat, the firmware update process should enforce mandatory access control-like mechanism by using access control lists with per-actor rights enforcement of the FM and the different producers (Internet Engineering Task Force (IETF), 2018).

Compromission of the intermediate agents. If an attacker succeeds in compromising an agent in the distribution phase, then she can inject a malicious/modified firmware bundle into the distribution chain. For instance, a malicious actor can compromise the MUA to replace the original firmware with a modified version after the firmware verification phase, thus installing a modified firmware on the IoT device: Schüll (2016) showed a new firmware modification attack against a fitness tracker, where an adversary manipulated plain HTTP traffic and TLS proxy between an original gateway and the update server. This threat can be mitigated by providing a secure distribution environment (e.g., a security-hardened MUA) and implementing a signature verification process on the FC.

Traffic interception. In such a scenario, an attacker intercepts all traffic to and from a device with the ability to monitor or modify any data sent to or received from the device (Mtetwa et al., 2019). This capability allows an attacker to alter or drop a valid firmware bundle or its associated metadata during the distribution phase.

This threat can be mitigated by enforcing a secure transmission protocol and/or encrypting the exchanged data (Nguyen et al., 2015).

Image replacement on the device. In this scenario, the attacker replaces a newly downloaded firmware after the device finishes verifying its metadata (e.g., it executes integrity checks on the manifest file), fooling the device into executing the attacker's image. This attack likely requires physical access to the device; however, it can be carried with another threat that allows remote execution. Common mitigation techniques consist of adopting a verification mechanism of the firmware bundle (e.g., signature/digest verification) on the FC and storing the bundle in immutable/protected memory (Internet Engineering Task Force (IETF), 2018).

Modification of metadata between authentication and use. If an attacker can modify the metadata information after it is authenticated (Time Of Check) but before it is used (Time Of Use) (Mtetwa et al., 2019). The attacker can replace any content whatsoever. For instance, she can replace the URI of the firmware bundle in the update manifest file after it is validated by the MUA, causing the FC to download and install a repackaged firmware. This threat can be mitigated by enforcing the verification of the metadata on the FC and not on intermediate agents (e.g., the MUA).

Exposure of signing keys. If an attacker obtains a key or even indirect access to a key, then she can perform the same actions as the legitimate user. In the worst case, if the key retrieved by the attacker is considered trusted by the firmware update chain, the attacker can perform firmware updates as though they were the legitimate owner of the key. For example, if the attacker can obtain the Firmware Manufacturer's signing key, she can generate malicious firmware updates and deliver them through the distribution framework. This threat can be mitigated by storing the signing keys in a protected/separated storage, implementing a key rotation mechanism, or using air-gapped devices to execute the signing process.

Unauthenticated images. In the case the IoT device does not verify the image, an attacker can install a custom firmware on a device by, for example, manipulating either the payload or the metadata gaining complete control of the device. This attack can be prevented by introducing digitally signed metadata that can be verified by the FC (Internet Engineering Task Force (IETF), 2018).

Table 1
List of threats affecting the integrity of firmware during the Firmware Production and Delivery Process.

ID	Name	Phase	Involved entities
IMG.MODIFICATION	Modification of firmware prior to signing	Generation	Producers, FM
META.OVERRIDE	Overriding critical metadata elements	Generation	Producers, FM
DIS.AGCOMPR	Compromission of the intermediate agents	Distribution	Access Point, MUA, IoT Client
DIS.MITM	Traffic interception	Distribution	Network, OTA links
IMG.REPLACE	Image replacement on the device	Distribution, Loading	MUA, IoT device
META.TOCTOU	Modification of metadata between authentication and use	Distribution, Loading	MUA, IoT device
KEY.EXPOSURE	Exposure of signing keys	Generation, Loading	FM, IoT device
IMG.NON_AUTH	Unauthenticated images	Loading	IoT device

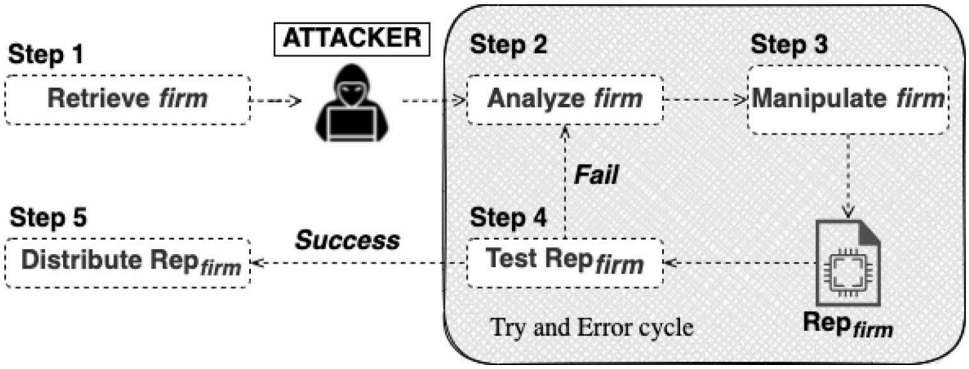


Fig. 2. Steps of a *firmware repackaging* attack.

4. Firmware repackaging: Background, attacker model & countermeasures

The goal of a firmware repackaging attack (Panchal et al., 2018) is to tamper with legitimate firmware and redistribute the modified - i.e., *repackaged* - version to IoT devices to perform further attacks. Repackaging attacks on IoT firmware are motivated by at least one of the following reasons:

- **Unauthorized access/usage.** The attacker modifies the firmware to bypass the predefined software access control privileges, e.g., to gain access to privileged functionalities or classified data.
- **Unlicensed clones.** The attacker's goal is to reuse the crucial firmware processes in some other programs. To this aim, the attacker can extract and partially reuse part of the firmware image to craft a clone of the original version (Al-Wosabi et al., 2015).
- **Malware injection.** By injecting malicious code, the attacker aims to breach the firmware integrity, thereby illegally altering the firmware behavior. In such a case, compromised firmware can disrupt the trustworthiness of an IoT device (ZDNet, 2016).
- **Disrupting the system availability.** The attacker aims to reduce the system availability, injecting code in the firmware image able to cause system halting (like DoS Attack) or significant delays in the regular operation of an IoT device.

To carry out a repackaging attack, a malicious actor jointly exploits the security threats discussed in Section 3. Fig. 2 illustrates the steps involved in the firmware repackaging attack.

The first step is retrieving the original firmware image ($firm$ in Fig. 2). Attackers can get hold of the firmware by: *(i)* dumping it using the physical/remote access interfaces of an IoT device or an intermediate agent (Vasile et al., 2018) (DIS.AGCOMPR), *(ii)* sniffing the package during an over the air (OTA) update (Gupta, 2019) (DIS.MITM), or *(iii)* downloading it from the vendor's website, support, and community forums, or public repositories.

Then, the attacker analyzes the target firmware (Step 2) using static and dynamic analysis techniques to inspect its behavior and extract sensitive information such as encryption keys, hard-coded credentials, or sensitive URLs. Notable tools to perform such analysis include Binwalk (ReFirm Labs, 2014), Firmware Analysis Comparison Toolkit (FACT) (Fraunhofer FKIE, 2015), Firmware Modification Kit (brianpow, 2015), and Firmwalker (Craig Smith, 2015). This is a crucial step that allows an attacker to thoroughly understand the firmware bundle, exposing the firmware producer to security and privacy issues. The extracted knowledge will be used in Step 3 to manipulate the original image to inject custom code (e.g., malware), modify the existing binary image to alter the legitimate behavior (IMG.MODIFICATION) or override critical metadata elements (META.OVERRIDE). In this step, the attacker can exploit binary rewriting techniques (Wenzl et al., 2019) to produce a crafted version of the firmware, i.e., REP_{firm}.

Finally, the attacker tests whether REP_{firm} works properly (Step 4). If this is the case, in Step 5, the attacker redistributes REP_{firm} in the delivery pipeline (e.g., DIS.AGCOMPR) or installs it directly on a target device (e.g., IMG.NON_AUTH, or META.TOCTOU). Otherwise, the attacker further analyzes and modifies the original $firm$ (back to Step 2). Steps 2 to 4 are also known as the *try and error cycle* that the attacker must keep executing until she gets a working repackaged firmware.

4.1. Anti-repackaging techniques

Anti-repackaging aims to protect software from being successfully repackaged. In the context of the IoT update process, anti-repackaging protects the integrity of the entire firmware to ensure that the IoT device will download, install, and execute the expected update. From the attacker's side, the activities to mount a repackaging attack will now include two additional steps (i.e., Steps 2 and 3 of Fig. 3) related to detecting and disabling repackaging protection techniques.

To this aim, an ideal anti-repackaging solution never lets the attacker obtain working repackaged firmware (i.e., moving out from the *try and error cycle*). A reliable anti-repackaging solution makes the repackaging non-cost-effective, i.e., it requires so much time to be disabled that the attacker gives up on repackaging the target.

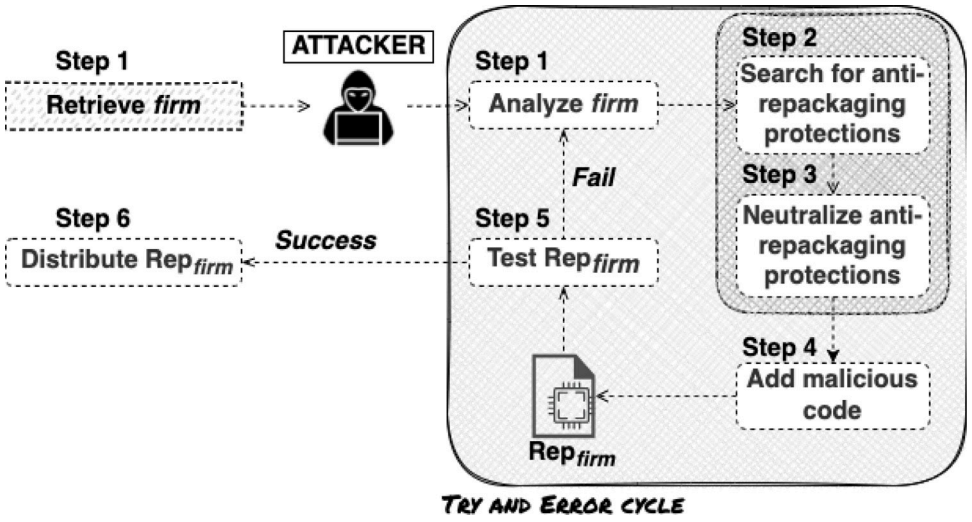


Fig. 3. Steps of a *firmware repackaging* attack for anti-repackaging protection.

Depending on the type of techniques enforced on the pipeline, we can distinguish between two categories:

External anti-repackaging. This category includes all the techniques relying on an external agent to execute anti-tampering checks. Examples are update agents (Langiu et al., 2019), trusted servers (Asokan et al., 2018), or even blockchains (Lee and Lee, 2017). For example, the authors of Perito and Tsudik (2010) propose a Secure Code Update By Attestation in sensor networks (SCUBA), which can be used to repair a compromised sensor through firmware updates. SCUBA utilizes an authentication mechanism and software-based attestation to identify memory regions infected by malware and transmits the repair update to replace these regions. However, the attestation technique based on self-checksumming code heavily relies on consistent timing characteristics of the measurement process and an optimal checksum function. Due to these assumptions, SCUBA is not a suitable approach for IoT settings.

To achieve the goal of protection, these techniques face the challenge of creating a communication channel between the firmware and the external authority to perform the check. However, such a prerequisite may not be feasible in real-world scenarios of IoT ecosystems with low connectivity or limited computational capabilities. In addition, this channel has to be protected from repackaging, exposing it as a single point of failure.

Internal anti-repackaging. This category of techniques aims to protect software from being successfully repackaged by adding some protection code – called *detection nodes* – in the source code before building the firmware image and delivering it to the distribution phase.

The idea of detection nodes has been put forward in Luo et al. (2016) and it refers to a self-protecting mechanism made by a piece of code inserted into the original software, which carries out integrity checks – called *anti-tampering* controls (e.g., signature check, package name check) – when executed at runtime. More specifically, anti-tampering checks compare the signature of a specific part of the firmware with a value pre-computed during the building of the original bundle; if such values differ, then a repackaging is detected, and the detection node usually leads the firmware to fail, thereby frustrating the repackaging effort.

