Over-the-Air Software Updates in the Internet-of-Things: An Overview of Key Principles.

Jan Bauwens, Peter Ruckebusch, Spilios Giannoulis, Ingrid Moerman, and Eli De Poorter

***Abstract*—Due to the fast pace at which Internet-of-Things (IoT) protocols and applications evolve, there is an increasing need to support over-the-air software updates for security updates, bug fixes and software extensions. To this end, multiple over-the-air techniques have been proposed each covering a specific aspect of the update process, such as (partial) code updates, data dissemination and security. However, each technique introduces an overhead, especially in terms of energy consumption, thereby impacting the operational lifetime of the constrained battery powered devices. Up until now, a comprehensive overview describing the different update steps and quantifying the impact of each step is missing in scientific literature, making it hard to assess the overall feasibility of an over-the-air update. To remedy this, our article (i) analyzes which parts of an IoT operating system are most updated after device deployment, (ii) proposes a step-by-step approach to integrate software updates in IoT solutions, and (iii) quantifies the energy cost of each of the involved step. The results show that besides the obvious dissemination cost, other phases such as security also introduce a significant overhead. For instance, a typical firmware update requires 135.026mJ, of which the main portions are data dissemination (63.11%) and encryption (5.29%). However, when modular updates are used instead, the energy cost (e.g. for a MAC update) is reduced to 26.743mJ (of which 48.69% for data dissemination and 26.47% for encryption).**

***Index Terms*—Internet-of-Things, Sensor networks, Over-the-air software updates, Code dissemination, Update security, Network management.**

I. Introduction

THE Internet-of-Things (IoT) refers to the trend to include small, cheap and/or energy efficient wireless radios in everyday objects. Most of these IoT devices are constrained in terms of processing power and memory storage to keep the unit price low, as well as in terms of energy since they are typically battery powered. IoT solutions are already digitizing an increasing amount of functionalities of modern day society, impacting application areas such as health care, surveillance, agriculture, personal fitness, and home and industry automation. This trend will lead to a further increase in (i) the number of devices per person and (ii) the number of devices per square meter, thereby introducing the need for well designed and maintainable IoT solutions.

However, the specific device limitations, fast technology evolution and the increasing pace at which new devices are rolled out in difficult to reach areas raise questions concerning the long term sustainability of previously installed IoT networks. For instance, security issues or bugs are often detected post deployment, thereby hindering the operational IoT network. Moreover, already deployed devices cannot take advantage of new features and/or optimizations, or even adapt to new application requirements. A recent industry study showed that the frequency of field updates will significantly increase in the upcoming five to ten years, even with the possibility of monthly software updates [1].

Despite this increasing interest in over-the-air updates, scientific literature discussing the impact of these updates on energy consumption is limited. For example, [2] calculates the energy cost for update data transmission, but ignores security and reliable dissemination. Similarly, the operational impact of software updates on code versioning are not discussed. This paper offers a remedy by providing the steps required for enabling over-the-air software updates, while discussing the impact on constrained devices. For each of the steps, state-of-the-art techniques are discussed and evaluated.

In summary, this article contains the following contributions and insights:

- An analysis concerning the distribution of software development effort in different parts (applications, network, core OS and platform) of widely used IoT operating systems.
- A comprehensive overview of the key steps in an over-the-air update process is given, as well as an overview on recent update approaches (firmware based, modular with dynamic linking, modular with prelinking).
- The energy overhead per phase is quantified, showing the relative energy impact of the different deployment phases.
- A discussion is provided on the impact of updates on operational processes, such as the versioning approach used for software modules.
- Finally, the article lists future research directions that could enhance the potential of over-the-air updates for IoT devices.

II. Analysing update requirements in IoT operating systems

It is important to recognize parts of IoT solutions which evolve quickly and are hence more likely to require software updates. Figure 1a depicts the wireless stack of a typical sensor application, containing the software modules and their interaction with the (non-upgradable) hardware modules. The software modules are divided into four blocks: (i) sensor and actuator application software (blue), (ii) network protocol stack software (orange), (iii) operating system (OS) core software (grey), and (iv) platform hardware driver software

All authors are with the departement of Information Technology - IDLAB - Ghent University IMEC, Ghent, Oost-Vlaanderen, 9000, Belgium. e-mail: fistname.surname@ugent.be (except jan.bauwens2@ugent.be).

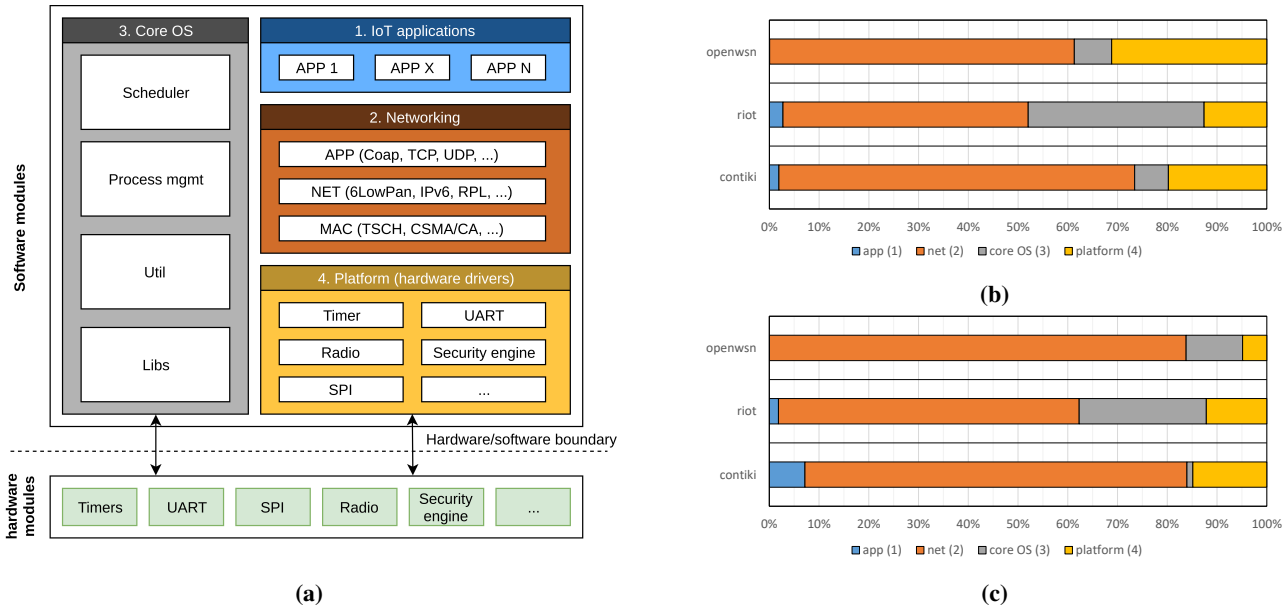

Fig. 1: Git commit statistics and memory usage of three operating systems for a typical IoT device. (a) The typical software components IoT stack (and their interface towards the hardware), divided in application code (1), networking functionality (2), OS core functionality (3), and platform specific code (4) (b) Relative memory size of the different components (%). (c) Relative Git commit statistics of the different software components, indicating the percentage of code lines changed between OS software releases (between versions 2018-01 and 2018.04 for RIoT, versions RB1.4 and RB1.8 for OpenWSN, and versions REL3.1 and the master branch in august 2018 for Contiki).