To protect detection nodes, anti-repackaging solutions hide them into the so-called *logic bombs* (Zeng et al., 2018), which has been originally conceived in the malware world to hide malicious payloads (Sharif et al., 2008). A logic bomb is a piece of code that is executed when specific conditions are met. While logic bombs are widely used by malware to introduce and trigger malicious conditions inside apparently unarmed code (Fratantonio et al., 2016; Brumley et al., 2008), this technique can also include tampering detection code (namely, AT checks) inside the activated bomb. In a nutshell, anti-tampering checks are self-protecting functions that aim to detect modifications in a piece of software. To this goal, AT checks may verify at runtime some relevant information of the code or the whole executable file against precomputed values. Several methods exist to detect tampering in executable files, as highlighted in Ahmadvand et al. (2019) and Eldefrawy et al. (2012).

To hide the behavior of a logic bomb, researchers introduced the concept of the Cryptographically Obfuscated Logic Bomb (hereafter, CLB). At build time, the content of a logic bomb is replaced by an encrypted version, which is only decrypted at runtime. In a typical scenario, to perform encryption, the developer should include (and thus reveal) the decryption key(s) in the executable. To avoid revealing the key, the authors in Zeng et al. (2018) proposed a novel form of CLB that exploits the executable's logic to hide the key value. In particular, the use of a CLB (Listing 1) consists in embedding a logic bomb in a *qualified condition* (QC) and replacing it with an encrypted form. A QC is a branch (if statement) containing an equality check where one of the operands is a constant value (e.g., `X == const`). Thus, since inside a QC, the variable (i.e., `X`) is *always* equal to the constant value (i.e., `const`), `X` can be used as a decryption key for the content of the branch. It is worth noting that the CLB is self-contained and does not require an external or internal thrust anchor: a developer should not include and expose the decryption key(s), which is automatically computed by the program at runtime.

To create this kind of CLB, the original condition is transformed into a new one where the pre-computed hash value of the constant (i.e., `Hconst`) is compared with the result of the hash function applied to variable `X` plus some `salt`. Besides, the original content of the qualifying condition is encrypted using the `const` value as the encryption key (i.e., `encrypted_content`). If the triggering condition is met, then the `X` value is used to decrypt the `encrypted_content` and, thus, launch the bomb.

```
1 if(H(X,salt) == Hconst){
2     body = decrypt(encrypted_content, X);
3     execute(body);
4 }
```

Listing 1: Example of Cryptographically Obfuscated Logic Bomb (CLB).

Logic bombs rely on the information asymmetry between the developer and the attacker, i.e., since the attacker has partial knowledge of the software behaviors, it is unlikely that she can correctly guess the

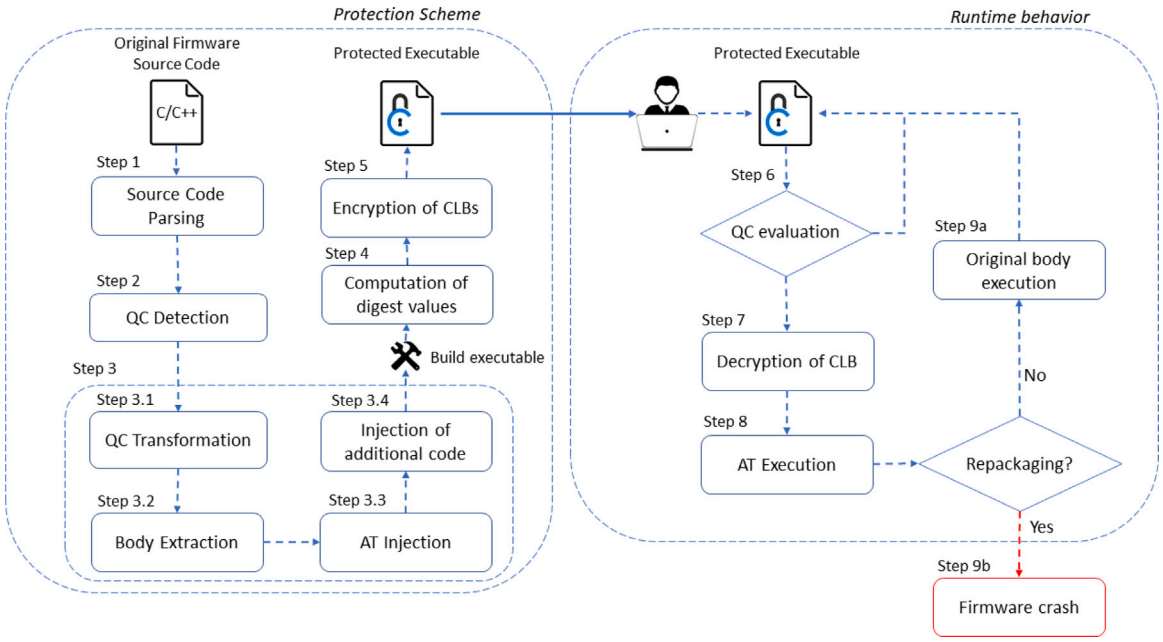


Fig. 4. High-level overview of PARIOT protection process and runtime behavior of a protected firmware.

key value used to encrypt the code. The protection of the CLB is granted by the one-way property of cryptographic hash functions, which makes it hard for the attacker to retrieve the original `const` value (i.e., the decryption key) from its hash.

5. Related works

Operating systems designed explicitly for constrained IoT devices (e.g., TinyOS Panchal et al., 2018 and Contiki Al-Wosabi et al., 2015) often embed or can be extended with over-the-air reprogramming capabilities. Still, many existing solutions focus only on a portion of the update process or do not properly verify the downloaded firmware and hence cannot ensure its integrity. This indeed results in OSes without an update system (e.g., NuttX Foundation, 2018) or with an incomplete one. For instance, Sparrow (SICS, 2018) (used by Contiki) and Deluge (Hui and Culler, 2004) (used by TinyOS) only verify the CRC to ensure the integrity of the firmware during transmissions, leading to the possibility of abusing the lack of controls by an attacker (e.g., through IMG.MODIFICATION, META.TOCTOU, and IMG.NON_AUTH threats).

To this aim, the research community has been working on the definition of secure IoT update processes (Arakadakis et al., 2020; El Jaouhari and Bouvet, 2022). For instance, Hyun et al. (2008), Lanigan et al. (2006), and Dutta et al. (2006) proposed secure extensions for Deluge that provide integrity assurance for the firmware image and resilience against DoS attacks that specifically target firmware dissemination protocols.

Other solutions, like UpKit (Langiu et al., 2019) or ASSURED (Asokan et al., 2018), put forward scalable and lightweight approaches able to perform software updates with end-to-end protection across different OSes and hardware platforms. Also, the authors in Zandberg et al. (2019) propose a secure firmware update mechanism for constrained IoT devices based on open standards such as CoAP, LwM2M, and SUIT.

Finally, several works (Lee and Lee, 2017; Hu et al., 2019; Yohan and Lo, 2018) exploited the blockchain technology to verify the authenticity and integrity of a firmware version and to distribute a specific version of firmware binary to the connected nodes in the blockchain network. Notable examples include Firmware-Over-The-Blockchain (FOTB) (Yohan and Lo, 2020) and CHAINIAC (Nikitin et al., 2017) that exploit Bitcoin and Ethereum blockchain, respectively.

In our work, we reviewed the main state-of-the-art firmware update solutions to identify the adopted integrity protection mechanisms, the mandatory requirements to ensure their proper functioning and the involved entities of the Firmware Production and Delivery Process. Table 2 reports the results of the analysis.

Unfortunately, most of the existing proposals build the integrity and authenticity of updates on one or more signing keys, which are prone to loss (ThreatPost, 2015), theft (FreeBSD, 2012), or misuse (Register, 2022) (i.e., KEY.EXPOSURE threat). Proper protection for signing keys to defend against such single points of failure is a top priority but requires secure storage technologies such as hardware security modules (Asokan et al., 2018). It is worth emphasizing that revoking and renewing signing keys (e.g., in reaction to a compromise) and informing all their clients about these changes is usually cumbersome.

Also, some existing techniques face the challenge of creating a communication channel between the IoT device and the external authority (e.g., the blockchain or a tamper-proof server) to perform the check (e.g., DIS.MITM, or IMG.REPLACE threats). However, such a prerequisite may not be feasible in real-world scenarios of IoT ecosystems with low connectivity or limited computational capabilities.

To overcome such limitations, in this paper, we will present PARIOT. This first solution ensures the integrity of the firmware update by relying on a self-protection mechanism that does not require signing keys, internet connection, secure storage technologies, or external trusted parties. In addition, our methodology is agnostic w.r.t. the firmware delivery model and, thus, can be adopted on top of existing solutions providing an extra layer of security with negligible overhead.

6. PARIOT

The analysis of the threats to the firmware integrity, the definition of the attacker model, and the evaluation of the main solutions to ensure the integrity of the IoT firmware during the update process allowed us to determine that:

1. state-of-the-art solutions are vulnerable to IMG.MODIFICATION, i.e., an attacker or a malicious FM can repack the content of the IoT firmware before the signing process in the generation phase;

Table 2
Analysis of the integrity protection mechanisms adopted by state-of-the-art firmware update solutions.

Work	Year	Integrity protection mechanism	Requirements
Lanigan et al. (2006)	2006	ECDSA signature verification of each transmitted block	– Pub Keys (IoT Devices) – Secure Storage Technology for Private Key (FM)
Dutta et al. (2006)	2006	RSA signature verification of each transmitted block	– Pub Keys (IoT Devices) – Secure Storage Technology for Private Key (FM)
Hyun et al. (2008)	2008	Merkle hash tree signature verification of each transmitted block	– Pub Keys (IoT Devices) – Secure Storage Technology for Private Key (FM)
Samuel et al. (2010)	2010	Signature verification with multiple roles	– Pub Keys (IoT Devices) – Secure Storage Technology for Private Key (FM)
Aschenbruck et al. (2012)	2012	Elliptic Curve Cryptography (ECC) signature verification	– Pub Keys (IoT Devices) – Secure Storage Technology for Private Key (FM)
Salas (2013)	2013	Biometric-aided ECC cryptosystem	– Dedicated Hardware (IoT devices) – Secure Storage Technology (IoT devices)
Doroodgar et al. (2014)	2014	Merkle hash tree signature verification of each transmitted block	– Pub Keys (IoT Devices) – Secure Storage Technology for Private Key (FM)
Chandra et al. (2016)	2016	No integrity verification	–
Karthik et al. (2016)	2016	Signature verification with multiple roles	– Pub Keys (IoT Devices) – Secure Storage Technology for Private Key (FM)
Doddapaneni et al. (2017)	2017	Signature Verification of FOTA Objects	– Pub Keys (IoT Devices) – Secure Storage Technology for Private Key (FM)
Nikitin et al. (2017)	2017	Collective signature verification	– Pub Keys (IoT Devices) – Secure Storage Technology for Private Key (FM) – Secure connection with the Blockchain (IoT device)
Lee and Lee (2017)	2017	Signature verification of the metadata file stored in the Blockchain	– Storage space for the Blockchain (IoT devices) – Secure connection with the Blockchain (IoT device) – Pub Keys (nodes) – Secure Storage Technology for Private Key (nodes)
Teng et al. (2017)	2017	Signature verification stored in Trusted Platform Modules	– Pub Keys (IoT Devices) – Secure Storage Technology for Private Key (FM)
Doddapaneni et al. (2017)	2017	Signature and Encryption of Secure Object Format	– Pub Keys (IoT Devices) – Secure Storage Technology for Private Key (FM)
Prada-Delgado et al. (2017)	2017	Encrpytion using shared keys of the update payload	– Physically Unclonable Functions (IoT Devices)
Asokan et al. (2018)	2018	RSA signature verification of the firmware metadata	– Hardware Security Module (all)
Yohan and Lo (2018)	2018	FMart Contract signature verification of the firmware	– Secure connection with the Blockchain (IoT device) – Pub Keys (nodes) – Secure Storage Technology for Private Key (nodes)
Kumar et al. (2018)	2018	Signature and Encryption using AES and RSA keys	– Pub Keys (IoT Devices) – Secure Storage Technology for Private Key (FM) – Secure Storage Technology for Shared Key (all)
Zandberg et al. (2019)	2019	Elliptic Curve Cryptography (ECC) signature verification of manifest metadata	– Pub Keys (IoT Devices) – Secure Storage Technology for Private Key (FM)
Hu et al. (2019)	2019	FMart Contract signature verification of the firmware	– Hardware Security Module (all nodes) – Secure connection with the Blockchain (IoT device)
Gupta and van Oorschot (2019)	2019	Elliptic Curve Cryptography (ECC) signature verification of software update	– Pub Keys (IoT Devices) – Secure Storage Technology for Private Key (FM)
Mbakoyiannis et al. (2019)	2019	Multi-trust signature verification of manifest metadata	– Pub Keys (IoT Devices) – Secure Storage Technology for Private Key (FM)
Langiu et al. (2019)	2019	Double signature on the update image	– Pub Keys (IoT Devices) – Secure Storage Technology for Private Key (FM)
Dhobi et al. (2019)	2019	Hardware-based signature verification	– Pub Keys (IoT Devices) – Secure Storage Technology (IoT devices) – Dedicated Hardware (IoT devices)
Kerliu et al. (2019)	2019	Hardware-based signature verification	– Secure Storage Technology (IoT devices) – Dedicated Hardware (IoT devices)
Pillai et al. (2019)	2019	Verification of Hash Chain	– Storage space for the Blockchain (IoT devices) – Secure connection with the Blockchain (IoT device) – Pub Keys (nodes) – Secure Storage Technology for Private Key (nodes)
Dhakal et al. (2019)	2019	Integrity verification executed in the Blockchain Server	– Tamper-proof Blockchain server – Secure connection with the Blockchain Server (IoT device)

(*continued on next page*)

Table 2 (*continued*).