(yellow). Contrary to traditional software systems, the applications running on constrained IoT devices are relatively simple (i.e. sense or actuate) and therefore small in size. Most logic resides in the various network protocols enabling end-to-end communication with IoT devices. To demonstrate this, the memory usage and Git commit history for each block is calculated for different IoT operating systems on a Zolertia Re-Mote, which is a typical constrained IoT device (ARM Cortex-M3 32 MHz clock speed, 512 KB ROM and 32 KB RAM), as depicted on Figure 1b and 1c (in %). **The network protocol stack comprises a significant portion of the firmware, occupying between 50% and 72%.** The complexity of the network stack is the main reason for the larger code size. This has a direct consequence on the development effort. Moreover, since network protocol standards are frequently added or updated, there is a continuous push to keep the code up to date and include the latest features. This in contrast to the core OS and platform code as illustrated in Figure 1c, showing the Git commit statistics for two consecutive releases of three different IoT operating systems. The statistics include all code lines changed in the software modules required to build a firmware of a typical sensor application on a Zolertia Remote IoT device. **Between 60% and 84% of the code changes are related to the network protocol stack.** The rate at which new standards are being proposed seems to be increasing, and therefore the wireless stack will likely not achieve a completely "stable" state in the near future [3].

Although an IoT network operator would be mostly interested to enable IoT application updates, Figure 1b demonstrates that other code blocks actually comprise a larger portion of the firmware, and are hence at higher risk of containing bugs. Without network protocol updates, it is not possible to guarantee an optimal performance during the operational lifetime of the device. A sustainable IoT solution therefore adopts a continuously running development process, taking into account the rapid rate in which technology and business requirements change. In this process, software updates are essential in keeping an IoT solution up to date for the following reasons:

- Protocol and standard version updates, improving efficiency.
- Critical bug fixes and security updates, increasing availability and security.
- New applications, providing additional functionalities;
- Integration with third party IoT systems (hardware or software), extending the scope.
- Adopting new communication standards and protocols, improving performance and interoperability.

However, because the network stack on constrained devices is currently included in the OS, it can only be upgraded by means of a full firmware update, consuming a substantial amount of energy. Given its significance and the rapid change rate, we argue that partial updates of a network stack should also be possible, thereby lowering the energy cost for protocol updates.

III. Software Update Process

Next, we provide a step-by-step overview of the process required to enable over-the-air software updates of the above components in a secure and reliable manner. Figure 2 illustrates the sequence of interactions during a software update. A network operator initiates this procedure by downloading a new or updated module from a software repository. First, during the "SW Module Management" phase, the code is verified offline before being released. This phase includes (i) a compilation step, automatically adding linker metadata

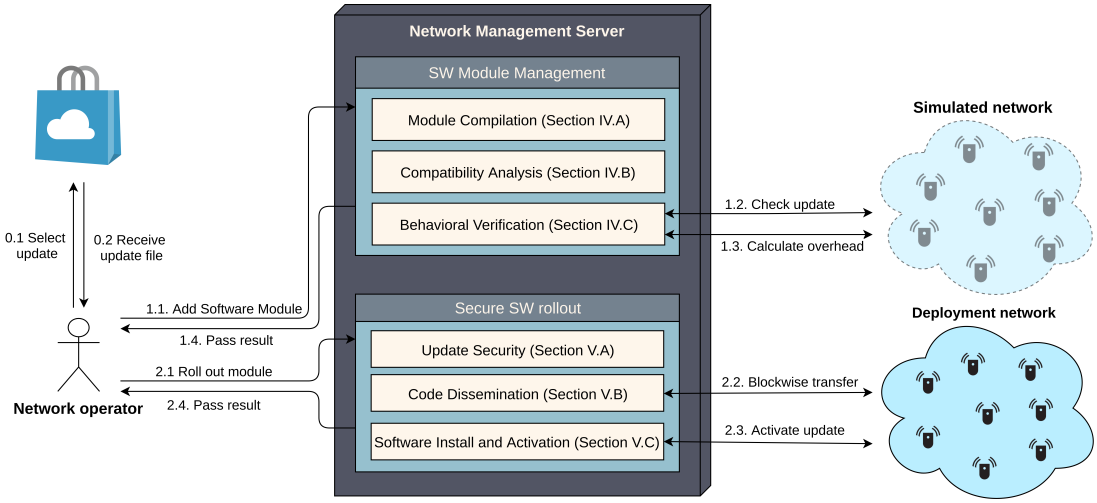

Fig. 2: Overview of the necessary steps to perform an over-the-air update. This process is split between (i) the management (compilation, validation and verification), of the software modules, and (ii) the secure software rollout process.

to the resulting binary module; (ii) a compatibility analysis step, checking compatibility with the already deployed modules maintained in a binary module repository; (iii) a functional verification step, verifying the new/update module in a simulation or digital twin network mirroring the actual network. On successful completion of the first phase, the "secure software rollout" phase can commence, which includes (i) a security step, encrypting and signing the binary module; (ii) a dissemination step, transferring the binary module to the devices; (iii) an activation step, simultaneously installing and making operational the binary module, and when needed safely rolling back the update on all devices in case of failure. Each of these steps is discussed in more detail in the next sections.

IV. Phase 1: Software Module Management

Updating the software on a remote wireless device is error prone: due to unforeseen code interactions the updated code might decrease the performance rather than improve the performance of a network, or even result in unstable operations. This could necessitate a rollback to a previous stable version. Since a wireless update is an intensive process in terms of medium usage, computational overhead and energy consumption (see Section V), it is important to automatically assess the validity of the update upfront, before actually performing the change and possible wasting valuable resources. For this purpose, a network operator will first pass the software modules to the SW module management services which include three distinct steps, each explained further in the remainder of this section: (i) module compilation, (ii) compatibility analysis checking the compatibility of software module versions, and (iii) qualitative predeployment analysis using a digital twin network or a sandbox. If all these steps are successfully completed, the network operator can initiate the secure rollout of the software module, explained in the next section.

A. Software Module Compilation

Module compilation ensures that new or updated software is transformed from source code into an efficient binary format that can be distributed to a running system. Three different over-the-air software compilation methods are available for constrained devices as discussed in [2]:

- Firmware based approaches replace the entire image. All source code is compiled into a single image and installed on each device. If an update is required, a new image must be compiled and distributed to all devices. Binary differential patching techniques (e.g. sending only the firmware differences) can be used in order to reduce the size of the image that needs to be transferred.
- Dynamic linking approaches require a linker on the constrained device to install or update software modules in an active system. The linker relocates code (data) section to the allocated ROM (RAM) memory regions and resolves undefined references to already present modules.
- Prelinking approaches offload the task of the dynamic linker to a more powerful server. An offline linker relocates and resolves undefined references before the modules are disseminated to the devices. This approach requires complete knowledge of the code and data memory location of each module installed on each device.

The latter two approaches, that is dynamic and prelinking, are modular and can be further categorized by their binding models [4], defining how code blocks are linked post deployment to the external functionality (functions, shared memory, etc.) provided by other modules:

- A linker that uses a strict binding model statically links code blocks to each other, replacing undefined symbols in one code block with the correct physical address of another code block. Because the physical addresses are hardcoded in memory, it is practically impossible to relink modules after installation if other modules are updated.
- A linker that uses a loosely coupled binding model employs an indirect function call mechanism and jump tables to redirect function calls between code blocks. By manipulating the jump tables, it is possible to update code blocks in the entire firmware even after installation. This comes at the cost of an increased complexity for jump table management and extra memory usage.

The choice of the update method and binding model has a considerable impact on the possible update scenarios (e.g. which code blocks can be updated) and the cost of the update in terms of bandwidth, latency and energy:

- Firmware updates allow to replace the entire code base but requires most bandwidth and, consequently, energy. The latency is also very high, especially because a reboot is required, potentially losing running network state.
- In all cases, prelinking outperforms dynamic linking because the resulting binary is smaller. This comes at the cost of additional computational complexity and requires that all devices have exactly the same firmware.
- A strict binding model only allows to update/add application level code blocks (i.e. blue part of Figure 1a) while a loosely coupled binding model [4] can update all code blocks, for example when including the network protocols.

B. Compatibility analysis

Because software modules are often developed independently of each other, some versions could prove incompatible with each other, leading to a degraded or broken network. As such, a versioning system needs to verify the compatibility of the different software modules. In case modular software updates are supported (e.g. only a single protocol or application is updated), the compatibility check system should also be applied on a module level.