Witanto et al. (2019)	2019	Peer-to-peer update with Smart Contract signature verification of the firmware	– Secure connection with the Blockchain (IoT device) – Pub Keys (nodes) – Secure Storage Technology for Private Key (nodes)
Yohan and Lo (2020)	2020	Peer-to-peer verification process through consensus	– Pub Keys (IoT Devices) – Secure Storage Technology for Private Key (FM)
Anastasiou et al. (2020)	2020	Smart Contract signature verification of the firmware	– Hardware Security Module (all nodes) – Secure connection with the Blockchain (IoT device)
Sahlmann et al. (2020)	2020	Digest signature verification	– Pub Keys (IoT Devices) – Secure Storage Technology for Private Key (FM)
Falas et al. (2021)	2021	Hardware-based signature verification	– Pub Keys (IoT Devices) – Secure Storage Technology for Private Key (FM)
Tsaur et al. (2022)	2022	Smart Contract signature verification of the firmware	– Storage space for the Blockchain (IoT devices) – Secure connection with the Blockchain (IoT device) – Pub Keys (nodes) – Secure Storage Technology for Private Key (nodes)
Ghosal et al. (2022)	2022	Digest signature verification	– Pub Keys (IoT Devices) – Secure Storage Technology for Private Key (FM)
de Sousa et al. (2022)	2022	Digest signature verification	– Pub Keys (IoT Devices) – Secure Storage Technology for Private Key (FM)

2. All solutions based on the signature of the firmware bundle or the associated metadata are vulnerable to the KEY.EXPOSURE and META.TOCTOU threats thereby vanishing the adopted enforcement mechanisms; the only way to cope with such threats is to adopt secure storage technologies or hardware security modules that may not be compatible with low-end IoT devices;

In this section, we introduce the basics of PARIOT (Pervasive Anti-Repackaging for IoT), the first solution of internal anti-repackaging for IoT firmware that can extend existing software update solutions to ensure resilience against repackaging attacks in the whole production and delivery process. PARIOT does not require signing keys, secure storage technologies, or hardware security modules to ensure full compatibility with low-end IoT devices.

PARIOT protects an IoT firmware by injecting self-protecting code directly inside the firmware code. The methodology exploits the use of Cryptographically obfuscated Logic Bombs (Zeng et al., 2018) (CLB) to hide anti-tampering (AT) checks in the firmware executable. Such CLBs will be triggered (i.e., *explode*) in case of any tampering is detected. Moreover, to defuse an AT check inside a CLB, the attacker would need to execute the CLB, retrieve the value of the decryption key to decrypt its content, and then bypass or remove the AT checks injected in the body of the qualified condition.

This section details the PARIOT protection scheme and its runtime behavior.

6.1. Protection scheme and runtime behavior

The PARIOT protection scheme is based on the dissemination in the IoT firmware of CLBs that hide a set of AT checks. To minimize the complexity, each CLB embeds a single AT check that performs a signature verification of a portion of the IoT firmware executable. Still, it is worth noticing that PARIOT supports the definition of other AT checks as well as different CLB schemes.

Fig. 4 shows a high-level overview of the proposed technique's protection process and the runtime behavior. PARIOT starts the protection process from the source code of the IoT firmware. In detail, it parses the source code (Step 1) to identify the set of suitably qualified conditions (Step 2). In Step 3, these conditions are transformed according to the CLB schema. During this process, the equality check of the QC is transformed into the cryptographically obfuscated form (Step 3.1). Then, PARIOT extracts the body of the qualified condition (Step 3.2), injects the AT code (Step 3.3), and adds the logic to decrypt and execute the new body once in the encrypted form (Step 3.4). The choice of injecting the CLBs directly into the source code allows for minimizing invasive procedures (e.g., binary rewriting) that may break the compatibility with existing firmware update frameworks or disrupt the firmware image.

Depending on the implemented AT controls, the scheme could compute some digest values (e.g., the hash code for some executable code), storing their values inside the corresponding bombs. To do so, PARIOT triggers the firmware compilation process to complete the protection. This process depends on the firmware being protected: PARIOT simply invokes its original build system. In Steps 4–5, the computed values are the expected results of each AT, which executes the integrity check by comparing a runtime computed value with the digest stored in the bomb.

Finally, PARIOT encrypts each CLB with their constant values (`const`) to obtain the protected firmware.

At runtime, the code of the CLBs is executed iff the value of the variable in the QC (plus a salt) is equal to the constant value (i.e., if its hash matches `Hconst` - Step 6). If this is the case, the body of the CLB is decrypted using the `const` value as the decryption key (Step 7), and the corresponding code is executed. This behavior triggers the AT check (Step 8) that computes the digest of a portion of the firmware executable and compares it with the stored one. If these values match, the execution can proceed normally (Step 9a). Otherwise, the AT reports a tampering attempt and executes an action, like sending an alert to the Firmware Manufacturer or triggering a Security Exception and aborting the execution (i.e., the case of Step 9b).

7. PARIOTIC

To demonstrate the applicability and the feasibility of PARIOT, we developed PARIOTIC (i.e., PARIOT for Integrated C-based firmware) to support the protection of IoT firmware designed in C/C++ programming language. The tool is publicly available on GitHub at Computer Security Laboratory (2022).

PARIOTIC consists of two main modules:

- CLB Injector. This module works directly on the firmware source code and is responsible for parsing of the source code, detect the QCs, and build of CLBs (Steps 1–3 of the protection process).
- CLB Protector. This module processes the compiled IoT firmware and is responsible for computing the signature-verification digests of the AT checks and encrypting the CLBs (Steps 4–5 of Fig. 4).

7.1. CLB Injector

CLB Injector is built using the Python language and leverages the Clang library[2] to pre-process the C/C++ source code.

[2] https://github.com/llvm-mirror/clang/tree/master/bindings/python.

During this phase, CLB Injector scans the source code to obtain the list of qualified conditions that can host a logic bomb. In the current implementation, CLB Injector supports if-then-else statements with an equality condition of the form of `X == const`. If we consider the source code of Listing 2 as an example, CLB Injector would detect one QC at row 7.

```
1  int funA(int val);
2
3  void funB(int val) {
4      int a = 0;
5      [...]
6      /* QC */
7      if (a == CONST) {
8          /* or_code */
9          int res = funA(val);
10         printf("The result is %d", res);
11     }
12     [...]
13 }
```

Listing 2: Example of a C source code.

After the QC detection phase, CLB Injector converts each QC in the corresponding CLB. To do so, the tool computes the hash of the constant value of the QC, generates a 4-bytes random salt, and modifies the `if` condition to match the form `if(H(X,salt) == Hconst)`. Then, it creates a new function (*ext_fun*) that encapsulates the QC original body and accepts – as input parameters – all the variables used inside the body. Moreover, during this phase, the module adds an AT control in *ext_fun* (Step 3.3 of Fig. 4). In the current implementation, the AT control evaluates the hash signature of a portion of the compiled firmware and raises a security exception in case of a signature mismatch.

Since CLB Injector works directly on the source code, it injects three placeholder values into the source code that will be updated by CLB Protector before the encryption phase. In detail, the module adds three variables, i.e., `offset`, and `count` to identify the part of the executable to evaluate in the AT control, and `control_value` for the expected result of the verification.

Finally, CLB Injector injects the functions to decrypt and execute *ext_fun* in the body of the CLB.

Listing 3 reports the processing result of CLB Injector on the example code of Listing 2. Starting from the QC located in *funB*, CLB Injector creates a new function (*ext_funB*) that contains the original body of the QC (rows 23–33). Then, it injects an anti-tampering control (row 29) that verifies a portion of the executable file (identified by `offset` and `count` - rows 25 and 26) against the expected hash value (i.e., the variable `control_value` - row 27). Finally, CLB Injector replaces the original body of the QC with the code to decrypt and execute the original function (rows 40 and 41).

```
1  int funA(int value);
2
3  /* ptr to start of the firmware */
4  uint8_t *ptr;
5
6  void at_check(off_t offset, size_t count, int
       control_value, uint8_t *ptr) {
7      int i;
8      uint8_t *buf = malloc(sizeof(uint8_t) * count);
9      /* Read the bytes to control */
10     if (i = 0; i < count; i++) {
11         buf[i] = ptr[offset+i];
12     }
13     /* Integrity check */
14     if (hash(buf) != control_value) {
15         puts("*Aborting: Security Exception (
               Repackaging detected)");
16         free(buf);
17         exit(123);
18     }
19     free(buf);
20     return;
21 }
22
23 void ext_funB(int* val) {
24     /* Placeholders */
25     off_t offset = 0x0ff53701 ;
26     size_t count = 0xb17e5010;
27     int control_value = 0x4559ffff;
28     /* AT check */
29     at_check(offset, count, control_value, ptr);
30     /* or_code of funB */
31     int res = funA(val);
32     printf("The result is %d\n", res);
33 }
34
35 void funB(int val) {
36     int a = 0;
37     [...]
38     /* CLB */
39     if (hash(a,salt) == Hconst) {
40         decrypt(&ext_funB, &a);
41         ext_funB(&val);
42     }
43     [...]
44 }
```

Listing 3: Output code produced by CLB Injector.

7.2. CLB Protector

CLB Protector is a Java command-line tool that processes the compiled IoT firmware to (i) update the control values of the AT checks and (ii) encrypt the content of the CLBs. The module receives from CLB Injector the list of CLBs, the functions that need encryption (i.e., the list of *ext_fun* methods), and their corresponding encryption keys (i.e., the *const* values).

For each CLB, the tool exploits `nm`[3] to locate in the firmware executable the corresponding *ext_fun* and the position of the embedded control values (i.e., `offset`, `count`, and `control_value`). Also, the module identifies the portion of code that the AT checks will evaluate. The current version of CLB Protector selects all the compiled code (the content of the `.text` elf section). From the selected code, the module computes: (i) the starting position (`offset`), (ii) the number of bytes (`count`), and (iii) the hash of the selection (`control_value`). Then, CLB Protector replaces the placeholder values with the obtained results (Step 4 of Fig. 4). Finally, CLB Protector encrypts the bytes of the *ext_fun* using the *const* value as the encryption key (step 5).

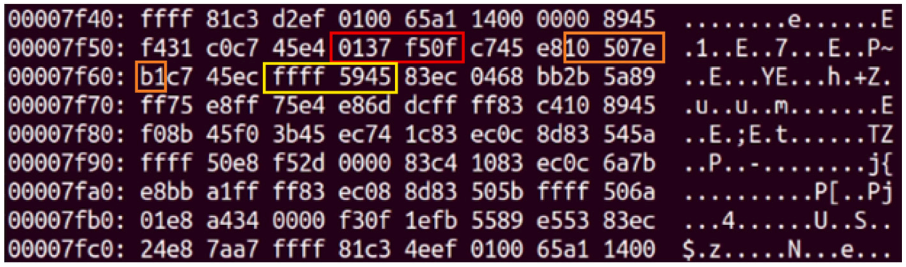
Fig. 5 shows the protection applied by CLB Protector on the part of the executable file containing *ext_funB* of Listing 3. In detail, the tool locates the control values (Fig. 5(a)), computes and updates their values (Fig. 5(b)) and, then, encrypts the entire function (Fig. 5(c)).