This validation process can be split up into several subprocesses (see Figure 3). First, a compatibility check verifies if the software module can run on the target hardware platform. This is denoted as *'platform compatibility'*. Second, compatibility between the different software modules installed on a single device is checked. This process is referred to as *'inter-module compatibility'*. Last, the *'network compatibility'* ensures that multiple versions of the same software module can co-exist within the same network (e.g. multiple versions of IPv6 on different devices).

On traditional component upgradable software systems, such as OSGi [5], only inter-module compatibility is included. This is not sufficient when applying the partial update methods described in the previous section because single network layers can be updated separately. For instance, an updated MAC layer could rely on information exchanges (e.g. enhanced beacons in the IEEE-802.15.4 TSCH mode) that are not yet available in older versions of the PHY layer, rendering the new MAC version incompatible with the previous PHY version. To counter this, the central management server keeps track of the software version(s) installed on each device and ensures compatibility. A possible approach utilises a matrix for keeping track of compatibility, for which an example is shown in Figure 3. This compatibility matrix can be updated by performing tests ranging from static code verification, over simulations, to analysis in a real life deployment.

C. Pre-deployment behavioral verification

While the previous verification step primarily focuses on compatibility of software versions, this section describes a

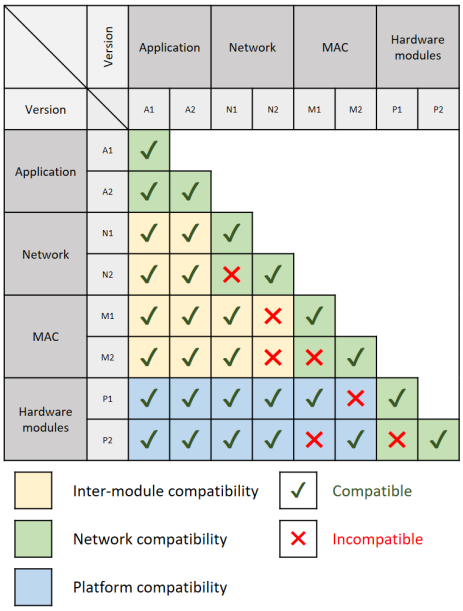

Fig. 3: An example matrix showing the compatibility between devices and network layers. For every combination, a test suite should perform interaction tests in order to verify the compatibility of the version combination.

methodology to also investigate if the update actually improves the network performance and stability. This is necessary because, contrary to typical software systems, a software update in constrained IoT networks reduces the battery lifetime and cannot be easily reverted. A qualitative analysis provides insights in the network operation and verifies the impact on both the node local and network wide Quality of Service (QoS) after the update. It can unveil issues that were not detected during the compatibility analysis using the compatibility matrix. For instance, it checks if the throughput and latency requirements are still fulfilled. This new information can also be used to extend the compatibility matrix.

There are a number of ways in which a qualitative analysis can be performed. A sandbox or testbed enables testing the software in a controlled environment on real hardware [6]. While this gives accurate information on a node local level, it is notoriously hard to verify network wide interactions since a sandbox is different from the actual environment. To overcome this, often a network simulator is used (e.g. CupCarbon, Cooja, OMNeT++, NS-3, QualNet, etc. [7]). This can provide a network wide view on the overall QoS, although the results are always an estimation based on a particular channel model.

A simulated virtual environment can be further enhanced using the digital twin concept, used primarily in the context of manufacturing and warehousing [8]. The digital twin mimics the behaviour of real physical objects, allowing to test new solutions and business processes in a non-invasive manner. In the context of over-the-air updates, a digital twin mimics a network of interacting IoT devices containing all node types and software combinations of the real life deployment. Moreover, the simulated environment is continuously updated with information retrieved from the actual network, improving the simulation model. If the digital twin network behaves as expected after the update, the network operator can initiate the

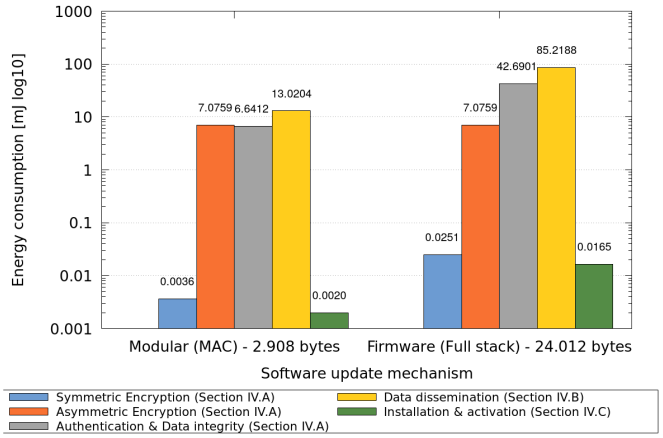

Fig. 4: Energy consumption of the different steps of the over-the-air update process for two update methods: a firmware update (full stack) and partial updates (MAC layer). All results are shown on a logarithmic scale.

actual software rollout as described in the next section.

V. Phase 2: Secure software rollout

After validating the correctness of an upcoming software update, still several steps need to be taken in order to complete the update. Minimally the deployment sequence includes: (i) securing and authenticating the data transfer, (ii) disseminating the software update module(s), and (iii) installing the software module(s) on all devices and coordinating a simultaneous activation. Note that each step introduces a non-negligible overhead in terms of device memory, network traffic and power consumption. This could drain the batteries, decreasing the operational lifetime of the constrained wireless devices. Therefore a trade-off should be made comparing the (possible) performance gains with the overhead of the update. To gain further insights regarding the overall energy cost of each step, the energy consumption has been measured on a typical IoT device (Zolertia Remote) for both a modular as well as a full firmware update, as shown in Figure 4 and detailed in the next subsections. The results were obtained by using the model for data dissemination and installation cost as defined in [2] and combined with extra measurements concerning the various security techniques. The aforementioned results were determined for a single hop topology and do not take into account possible packet loss and/or retransmissions. The energy consumption was calculated only for the battery-powered end devices. **Overall, a typical full firmware update requires about 135.026mJ, while a modular update only requires 26.743mJ.**

A. Software update security

While wireless updates can be used to fix security vulnerabilities, the update process can also open possible exploits and enable malicious control over the software stack. Several techniques have been proposed in order to secure over-the-air updates in Wireless Sensor Networks (WSN), or data dissemination in general. For example, the authors of [9] analyzed how a data transfer can occur while maintaining the four major security aspects: confidentiality, integrity, authentication and availability. The software updates are typically provided by the manufacturer or the local network operator.

Although security is clearly beneficial, the impact on the device functionality as well as network performance should not be underestimated. The remainder of this section will elaborate which security measures are feasible and will quantify the overhead for the constrained end devices. These end nodes typically have to verify the origin of the software update, and decrypt the packet contents.

In order to verify the data integrity and origin of traffic, a hash based message authentication code (HMAC) can be appended to each data packet. However, HMAC is commonly used with SHA-256, which requires 32 bytes per packet or 25% of the 127 bytes IEEE802.15.4 Maximum Transmission Unit (MTU). **Figure 4 (third column) shows the measured energy consumption for data integrity and authentication, requiring 6.641 (42.690) mJ for a modular update (full firmware update), which accounts for 24.83% (31.62%) of the total energy consumption.** The HMAC packet overhead is the main reason for the high energy cost of authentication and data integrity, while the computational overhead for calculating the HMAC only constitutes to ±0.01% for both methods. The energy cost can be drastically lowered by appending a single HMAC for the entire update file on the last packet. On the other hand, this disables the possibility to verify data integrity on a per packet basis.