8. Experimental evaluation

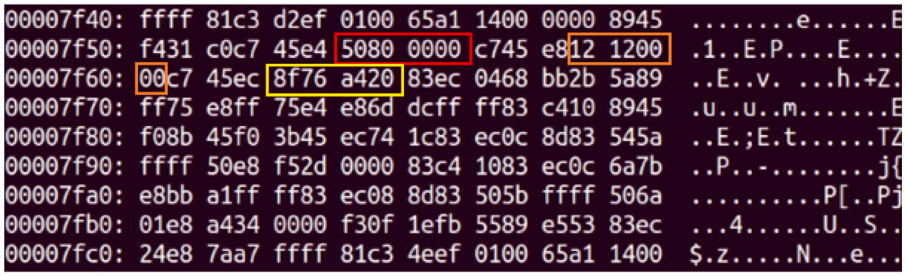
We empirically assessed the applicability of PARIOT by applying the PARIOTIC protection on 50 real-world samples for resource-constrained IoT devices in a RIOT-based ecosystem with the SUIT update framework.

RIOT is an open-source OS designed for resource-constrained IoT devices that have gained the scientific community's attention in the last few years (Baccelli et al., 2018). RIOT allows for standard C and C++ application programming, provides multi-threading and real-time capabilities, and only requires a minimum of 1.5 KB of RAM. RIOT

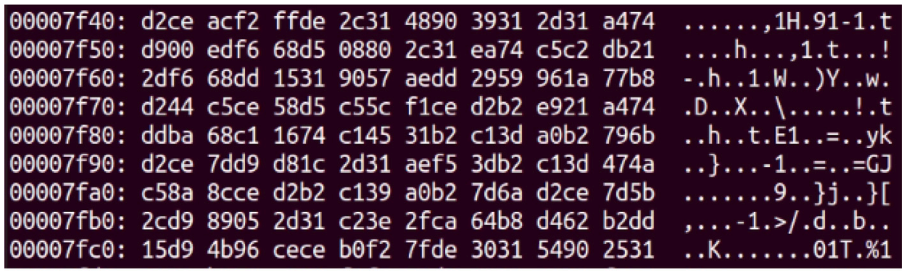
[3] https://linux.die.net/man/1/nm.



(a) Identification of the control values of ext_funB.



(b) Computation and updating of the control values of ext_funB.



(c) Encryption of ext_funB.

Fig. 5. CLB Protector protection process on the part of the executable file containing *ext_funB* of Listing 3.

consists of a microkernel architecture with core functionalities and pluggable modules that support multiple network stacks, libraries, and utility (RIOT OS, 2022b).

In detail, the experimental campaign applied the tool on each firmware to evaluate the distribution of the protection controls and the introduced size overhead. Each firmware is built using RIOT OS version 2021.05 and a different RIOT app (as detailed in Section 8.1) for two different boards, i.e., *native* (RIOT OS, 2022c) and *iotlab-m3* (RIOT OS, 2022a). Hereafter, we refer to each firmware with the name of the contained application.

The building and protection phase was performed on a virtual machine running Ubuntu 20.04 with four processors and 16 GB RAM.

Then, we evaluated the reliability of the protection scheme at runtime. First, we executed the protected firmware images to check the proper functioning. Then, we tested the solution against actual repackaging attacks by attempting to repackage each protected firmware and testing the tampered file. The runtime evaluations were executed on a real *iotlab-m3 board* hosted by the FIT IoT-LAB testbed (Adjih et al., 2015). The iotlab-m3 board has an STM32 MCU, 32-bit Cortex M3 CPU, 64 KB of RAM, and 256 KB of ROM.

8.1. Dataset

The preparation of the dataset consisted of the following steps. First, we scraped GitHub (GitHub, 2022) looking for recent repositories (i.e., since January 2019) that contain the words "RIOT OS" and "IoT" and at least one of the following keywords: {"app", "application", "firmware", "code"} (e.g., query `RIOT OS IoT firmware site:github.com after:2019-1-1`). The analysis resulted in the identification of 150 unique public GitHub repositories matching our criteria. From these, we manually examined the collected repositories to retain only the ones that contain IoT apps compatible with recent RIOT OS versions (i.e., at least $\geq$ 2021.05) and support the IoT boards included in the experimental evaluation (i.e., *native* and *iotlab-m3*). After the review process, we obtained a set of 50 different apps distributed across 16 GitHub repositories (~10% – 16/150). Table A.3 in Appendix reports the name of each app, the link to the GitHub project, and the subfolder that contains the source code. Finally, we built each firmware using RIOT OS version 2021.05 and a different RIOT app for the *native* and *iotlab-m3* boards.

It is important to notice that the low availability of open-source and working apps for the RIOT OS directly influenced the magnitude of the dataset. Still, we obtained a representative collection of samples as the apps come from heterogeneous scenarios and use different RIOT modules. Notables examples are the `museum` app, which is part of the *ARte* (*Augmented Reality to educate*) project, and the `election_master` app. The former leverages the MQTT protocol (Stanford-Clark and Nipper, 2022) relying on the `emcute` RIOT module to improve the interaction between visitors and artworks with the COVID-19 restrictions. The latter implements a custom leader election algorithm on RIOT OS nodes connected through the network and includes the modules for the routing protocol (i.e., `gnrc_rpl`, and `auto_init_gnrc_rpl`). We reported in Table A.4 in Appendix a detail of the RIOT modules used in each app of the dataset. Finally, It is worth empathizing that large-scale analysis is out of the scope of this work: our evaluation aims to demonstrate the effectiveness and the enforceability of PARIOTIC regardless of the features (i.e., included RIOT OS modules) of the RIOT apps.

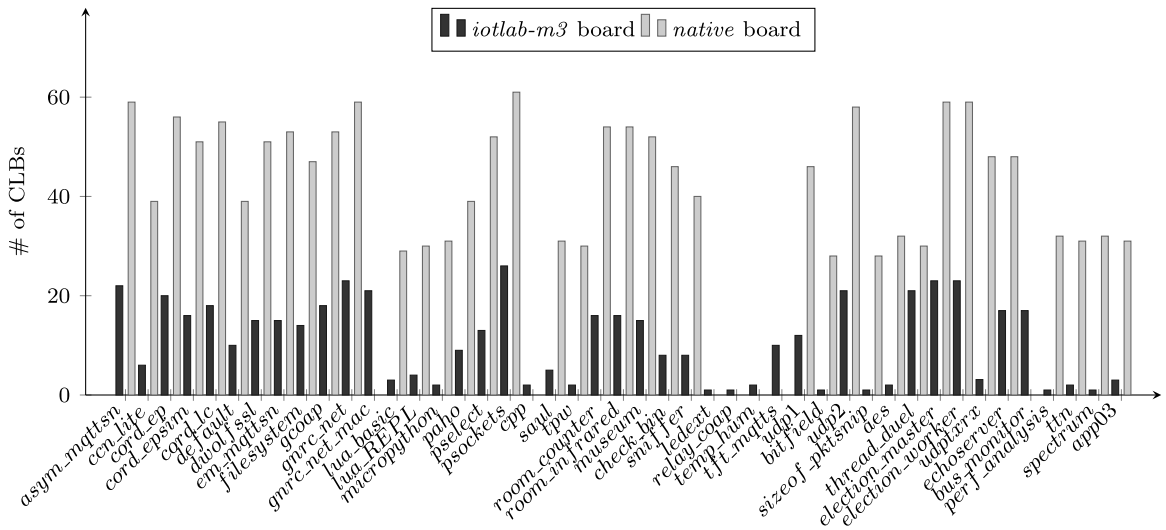


Fig. 6. Number of CLBs injected in the RIOT firmware samples of the dataset.

8.2. Protection evaluation

PARIOTIC was able to apply the protections over the entire dataset in nearly 53 min (i.e., 3208 s) for iotlab-m3 board and 46 min (2818 s) for the native board. The protection of a single firmware took, on average, 64.2 s.

PARIOTIC worked successfully in 90% (45/50) of the cases, i.e., it generated valid protected firmware. The remaining 10% (i.e., 5 firmware images based on `decho`, `dsock`, `gnrc_brouter`, `nanocoap`, and `ndn` apps) failed due to errors in the building phase. We manually investigated such problems to discover that the build process failed due to the presence of at least an *ext_fun* with (i) unsupported instructions (e.g., goto statements to undefined portions of code), or (ii) undefined variables. These problems are mainly attributable to the parsing of the source code (i.e., Step 1 of Fig. 4) that leads to incorrect identification of the body of the QCs by the Clang python extension.

Fig. 6 shows the number of logic bombs distributed in each protected RIOT firmware for the two boards. PARIOTIC injected, on average, 10.9 CLBs (st. dev. 8.1) on firmware images for iotlab-m3 and 43 CLBs (st. dev. 11.9) on the ones compatible with the native board. The significant difference in injected bombs reflects the peculiarities of the two target devices regarding the codebase, compatible APIs, and external modules included by default. In particular, the native board encloses additional code and, consequently, more QCs to apply the protection scheme concerning the Cortex-based counterpart. Finally, it is worth noting that 12 executables for iotlab-m3 boards have few CLBs (i.e., less than 3) since they include a basic app (e.g., similar to *hello world*), thereby exacerbating such a gap.

Fig. 7 shows the percentage size overhead introduced by the protection on the firmware executable. The graph reflects the expected trend following the number of injected CLBs for each firmware: the average size overhead is higher in the native w.r.t. the iotlab-m3 board. In the first case, the average size overhead is 11.3% with a standard deviation of 2.8%; in the latter case, it remains within the 5% range (avg. 3.52%, st. dev. 1.32%). In terms of absolute size, the actual increase consists of a few kilobytes. The native board has an average growth of 134.3 KB and a standard deviation of 46.7 KB. The size of the iotlab-m3 board firmware increased to 82.9 KB on average, with a standard deviation of 34.4 KB. Also, it is worth noting that the size overhead is always less than 15.8%, even when PARIOTIC injects more than 60 CLBs, thereby corresponding to a maximum growth of 197.9 KB for the native board and 160.3 KB for the iotlab-m3.

Then, we empirically tested the protected firmware at runtime to verify that the introduced protections did not harm the normal functioning of the software bundle.

In detail, the runtime evaluation consisted of 5 test runs for each protected firmware; each app has been stimulated for 2 min with a sequence of manual inputs obtained from the `help` command of the app or extracted from the documentation (where available). Including the iotlab-m3 physical board (among the most used in the FIT IoT-LAB testbed) in the experimental campaign allowed us to demonstrate the solution's applicability in real-world environments. The experimental evaluation reported that all the protected firmware samples executed correctly, i.e., they did not crash nor trigger exceptions.

8.3. Protection overhead

This set of experiments aims to evaluate the runtime overhead introduced by the CLBs in a real IoT device. To do so, we leveraged the consumption monitoring tool (RIOT OS, 2023) provided by IoT-LAB. It allows to measure the energy consumption of a node in terms of current (Ampere), voltage (Volt), and power (Watt), through an INA226 hardware component. INA226 has a programmable conversion times (CT), which allows it to be configured to optimize the available timing requirements in a given application. Along with the CT, the averaging mode (AV) allows the INA226 to be more effective in reducing the noise component during the measurement. Thus, the periodic measure (PM) (or sampling period) is given by the formula

$$PM = CT * AV * 2$$

We created a monitor profile with a CT of 8244 μs and AV of 4, resulting in a sampling rate of almost 66 ms.

We executed both the original and the protected version of the 45 firmwares in an iotlab-m3 node with the consumption monitoring enable. It is worth noticing that each firmware is tested under the same input commands sequence.

Fig. 8 shows the overall mean and standard deviation for power, voltage, and current consumptions for both the original and the protected firmware.