Besides authentication and data integrity, confidentiality is an equally important security aspect. Two flavours of data encryption methods exist: symmetric key encryption (e.g. AES), and asymmetric encryption (e.g. RSA or ECC). In general, symmetric key encryption is faster then the asymmetric counter part. For instance, on the IEEE 802.15.4 CC2538 radio chip it takes 3.4ms to decrypt a full firmware upgrade of 24012 bytes using AES-256, while the same decryption takes 12606.3ms using RSA-1024. On the other hand, symmetric keys offer less security, since the shared key enables unauthorized network access when compromised. Also using multicast traffic in combination with asymmetric encryption is difficult to realise, since all data recipients should have the same decryption key requiring a key sharing protocol. Therefore it is not feasible to solely rely on either symmetric or asymmetric encryption to guarantee confidentiality.

In order to achieve a high level of security with a decreased energy cost and a lower computational overhead, it is advised to combine both methods. For instance, the Datagram Transport Layer Security (DTLS) standardized protocol [10] implements a combination of the two previously described approaches. During the initial device bootstrapping, server and clients exchange certificates containing their public key. Per software update a new symmetric session key is generated, which is subsequently asymmetrically encrypted and transmitted via a ‘Server Key Exchange‘. Any further over-the-air traffic, related to the current software update, is encrypted using this session key. This mechanism offers the advantages of (i) minimizing the consequences of a compromised symmetric encryption key; (ii) enabling multicast during over-the-air update; and (iii) offering a good trade-off between

performance and security. **Summarizing the above results, when using DTLS like approaches, encryption still requires a significant amount of energy: 7.080mJ for a modular update and 7.101mJ for a full firmware update. Compared to the total update cost this accounts to 26.47% and 5.29% respectively.**

B. Code dissemination

Before the installation of a software update, it must be possible to guarantee successful dissemination of the update to all devices in the network. Several techniques were proposed, as surveyed in [11]. They focus on minimal energy consumption by applying efficient broadcast schemes and try to avoid flooding the network. More recently, with the rise of low power wide area networks, operating in the duty cycle restricted unlicensed sub-GHz bands, novel techniques [12] have been proposed to overcome the duty-cycle restrictions imposed by regulatory bodies such as FCC and ETSI. Dissemination techniques that rely on unicast transmission schemes cannot be applied in large scale networks due to these restrictions. For networks with such limitations, coordinated multicast techniques [13] should be applied that can initiate a over-the-air update session on groups of devices [14].

In most cases, software updates exceed the MTU of packets, hence fragmentation is required. A software dissemination protocol must make sure that all devices receive the entire update file. This process does not tolerate any lost packets or bit errors, as this results in corrupted binary code. To overcome this problem efficiently (e.g. with minimal impact on energy and latency), special measures such as block (N)acks and caching on intermediate devices in case of a multi-hop topology are required. Especially when employing multicast dissemination, retransmissions should also be grouped and multicasted to reduce overhead.

Figure 4 shows the energy usage for the constrained wireless end devices during an over-the-air operation. A large portion of the overall energy usage can be accounted to the dissemination (yellow column), especially when considering that the HMAC message overhead (gray column) is calculated separately. Also notable are the differences between full firmware and modular updates. **Overall, dissemination costs 13.020 (85.219) mJ for a modular (full firmware) update. Relative to the total energy cost of the update, this constitutes to 48.68% (63.11%).**

C. Software module installation and activation

After disseminating the update, the software must be installed and subsequently activated on all IoT devices. The installation procedure is different for each of the update methods (i.e. firmware based, dynamic linking and prelinking) as discussed earlier in Software Module Compilation. The installation starts as soon as the server notifies that all devices have received and verified the complete update file. The result of this process is reported back to the server after which it can initiate the activation procedure.

Activating software on a group of networked devices should happen simultaneously, especially when concerning network functionality. A failed or delayed activation on one or more devices could introduce protocol inconsistencies, resulting in network connectivity issues. Even worse, if the connection between the server and (some) end devices is completely broken, it is even not possible to fix the issues remotely. This is also the reason why automatic rollback mechanisms should be incorporated, forcing devices to restore the previous software version, either when demanded by the server, or when the connection to the server has been lost.