The results highlight that the introduced protections do not significantly impact resource usage, even if the CLBs are decrypted during the firmware's executions. In particular, the voltage and current consumptions are comparable between the original and the protected versions, which on average are 3.30 (st. dev. 0.0070) vs. 3.31 (st. dev. 0.0071) for the voltage and 2.75 (st. dev. 0.8) vs. 2.8 (st. dev. 0.82) for the

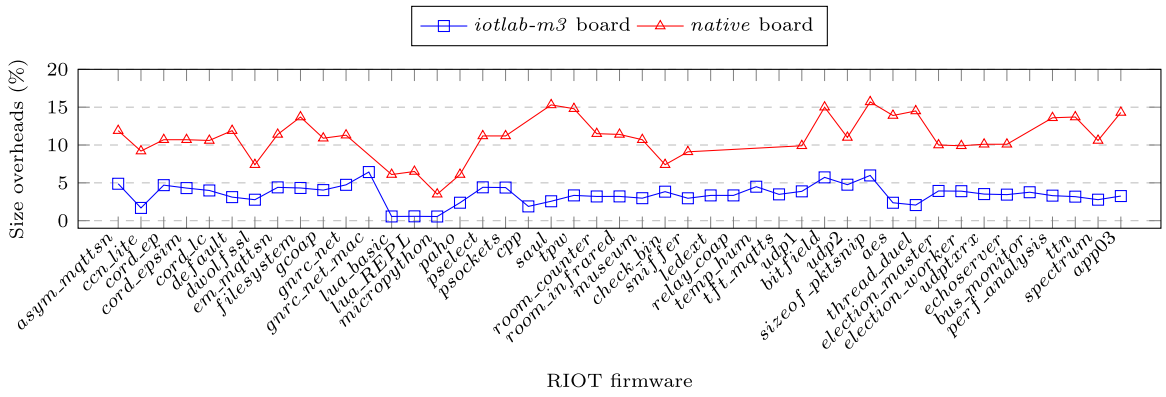


Fig. 7. Distribution of the size overhead over the protected RIOT apps.

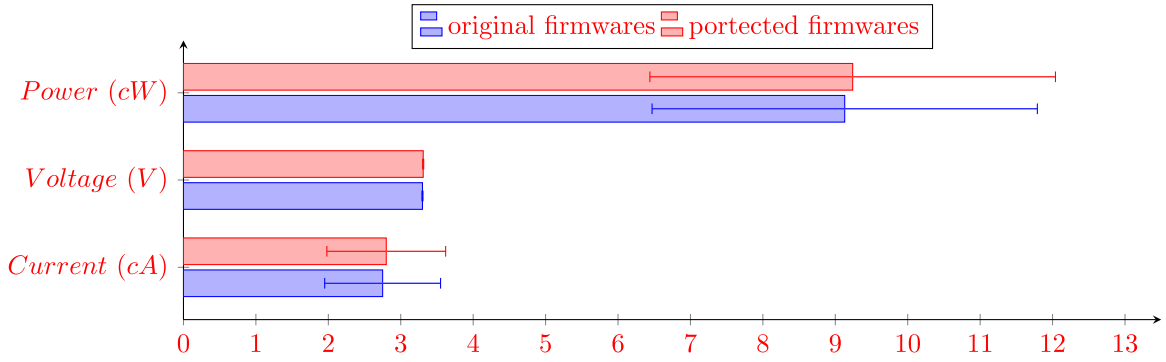


Fig. 8. Average and standard deviation of the current, voltage, and power usage for the original and protected firmware samples.

current user. However, we noticed a clear but still limited difference in the power consumption. The protected firmware consumes on average 1.2 mW more compared to their respective originals.

8.4. Efficacy of the protection scheme

The last set of experiments aims to evaluate the efficacy of the protections introduced by PARIOT. To do so, we automatically exploited a real repackaging attack on each of the firmware of the dataset. In detail, we built a script that modifies a set of random values in the compiled firmware. For instance, in Fig. 9, the repackaging script replaced the original string (i.e., `RIOT native interrupts/signals initialized`) with a custom one (i.e., `RIOT has been repackaged!`).

We repackaged all the protected firmware samples (i.e., 45 executables) and repeated the execution with the same set of inputs of the previous phase on an iotlab-m3 board hosted in the FIT IoT-LAB datacenter in Lille. The experiments showed that 69% of the firmware samples (i.e., 31/45) successfully detected the repackaging attempt. In particular:

- 23 apps detect the repackaging at the startup;
- 8 apps detect the repackaging during the execution of the user inputs, i.e., when a specific input is sent to the firmware;
- 14 apps do not detect the repackaging within the end of the test.

The results suggest a high repackaging detection rate. However, 31% of the tested apps failed to detect firmware modification. In the previous phase, we highlighted how 12 firmware images of the dataset contain only a few CLBs (i.e., less than 5). In fact, most of the failed detections are related to basic firmware samples that include an app with few lines of code. For instance, Listing 4 shows the `main` function of the *twp* RIOT app.

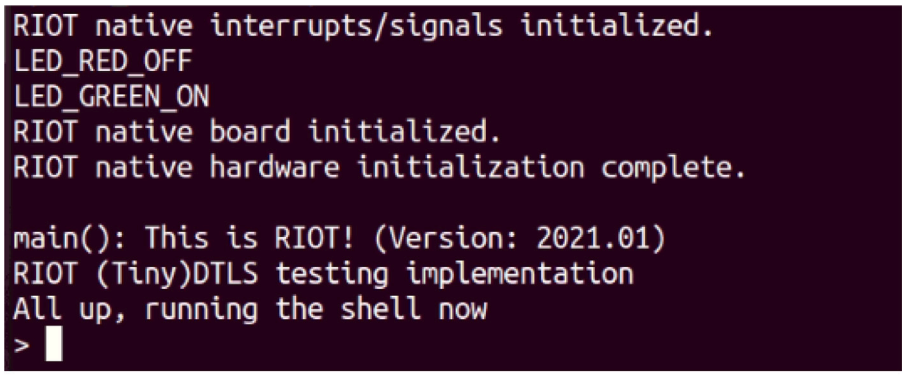
```
1  int main(void) {
2      xtimer_ticks32_t last_wakeup = xtimer_now();
3      while(1) {
4          xtimer_periodic_wakeup(&last_wakeup,
               INTERVAL);
5          printf("slept until %" PRIu32 "\n",
               xtimer_usec_from_ticks(xtimer_now()
               ));
6      }
7      return 0;
8  }
```

Listing 4: Example of elementary app.

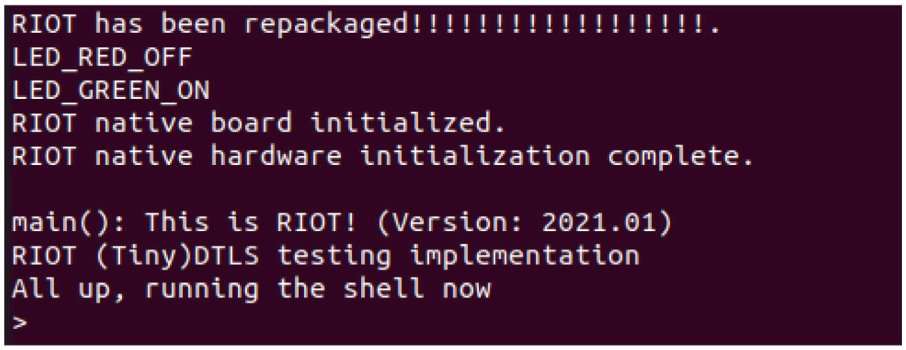
The `main` function and the invoked ones (i.e., `xtimer_now`, `printf`, and `xtimer_usec_from_ticks`) do not contain any QC or complex statements, resulting in a lack of CLBs injected in the app code. In addition, the simplicity of the app's logic limited the range of potential inputs that can trigger the other CLBs included in the OS kernel, resulting in a lack of detection. Nonetheless, for the sake of this paper, our experimental setup has been sufficient to prove that the protection scheme is reliable. Such a consideration is supported by positive detection in the case of firmware samples with real-world apps.

8.5. Prototype limitations

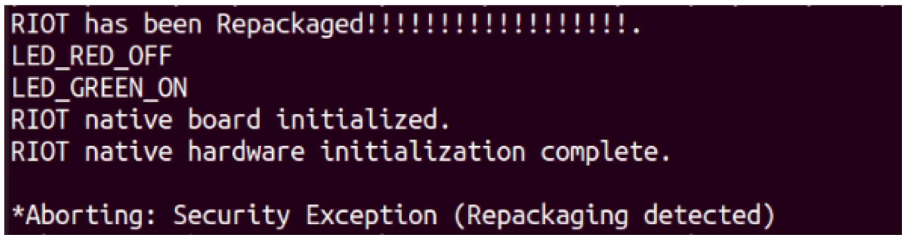
The experimental campaign identified some limitations of PARIOTIC that will be discussed below. In the current implementation, CLB Injector injects the functions to decrypt and execute *ext_fun* as static methods in each file that contains at least a CLB. Such a choice removes any dependency from third-party libraries that may not be natively supported by the firmware (e.g., AES) or included in the firmware bundle at the cost of introducing potentially redundant code. Moreover, this technique allows using decryption and hashing functions independently from the part of the code that contains the CLB, which



(a) Execution of the original firmware.



(b) Execution of the repackaged firmware without protection.



(c) Execution of the repackaged firmware protected with PARIOTIC.

Fig. 9. Steps of the security evaluation of PARIOTIC on a firmware sample.

may not have access to the corresponding decryption library (e.g., a CLB in the startup code). PARIOTIC implements a xor-based cipher algorithm to encrypt the CLBs to guarantee that the encrypted payload would have the same size/offset, thus avoiding potential misalignment or overrides during the binary rewriting process.

The experimental results also highlighted a direct correlation between the complexity of the *const* value used as the encryption key and the hiding capability offered by the CLBs. For instance, we discovered that many constant values in the RIOT OS are 2 bytes, thus limiting the resiliency of PARIOTIC against brute force attacks to guess the encryption key and bypass the CLBs. To overcome this limitation, PARIOTIC could be extended to detect different types of logic bombs, such as QCs containing a string comparison function like `strcmp()` and `strncmp()`.

Also, it is worth noticing that even if PARIOT supports generic firmware images, either packed in a single executable or a package (e.g., OpenWRT Holt and Huang, 2014), we focused our work on firmware bundles assembled as an executable file, such as the RIOT firmware. Thus, PARIOTIC implements AT controls to verify a set of properties of the executable file, albeit they can be extended to check the integrity of other executables or non-code files (e.g., a configuration file).

Finally, it is worth noting that each processor provides a different set of capabilities. For instance, in the native board, we can leverage the memory management APIs of the host processor (e.g., mprotect Linux, 2021) to alter the memory protection bits and modify the content of the executable section (`.text`). In the case of the ARM Cortex CPU and the iotlab-m3 board, these APIs are not available, thereby requiring a change in the implementation of the decryption logic (e.g., execute/-modify some methods on the `.data` section). In other words, the PARIOT methodology can be applied to generic IoT devices, but its implementation (i.e., PARIOTIC) may require slight changes depending on the target board and hardware/software constraints.

9. Conclusion

In this work, we proposed PARIOT, a self-protection mechanism that ensures the resiliency of IoT firmware images against repackaging attacks through the entire production and delivery process. The methodology exploits the use of CLBs to hide anti-tampering checks in the firmware executable that will trigger (i.e., explode) in case of any tampering is detected.

Furthermore, we implemented PARIOT in a tool for protecting C/C++ firmware, called PARIOTIC, that is publicly available on GitHub (Computer Security Laboratory, 2022). PARIOTIC modifies the firmware source code to inject CLBs and AT controls and the compiled binary to build the CLBs by encrypting specific binary portions. It is worth emphasizing that the integrity controls do not rely on external

Table A.3
List of RIOT apps composing the dataset.

App name	GitHub Repo	Folder
asym_mqttsn	RIOT-OS/RIOT	examples/asymcute_mqttsn
ccn_lite	RIOT-OS/RIOT	examples/ccn-lite-relay
cord_ep	RIOT-OS/RIOT	examples/cord_ep
cord_epsim	RIOT-OS/RIOT	examples/cord_epsim
cord_lc	RIOT-OS/RIOT	examples/cord_lc
default	RIOT-OS/RIOT	examples/default
decho	RIOT-OS/RIOT	examples/dtls-echo
dsock	RIOT-OS/RIOT	examples/dtls-sock
dwolfssl	RIOT-OS/RIOT	examples/dtls-wolfssl
em_mqttsn	RIOT-OS/RIOT	examples/emcute_mqttsn
filesystem	RIOT-OS/RIOT	examples/filesystem
gcoap	RIOT-OS/RIOT	examples/gcoap
gnrc_brouter	RIOT-OS/RIOT	examples/gnrc_border_router
gnrc_net	RIOT-OS/RIOT	examples/gnrc_networking
gnrc_net_mac	RIOT-OS/RIOT	examples/gnrc_networking_mac
lua_basic	RIOT-OS/RIOT	examples/lua_basic
lua_REPL	RIOT-OS/RIOT	examples/lua_REPL
micropython	RIOT-OS/RIOT	examples/micropython
nanocoap	RIOT-OS/RIOT	examples/nanocoap_server
ndn	RIOT-OS/RIOT	examples/ndn-ping
paho	RIOT-OS/RIOT	examples/paho-mqtt
pselect	RIOT-OS/RIOT	examples/posix_select
psockets	RIOT-OS/RIOT	examples/posix_sockets
cpp	RIOT-OS/RIOT	examples/riot_and_cpp
saul	RIOT-OS/RIOT	examples/saul
tpw	RIOT-OS/RIOT	examples/timer_periodic_wakeup
room_counter	ARte-team/ARte	src/Boards/room_counter_emcute
room_infrared	ARte-team/ARte	src/Boards/room_infrared_emcute
museum	ARte-team/ARte	src/Boards/museum_counter_emcute
check_bin	andreamazzitelli/checkBin	RiotCode
sniffer	fu-ilab-swp2021/LoRa-Packet-Sniffer	b-l072z
ledext	ichatz/riotos-apps	ledext
relay_coap	ichatz/riotos-apps	relay_coap
temp_hum	ichatz/riotos-apps	temperature_humidity
tft_mqtts	ichatz/riotos-apps	tft_mqtts
udp1	ichatz/riotos-apps	udp_usb
bitfield	miri64/RIOT_playground	bitfield_test/bitfield
udp2	miri64/RIOT_playground	udp_test
sizeof_pktsnip	miri64/RIOT_playground	sizeof_pktsnip
aes	deus778/riot-aes-benchmark	
thread-duel	kfessel/riot-thread-duel	
election_master	maconard/RIOT_leader-election	cpsiot_masternode
election_worker	maconard/RIOT_leader-election	cpsiot_workernode
udptxrx	induarun9086/RIOT_UDP_EchoServerExample	udptxrx
echoserver	induarun9086/RIOT_UDP_EchoServerExample	udpechoserver
bus_monitor	FrancescoCrino/ConnectedBusMonitor	src/proto_ethos
perf_analysis	StefanoMilani/RIOT-OS-examples	performance-analysis
ttn	yegorich/ttn-mapper-riot	
spectrum	RIOT-OS/applications	spectrum-scanner
app03	Ciusss89/gtip-riotos	test_03

trust anchors or verification processes. The evaluation of 50 firmware samples for RIOT OS on two boards (i.e., *native* and *iotlab-m3*) demonstrated the applicability and efficacy of the tool and the proposed protection scheme at the cost of a reasonable size overhead of the firmware image.

As future extensions of this work, we plan to extend the protection scheme by adding multi-patter (i.e., heterogeneous) AT controls and evaluate the impact of the PARIOT protection scheme on the repackaging attack steps presented in Fig. 3. Also, we would like to measure the computational and energy footprint of the protection scheme on resource-constrained IoT devices and extend the support to other IoT OSes, e.g., Contiki-NG and FreeRTOS.

CRediT authorship contribution statement

Luca Verderame: Conceptualization, Methodology, Writing – original draft, Writing – review & editing. **Antonio Ruggia:** Conceptualization, Methodology, Software, Investigation, Writing – original draft, Writing – review & editing. **Alessio Merlo:** Conceptualization, Supervision, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The data and code required to reproduce the above findings are available at https://github.com/Mobile-IoT-Security-Lab/PARIOTIC.

Acknowledgements

This work was partially supported by the Curiosity Driven grant "Security Assessment of Cross-domain Application Ecosystems" of the University of Genova funded by the EU - NGEU.

Appendix. Experimental dataset

The dataset used for evaluating PARIOTIC consists of 50 firmware images built using RIOT OS 2021.05 and a different user app that has

Table A.4
RIOT OS Modules included in the firmware images of the dataset.

Firmware name	Included RIOT modules
asym_mqtt	$BOARD$, asymcute, auto_init, auto_init_gnrc_netif, gnrc_icmpv6_echo, gnrc_ipv6_default, netdev_default, ps, shell, shell_cmds_default
ccn_lite	$BOARD$, auto_init, auto_init_gnrc_netif, ccn-lite, gnrc_pktdump, netdev_default, prng_xorshift, ps, shell, shell_cmds_default
cord_ep	$BOARD$, auto_init, auto_init_gnrc_netif, cord_ep_standalone, fmt, gnrc_icmpv6_echo, gnrc_ipv6_default, netdev_default, ps, shell, shell_cmds_default
cord_epsim	$BOARD$, auto_init, auto_init_gnrc_netif, cord_epsim, gnrc_ipv6_default, netdev_default, xtimer
cord_lc	$BOARD$, auto_init, auto_init_gnrc_netif, cord_lc, gnrc_icmpv6_echo, gnrc_ipv6_default, netdev_default, ps, shell, shell_cmds_default
default	$BOARD$, auto_init, auto_init_gnrc_netif, gnrc, gnrc_pktdump, gnrc_txtsnd, mci, netdev_default, ps, random, saul_default, schedstatistics, shell, shell_cmds_default
decho	$BOARD$, auto_init, auto_init_gnrc_netif, gnrc_ipv6_default, netdev_default, prng_sha1prng, shell, shell_cmds_default, sock_udp, tinydtls
dsock	$BOARD$, auto_init, auto_init_gnrc_neti, gnrc_ipv6_default, netdev_default, prng_sha1prng, shell, shell_cmds_default, sock_dtls, sock_udp, sock_util, tinydtls
dwolfssl	$BOARD$, auto_init, auto_init_gnrc_netif, gnrc_ipv6_default, netdev_default, shell, shell_cmds_default, sock_udp, wolfcrypt, wolfcrypt_dh, wolfcrypt_ecc, wolfcrypt_rsa, wolfssl, wolfssl_dtls, wolfssl_psk
em_mqtt	$BOARD$, auto_init, auto_init_gnrc_netif, emcute, gnrc_icmpv6_echo, gnrc_ipv6_default, gnrc_netif_single, netdev_default, ps, shell, shell_cmds_default
filesystem	$BOARD$, auto_init, constfs, devfs, ps, shell, shell_cmds_default, vfs_auto_format, vfs_default
gcoap	$BOARD$, auto_init, auto_init_gnrc_netif, fmt, gcoap, gnrc_icmpv6_echo, gnrc_ipv6_default, ipv4_addr, ipv6_addr, lwip_arp, lwip_dhcp_auto, lwip_ipv4, lwip_ipv6, lwip_ipv6_autoconfig, lwip_netdev, netdev_default, netutils, od, ps, random, shell, shell_cmds_default, socket_zep
gnrc_brouter	$BOARD$, auto_init, auto_init_gnrc_netif, gnrc_dhcpv6_client_6lbr, gnrc_icmpv6_echo, gnrc_ipv6_auto_subnets_simple,gnrc_ipv6_nib_dns, gnrc_rpl, gnrc_sixlowpan_border_router_default, gnrc_uhcpc, netdev_default, ps, shell, shell_cmds_default, sock_dns
gnrc_net	$BOARD$, auto_init, auto_init_gnrc_netif, auto_init_gnrc_rpl, gnrc_icmpv6_echo, gnrc_icmpv6_error, gnrc_ipv6_nib_dns, gnrc_ipv6_router_default, gnrc_rpl, netdev_default, netstats_ipv6, netstats_l2, netstats_rpl, ps, shell, shell_cmd_gnrc_udp, shell_cmds_default, sock_dns, socket_zep
gnrc_net_mac	$BOARD$, auto_init, auto_init_gnrc_netif, auto_init_gnrc_rpl, gnrc_gomach, gnrc_icmpv6_echo, gnrc_ipv6_router_default, gnrc_lwmac, gnrc_pktdump, gnrc_rpl, gnrc_udp, netdev_default, netstats_ipv6, netstats_l2, netstats_rpl, ps, shell, shell_cmds_default
lua_basic	$BOARD$, auto_init, lua
lua_REPL	$BOARD$, auto_init, lua
micropython	$BOARD$, auto_init, micropython
nanocoap	$BOARD$, auto_init, auto_init_gnrc_netif, fmt, gnrc_icmpv6_echo, gnrc_ipv6_default, hashes, nanocoap_sock, netdev_default, prng_minstd, sock_udp, xtimer
ndn	$BOARD$, auto_init, auto_init_gnrc_netif, ndn-riot, netdev_default, random, shell, shell_cmds_default
paho	$BOARD$, auto_init, auto_init_gnrc_netif, gnrc_icmpv6_echo, gnrc_icmpv6_error, gnrc_ipv6_default, ipv4_addr, ipv6_addr, lwip, lwip_arp, lwip_dhcp_auto, lwip_ipv4, lwip_ipv6_autoconfig, lwip_netdev, netdev_default, ps, shell, shell_cmds_default, sock_async_event, sock_ip, sock_tcp, sock_udp, ztimer, ztimer_msec
pselect	$BOARD$, auto_init, auto_init_gnrc_netif, gnrc_ipv6_default, netdev_default, posix_inet, posix_select, posix_sockets, sock_udp
psocket	$BOARD$, auto_init, auto_init_gnrc_netif, gnrc_ipv6_default, netdev_default, posix_inet, posix_sleep, posix_sockets, ps, shell, shell_cmds_default, sock_udp
cpp	$BOARD$, auto_init, cpp, libstdcpp
saul	$BOARD$, auto_init, ps, saul_default, shell, shell_cmds_default
tpw	$BOARD$, auto_init, ztimer_msec
room_counter	$BOARD$, auto_init, auto_init_gnrc_netif, emcute, gnrc_icmpv6_echo, gnrc_ipv6_default, gnrc_netdev_default, gnrc_sock_udp, ps, shell, shell_commands, xtimer
room_infrared	$BOARD$, auto_init, auto_init_gnrc_netif, emcute, gnrc_icmpv6_echo, gnrc_ipv6_default, gnrc_netdev_default, gnrc_sock_udp, ps, shell, shell_commands, xtimer
museum	$BOARD$, auto_init, auto_init_gnrc_netif, emcute, gnrc_icmpv6_echo, gnrc_ipv6_default, gnrc_netdev_default, gnrc_sock_udp, ps, shell, shell_commands, xtimer

(*continued on next page*)

Table A.4 (*continued*).

check_bin	$BOARD$, $DRIVER$, auto_init, auto_init_loramac, fmt, periph_gpio, periph_gpio_irq, periph_i2c, semtech-loramac, semtech_loramac_rx, shell, shell_commands, u8g2, xtimer, ztimer, ztimer_usec
sniffer	$BOARD$, $DRIVER$, auto_init, fatfs_vfs, mtd_sdcard, periph_gpio_irq, vfs, xtimer
ledex	$BOARD$, auto_init, periph_gpio, xtimer
relay_coap	$BOARD$, auto_init, periph_gpio, xtimer
temp_hum	$BOARD$, auto_init, dht, fmt, periph_rtc
tft_mqtt	$BOARD$, auto_init, auto_init_gnrc_netif, emcute, gnrc_ipv6_default, gnrc_netdev_default, gnrc_netif_single, gnrc_uhcpc, periph_gpio, periph_spi, stdio_ethos, ucglib, xtimer
udp1	$BOARD$, auto_init, auto_init_gnrc_netif, gnrc_icmpv6_echo, gnrc_ipv6_default, gnrc_netdev_default, gnrc_uhcpc, stdio_ethos, xtimer
bitfield	$BOARD$, auto_init
udp2	$BOARD$, auto_init, auto_init_gnrc_netif, auto_init_gnrc_rpl, gnrc_icmpv6_echo, gnrc_ipv6_router_default,gnrc_netdev_default, gnrc_rpl, gnrc_udp, netstats_ipv6, netstats_l2, od, ps, shell, shell_commands
sizeof_pktsnip	fmt, $BOARD$
aes	$BOARD$, auto_init, cipher_modes, crypto_aes_128, crypto_aes_192, crypto_aes_256, od, od_string, random, shell, shell_commands, xtimer
thread-duel	$BOARD$, auto_init, sched_cb, sema, xtimer
election_master	$BOARD$, auto_init, auto_init_gnrc_netif, auto_init_gnrc_rpl, gnrc, gnrc_icmpv6_echo, gnrc_icmpv6_error, gnrc_ipv6_default, gnrc_ipv6_router_default, gnrc_netdev_default, gnrc_pktdump, gnrc_rpl,gnrc_sock_udp, gnrc_txtsnd, gnrc_udp, netstats_ipv6, netstats_l2, netstats_rpl,periph_rtc, ps, random, schedstatistics, shell, shell_commands, xtimer
election_worker	$BOARD$, auto_init, auto_init_gnrc_netif, auto_init_gnrc_rpl, gnrc, gnrc_icmpv6_echo, gnrc_icmpv6_error, gnrc_ipv6_default, gnrc_ipv6_router_default, gnrc_netdev_default, gnrc_pktdump, gnrc_rpl,gnrc_sock_udp, gnrc_txtsnd, gnrc_udp, netstats_ipv6, netstats_l2, netstats_rpl,periph_rtc, ps, random, schedstatistics, shell, shell_commands, xtimer
udptxrx	$BOARD$, auto_init, auto_init_gnrc_netif, auto_init_gnrc_rpl, gnrc_ipv6_default, gnrc_netdev_default, gnrc_rpl, gnrc_sock_udp, gnrc_udp
echoserver	$BOARD$, auto_init, auto_init_gnrc_netif, auto_init_gnrc_rpl, gnrc_ipv6_default, gnrc_netdev_default, gnrc_rpl, gnrc_sock_udp, gnrc_udp
bus_monitor	$BOARD$, $DRIVER$, auto_init, auto_init_gnrc_netif, emcute, gnrc_icmpv6_echo, gnrc_ipv6_default, gnrc_netdev_default, gnrc_uhcpc, hts221, ps, shell, shell_commands, stdio_ethos, xtimer, ztimer, ztimer_msec
perf_analysis	$BOARD$, auto_init, crypto, prng_minstd, ps, shell, shell_commands, xtimer
ttn	$BOARD$, auto_init, minmea, periph_uart, xtimer
spectrum	$BOARD$, auto_init, auto_init_gnrc_netif, fmt, gnrc, netdev_default, xtimer, ztimer64_xtimer_compat
app03	$BOARD$, auto_init, ps, shell, shell_commands, uptime,xtimer

a publicly-available source repository on GitHub. The dataset samples use more than 130 different RIOT modules, and each app includes, on average, 12 modules with a standard deviation of 6.

Table A.3 reports the name of each app, the link to the GitHub project, and the subfolder that contains the source code. Table A.4 details the list of all the RIOT OS modules included for each firmware in the dataset. We also specified $BOARD$ and $DRIVER$ to collectively refer to the additional modules needed by a firmware to execute in a particular board/driver.

References

Adjih, C., Baccelli, E., Fleury, E., Harter, G., Mitton, N., Noel, T., Pissard-Gibollet, R., Saint-Marcel, F., Schreiner, G., Vandaele, J., et al., 2015. FIT IoT-LAB: A large scale open experimental IoT testbed. In: 2015 IEEE 2nd World Forum on Internet of Things (WF-IoT). IEEE, pp. 459–464.

Ahmadvand, M., Pretschner, A., Kelbert, F., 2019. In: Memon, A.M. (Ed.), Chapter Eight - A Taxonomy of Software Integrity Protection Techniques. In: Advances in Computers, vol. 112, Elsevier, pp. 413–486. http://dx.doi.org/10.1016/bs.adcom.2017.12.007, URL https://www.sciencedirect.com/science/article/pii/S0065245817300591.

Al-Wosabi, A.A.A., Shukur, Z., Ibrahim, M.A., 2015. Framework for software tampering detection in embedded systems. In: 2015 International Conference on Electrical Engineering and Informatics. ICEEI, IEEE, pp. 259–264.

Anastasiou, A., Christodoulou, P., Christodoulou, K., Vassiliou, V., Zinonos, Z., 2020. Iot device firmware update over lora: The blockchain solution. In: 2020 16th International Conference on Distributed Computing in Sensor Systems. DCOSS, IEEE, pp. 404–411.

Arakadakis, K., Charalampidis, P., Makrogiannakis, A., Fragkiadakis, A., 2020. Firmware over-the-air programming techniques for IoT networks–a survey. arXiv preprint arXiv:2009.02260.

Aschenbruck, N., Bauer, J., Bieling, J., Bothe, A., Schwamborn, M., 2012. Selective and secure over-the-air programming for wireless sensor networks. In: 2012 21st International Conference on Computer Communications and Networks. ICCCN, pp. 1–6. http://dx.doi.org/10.1109/ICCCN.2012.6289278.

Asokan, N., Nyman, T., Rattanavipanon, N., Sadeghi, A.-R., Tsudik, G., 2018. ASSURED: Architecture for secure software update of realistic embedded devices. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 37 (11), 2290–2300.

Baccelli, E., Gündoğan, C., Hahm, O., Kietzmann, P., Lenders, M.S., Petersen, H., Schleiser, K., Schmidt, T.C., Wählisch, M., 2018. RIOT: An open source operating system for low-end embedded devices in the IoT. IEEE Internet Things J. 5 (6), 4428–4440.

Baccelli, E., Hahm, O., Günes, M., Wählisch, M., Schmidt, T.C., 2013. RIOT OS: Towards an OS for the internet of things. In: 2013 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS). IEEE, pp. 79–80.

brianpow, 2015. Firmware modification kit. [Online; accessed April 28, 2023] https://github.com/brianpow/firmware-mod-kit.

Brumley, D., Hartwig, C., Liang, Z., Newsome, J., Song, D., Yin, H., 2008. In: Lee, W., Wang, C., Dagon, D. (Eds.), Automatically Identifying Trigger-based Behavior in Malware. Springer US, Boston, MA, pp. 65–88. http://dx.doi.org/10.1007/978-0-387-68768-1_4.

Carrillo-Mondéjar, J., Turtiainen, H., Costin, A., Martínez, J., Suarez-Tangil, G., 2022. HALE-IoT: Hardening legacy internet-of-things devices by retrofitting defensive firmware modifications and implants. IEEE Internet Things J..

Chandra, H., Anggadjaja, E., Wijaya, P.S., Gunawan, E., 2016. Internet of things: Over-the-air (OTA) firmware update in lightweight mesh network protocol for smart

urban development. In: 2016 22nd Asia-Pacific Conference on Communications. APCC, pp. 115–118. http://dx.doi.org/10.1109/APCC.2016.7581459.
Choi, S., Lee, J.-H., 2020. Blockchain-based distributed firmware update architecture for IoT devices. IEEE Access 8, 37518–37525.
Christensen, J., Anghel, I.M., Taglang, R., Chiroiu, M., Sion, R., 2020. {DecAF}: Automatic, adaptive de-bloating and hardening of {coTS} firmware. In: 29th USENIX Security Symposium (USENIX Security 20). pp. 1713–1730.
Computer Security Laboratory, 2022. PARIOT. [Online; accessed April 28, 2023] https://github.com/Mobile-IoT-Security-Lab/PARIOT.
Costin, A., Zaddach, J., Francillon, A., Balzarotti, D., 2014. A {large-scale} analysis of the security of embedded firmwares. In: 23rd USENIX Security Symposium (USENIX Security 14). pp. 95–110.
Costin, A., Zarras, A., Francillon, A., 2016. Automated dynamic firmware analysis at scale: a case study on embedded web interfaces. In: Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security. pp. 437–448.
Craig Smith, 2015. Firmwalker. [Online; accessed April 28, 2023] https://github.com/craigz28/firmwalker.
Cui, A., Costello, M., Stolfo, S., 2013. When firmware modifications attack: A case study of embedded exploitation.
David, Y., Partush, N., Yahav, E., 2018. Firmup: Precise static detection of common vulnerabilities in firmware. ACM SIGPLAN Notices 53 (2), 392–404.
de Sousa, M.J.B., Gonzalez, L.F.G., Ferdinando, E.M., Borin, J.F., 2022. Over-the-air firmware update for IoT devices on the wild. Internet Things 19, 100578.
Dejon, N., Caputo, D., Verderame, L., Armando, A., Merlo, A., 2019. Automated security analysis of IoT software updates. In: IFIP International Conference on Information Security Theory and Practice. Springer, pp. 223–239.
Dhakal, S., Jaafar, F., Zavarsky, P., 2019. Private blockchain network for IoT device firmware integrity verification and update. In: 2019 IEEE 19th International Symposium on High Assurance Systems Engineering. HASE, pp. 164–170. http://dx.doi.org/10.1109/HASE.2019.00033.
Dhobi, R., Gajjar, S., Parmar, D., Vaghela, T., 2019. Secure firmware update over the air using TrustZone. In: 2019 Innovations in Power and Advanced Computing Technologies (I-PACT), Vol. 1. pp. 1–4. http://dx.doi.org/10.1109/i-PACT44901.2019.8959992.
Doddapaneni, K., Lakkundi, R., Rao, S., Kulkarni, S.G., Bhat, B., 2017. Secure fota object for iot. In: 2017 IEEE 42nd Conference on Local Computer Networks Workshops (LCN Workshops). IEEE, pp. 154–159.
Doroodgar, F., Razzaque, M.A., Isnin, I.F., 2014. Seluge++: A secure over-the-air programming scheme in wireless sensor networks. Sensors 14 (3), 5004–5040.
Dronebl, 2008. Network bluepill. [Online; accessed April 28, 2023] http://www.dronebl.org/blog/8.
Dunkels, A., Gronvall, B., Voigt, T., 2004. Contiki-a lightweight and flexible operating system for tiny networked sensors. In: 29th Annual IEEE International Conference on Local Computer Networks. IEEE, pp. 455–462.
Dutta, P.K., Hui, J.W., Chu, D.C., Culler, D.E., 2006. Securing the deluge network programming system. In: 2006 5th International Conference on Information Processing in Sensor Networks. IEEE, pp. 326–333.
El Jaouhari, S., Bouvet, E., 2022. Secure firmware over-the-air updates for IoT: Survey, challenges, and discussions. Internet Things 18, 100508.
Eldefrawy, K., Tsudik, G., Francillon, A., Perito, D., 2012. Smart: secure and minimal architecture for (establishing dynamic) root of trust. In: Ndss, Vol. 12. pp. 1–15.
Falas, S., Konstantinou, C., Michael, M.K., 2021. A modular end-to-end framework for secure firmware updates on embedded systems. ACM J. Emerg. Technol. Comput. Syst. (JETC) 18 (1), 1–19.
Foundation, A., 2018. Nuttx OS. [Online; accessed April 28, 2023] https://nuttx.apache.org.
Fratantonio, Y., Bianchi, A., Robertson, W., Kirda, E., Kruegel, C., Vigna, G., 2016. Triggerscope: Towards detecting logic bombs in android applications. In: 2016 IEEE Symposium on Security and Privacy. SP, pp. 377–396. http://dx.doi.org/10.1109/SP.2016.30.
Fraunhofer FKIE, 2015. The firmware analysis and comparison tool (FACT). [Online; accessed April 28, 2023] https://github.com/craigz28/firmwalker.
FreeBSD, 2012. Security incident on freebsd infrastructure. [Online; accessed April 28, 2023] https://www.freebsd.org/news/2012-compromise/.
Ghosal, A., Halder, S., Conti, M., 2022. Secure over-the-air software update for connected vehicles. Comput. Netw. 218, 109394.
GitHub, 2022. GitHub. [Online; accessed April 28, 2023] https://github.com/.
Gupta, A., 2019. The IoT Hacker's Handbook. Springer.
Gupta, H., van Oorschot, P.C., 2019. Onboarding and software update architecture for IoT devices. In: 2019 17th International Conference on Privacy, Security and Trust. PST, pp. 1–11. http://dx.doi.org/10.1109/PST47121.2019.8949023.
Holt, A., Huang, C.-Y., 2014. Openwrt. In: Embedded Operating Systems. Springer, pp. 161–181.
Hu, J.-W., Yeh, L.-Y., Liao, S.-W., Yang, C.-S., 2019. Autonomous and malware-proof blockchain-based firmware update platform with efficient batch verification for internet of things devices. Comput. Secur. 86, 238–252.
Hui, J.W., Culler, D., 2004. The dynamic behavior of a data dissemination protocol for network programming at scale. In: Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems. pp. 81–94.
Hyun, S., Ning, P., Liu, A., Du, W., 2008. Seluge: Secure and dos-resistant code dissemination in wireless sensor networks. In: 2008 International Conference on Information Processing in Sensor Networks (Ipsn 2008). IEEE, pp. 445–456.
Internet Engineering Task Force (IETF), 2018. Software updates for internet of things (suit). [Online; accessed April 28, 2023] https://datatracker.ietf.org/wg/suit/documents/.
Karthik, T., Brown, A., Awwad, S., McCoy, D., Bielawski, R., Mott, C., Lauzon, S., Weimerskirch, A., Cappos, J., 2016. Uptane: Securing software updates for automobiles. In: International Conference on Embedded Security in Car. pp. 1–11.
Kerliu, K., Ross, A., Tao, G., Yun, Z., Shi, Z., Han, S., Zhou, S., 2019. Secure over-the-air firmware updates for sensor networks. In: 2019 IEEE 16th International Conference on Mobile Ad Hoc and Sensor Systems Workshops. MASSW, pp. 97–100. http://dx.doi.org/10.1109/MASSW.2019.00026.
Khan, R., McLaughlin, K., Laverty, D., Sezer, S., 2017. STRIDE-based threat modeling for cyber-physical systems. In: 2017 IEEE PES Innovative Smart Grid Technologies Conference Europe (ISGT-Europe). IEEE, pp. 1–6.
Kumar, S.K., Sahoo, S., Kiran, K., Swain, A.K., Mahapatra, K., 2018. A novel holistic security framework for in-field firmware updates. In: 2018 IEEE International Symposium on Smart Electronic Systems (ISES) (Formerly INiS). pp. 261–264. http://dx.doi.org/10.1109/iSES.2018.00063.
Langiu, A., Boano, C.A., Schuß, M., Römer, K., 2019. Upkit: An open-source, portable, and lightweight update framework for constrained IoT devices. In: 2019 IEEE 39th International Conference on Distributed Computing Systems. ICDCS, IEEE, pp. 2101–2112.
Lanigan, P.E., Gandhi, R., Narasimhan, P., 2006. Sluice: Secure dissemination of code updates in sensor networks. In: 26th IEEE International Conference on Distributed Computing Systems (ICDCS'06). IEEE, p. 53.
Lee, B., Lee, J.-H., 2017. Blockchain-based secure firmware update for embedded devices in an internet of things environment. J. Supercomput. 73 (3), 1152–1167.
Linux, 2021. Mprotect. [Online; accessed April 28, 2023] https://man7.org/linux/man-pages/man2/mprotect.2.html.
Luo, L., Fu, Y., Wu, D., Zhu, S., Liu, P., 2016. Repackage-proofing android apps. In: 2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. DSN, pp. 550–561.
Maroof, U., Shaghaghi, A., Michelin, R., Jha, S., 2022. Irecover: Patch your IoT on-the-fly. Future Gener. Comput. Syst. 132, 178–193.
Mbakoyiannis, D., Tomoutzoglou, O., Kornaros, G., 2019. Secure over-the-air firmware updating for automotive electronic control units. In: Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing. pp. 174–181.
Merlo, A., Ruggia, A., Sciolla, L., Verderame, L., 2021. You shall not repackage! demystifying anti-repackaging on android. Comput. Secur. 103, 102181. http://dx.doi.org/10.1016/j.cose.2021.102181.
Mtetwa, N.S., Tarwireyi, P., Abu-Mahfouz, A.M., Adigun, M.O., 2019. Secure firmware updates in the internet of things: A survey. In: 2019 International Multidisciplinary Information Technology and Engineering Conference. IMITEC, IEEE, pp. 1–7.
Nguyen, K.T., Laurent, M., Oualha, N., 2015. Survey on secure communication protocols for the internet of things. Ad Hoc Netw. 32, 17–31.
Nikitin, K., Kokoris-Kogias, E., Jovanovic, P., Gailly, N., Gasser, L., Khoffi, I., Cappos, J., Ford, B., 2017. {ChainiAC}: Proactive {software-update} transparency via collectively signed skipchains and verified builds. In: 26th USENIX Security Symposium (USENIX Security 17). pp. 1271–1287.
Panchal, A.C., Khadse, V.M., Mahalle, P.N., 2018. Security issues in iIoT: A comprehensive survey of attacks on iIoT and its countermeasures. In: 2018 IEEE Global Conference on Wireless Computing and Networking. GCWCN, IEEE, pp. 124–130.
Perito, D., Tsudik, G., 2010. Secure code update for embedded devices via proofs of secure erasure. In: European Symposium on Research in Computer Security. Springer, pp. 643–662.
Pillai, A., Sindhu, M., Lakshmy, K., 2019. Securing firmware in internet of things using blockchain. In: 2019 5th International Conference on Advanced Computing & Communication Systems. ICACCS, pp. 329–334. http://dx.doi.org/10.1109/ICACCS.2019.8728389.
Prada-Delgado, M.A., Vázquez-Reyes, A., Baturone, I., 2017. Trustworthy firmware update for internet-of-thing devices using physical unclonable functions. In: 2017 Global Internet of Things Summit (GIoTS). pp. 1–5. http://dx.doi.org/10.1109/GIOTS.2017.8016282.
ReFirm Labs, 2014. Binwalk. [Online; accessed April 28, 2023] https://github.com/ReFirmLabs/binwalk.
Register, T., 2022. Software developer cracks hyundai car security with google search. [Online; accessed April 28, 2023] https://www.theregister.com/2022/08/17/software_developer_cracks_hyundai_encryption/.
RIOT OS, 2022a. IoT-LAB M3 open node. [Online; accessed April 28, 2023] https://doc.riot-os.org/group_boards_iotlab-m3.html.
RIOT OS, 2022b. Modules. [Online; accessed April 28, 2023] https://api.riot-os.org/modules.html.
RIOT OS, 2022c. Native board. [Online; accessed April 28, 2023] https://doc.riot-os.org/group_boards_native.html.
RIOT OS, 2023. IoT-LAB consumption monitoring. [Online; accessed April 28, 2023] https://iot-lab.github.io/docs/tools/consumption-monitoring/.
Ronen, E., Shamir, A., Weingarten, A.-O., O'Flynn, C., 2017. Iot goes nuclear: Creating a ZigBee chain reaction. In: 2017 IEEE Symposium on Security and Privacy. SP, IEEE, pp. 195–212.

Sahlmann, K., Clemens, V., Nowak, M., Schnor, B., 2020. MUP: Simplifying secure over-the-air update with MQTT for constrained IoT devices. Sensors 21 (1), 10.

Salas, M., 2013. A secure framework for OTA smart device ecosystems using ECC encryption and biometrics. In: Advances in Security of Information and Communication Networks: First International Conference, SecNet 2013, Cairo, Egypt, September 3-5, 2013. Proceedings. Springer, pp. 204–218.

Samuel, J., Mathewson, N., Cappos, J., Dingledine, R., 2010. Survivable key compromise in software update systems. In: Proceedings of the 17th ACM Conference on Computer and Communications Security. pp. 61–72.

Schüll, N.D., 2016. Data for life: Wearable technology and the design of self-care. BioSocieties 11 (3), 317–333.

Sharif, M.I., Lanzi, A., Giffin, J.T., Lee, W., 2008. Impeding malware analysis using conditional code obfuscation. In: NDSS. Citeseer.

Shim, J., Jung, K., Cho, S., Park, M., Han, S., 2017. A case study on vulnerability analysis and firmware modification attack for a wearable fitness tracker. IT Converg. Pract. 5 (4), 25–33.

SICS, R., 2018. The sparrow application layer and tools. [Online; accessed April 28, 2023] https://github.com/sics-iot/sparrow.

Stanford-Clark, A., Nipper, A., 2022. MQTT protocol. [Online; accessed April 28, 2023] https://mqtt.org/.

Teng, C.-C., Gong, J.-W., Wang, Y.-S., Chuang, C.-P., Chen, M.-C., 2017. Firmware over the air for home cybersecurity in the internet of things. In: 2017 19th Asia-Pacific Network Operations and Management Symposium. APNOMS, IEEE, pp. 123–128.

ThreatPost, 2015. D-link accidentally leaks private code-signing keys. [Online; accessed April 28, 2023] https://threatpost.com/d-link-accidentally-leaks-private-code-signing-keys/114727/.

Tsaur, W.-J., Chang, J.-C., Chen, C.-L., 2022. A highly secure IoT firmware update mechanism using blockchain. Sensors 22 (2), 530.

Vasile, S., Oswald, D., Chothia, T., 2018. Breaking all the things—A systematic survey of firmware extraction techniques for IoT devices. In: International Conference on Smart Card Research and Advanced Applications. Springer, pp. 171–185.

Wenzl, M., Merzdovnik, G., Ullrich, J., Weippl, E., 2019. From hack to elaborate technique—a survey on binary rewriting. ACM Comput. Surv. 52 (3), 1–37.

Witanto, E.N., Oktian, Y.E., Kumi, S., Lee, S.-G., 2019. Blockchain-based OCF firmware update. In: 2019 International Conference on Information and Communication Technology Convergence. ICTC, pp. 1248–1253. http://dx.doi.org/10.1109/ICTC46691.2019.8939910.

Yohan, A., Lo, N.-W., 2018. An over-the-blockchain firmware update framework for IoT devices. In: 2018 IEEE Conference on Dependable and Secure Computing. DSC, IEEE, pp. 1–8.

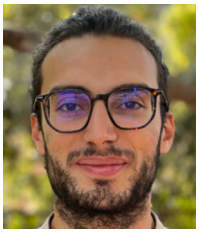
Yohan, A., Lo, N.-W., 2020. FOTB: a secure blockchain-based firmware update framework for IoT environment. Int. J. Inf. Secur. 19 (3), 257–278.

Zandberg, K., Schleiser, K., Acosta, F., Tschofenig, H., Baccelli, E., 2019. Secure firmware updates for constrained iot devices using open standards: A reality check. IEEE Access 7, 71907–71920.

ZDNet, 2016. Surveillance cameras sold on amazon infected with malware. [Online; accessed April 28, 2023] https://www.zdnet.com/article/amazon-surveillance-cameras-infected-with-malware/.

Zeng, Q., Luo, L., Qian, Z., Du, X., Li, Z., 2018. Resilient decentralized android application repackaging detection using logic bombs. In: Proceedings of the 2018 International Symposium on Code Generation and Optimization. In: CGO 2018, Association for Computing Machinery, New York, NY, USA, pp. 50–61. http://dx.doi.org/10.1145/3168820.

Luca Verderame is an Assistant Professor in Computer Engineering at the University of Genoa (Italy). He obtained his Ph.D. in Electronic, Information, Robotics, and Telecommunication Engineering in 2016, where he worked on mobile security. His research interests mainly cover the security of software supply chains and application ecosystems. Luca is also the CEO and Co-founder of Talos, a cybersecurity SME, and university spin-off.



Antonio Ruggia is a Ph.D. student in Security, Risk, and Vulnerability at the University of Genoa since November 2020, supervised by prof. Alessio Merlo. He is interested in several security topics, including Mobile Security, with a specific interest in Android, malware, and data protection. He graduated in October 2020 from the University of Genoa and participated in the 2019 CyberChallenge.it, an Italian practical competition for students in Cybersecurity. Since 2018, he has worked as a full-stack developer in a multinational corporation.

Alessio Merlo is a Full Professor in Computer Engineering at CASD, the Centre for Advanced Defence Studies (CASD) in Rome, Italy. He received the Ph.D. degree in Computer Science from the University of Genoa in 2010. His main research interest is Cybersercurity, with a specific focus on Mobile, where he contributed to discovering several high-risk vulnerabilities both in applications and the Android OS, and Systems Security. He has published more than 130 scientific papers in international conferences and journals.