Figure 4 demonstrates that the installation and activation overhead is negligible, as installing only requires copying the relevant sections to RAM or ROM. Note that when using a dynamic linker on the device, a small portion of CPU overhead is added. **Overall installing and activating an update requires 0.002 (0.017) mJ for a modular (full firmware) update, constituting only ±0.01% of the overall energy cost for both methods.**

VI. Future research directions

When IoT systems become more mature, their ecosystems grow, often attracting third party developers that want to add custom software. This is a natural evolution that helps extending software systems beyond its originally intended scope. This will put forward many challenges that need to be tackled properly before IoT networks are truly sustainable. (i) The trustworthiness of third party code needs to be verified. Verifying the origin of a software update is therefore important. (ii) The solutions which are offered by this (and other) article(s) do not take into account the existence of multiple owners or owner groups, which has a deep impact on the properties of secure software dissemination protocols. (iii) Code isolation techniques should be developed in order to prevent attacks from inserting malicious code. This will have an impact on run-time performance and memory requirements. (iv) Recent software-defined-radio (SDR) platforms also allow partial updates of Field-Programmable Gate Array (FPGA) functionality. By combining both micro controller and SDR reprogramming, the entire network stack, including the physical layer, becomes upgradable. (v) The recent trends towards software defined networking (SDN) approaches and virtualization, allows networks to inject new network rules into the application layer, thereby influencing lower layer protocol behaviour. These approaches could be extended by injecting not only rules, but even full software components or new network stacks at the application layer. (vi) An edge/fog-based architecture could be used to to more efficiently disseminate update data to the end devices, minimising the impact on the network.

VII. Conclusions

In the fast growing world of the Internet-of-Things, networks are deployed in increasingly diverse application domains, ranging from smart homes to Industry 4.0 factories. Most IoT devices are constrained in terms of energy, memory and processing power. In order to make IoT solutions truly sustainable, it is necessary to periodically update (parts of) the software post deployment. This article gave a comprehensive

overview of the principles, necessary to implement a secure and efficient over-the-air software update mechanism, resulting in a step-by-step approach as summarised in Figure 2.

Two distinct phases were identified: (i) the software module management phase, and (ii) the secure software rollout. The first phase is performed completely offline, in order to minimise the impact for the deployment network. Using the combination of a compatibility matrix and digital twin network it is possible to identify bugs, version incompatibilities or performance issues even before the update is executed. The second phase elaborated on the eventual rollout of the software modules to the devices, quantifying the energy overhead per step (symmetric/asymmetric encryption, authentication, data integrity, data dissemination, installation, and activation) as can be seen in Figure 4. The results show that beside the obvious dissemination cost, the other steps also introduce a significant overhead especially for modular updates (i.e. 51.31% vs. 36.89% for a full firmware update). The use of HMAC for data integrity and authentication, and the use of ECC for (a single) encryption occupy the main portion of this overhead.

To conclude, using this step-by-step approach, it is possible to improve the sustainability of IoT solutions and calculate the possible overhead upfront. The results obtained in the second phase allow a network operator to estimate the cost of either a modular or a full firmware update in terms of energy. This enables the operator to make a trade-off between this cost and the impact on the performance after the update. Moreover, the digital twin network, included in the first phase, can be used to evaluate the potential performance gains as well upfront.

ACKNOWLEDGMENT

This work was partially supported by the FWO SBO IDEAL-IoT project (S004017N), the FWO EOS MUSE-WINET project (G0F4918N), and the H2020 ORCA project (732174).

REFERENCES

[1] H. Guissouma *et al.*, "An empirical study on the current and future challenges of automotive software release and configuration management," in *Euromicro Conf. on Software Engineering and Advanced Applications*. IEEE, 2018, pp. 298–305.

[2] P. Ruckebusch *et al.*, "Modelling the energy consumption for over-the-air software updates in lpwan networks: Sigfox, lora and ieee 802.15. 4g," *Internet of Things*, vol. 3, pp. 104–119, 2018.

[3] J. C. Cano *et al.*, "Evolution of iot: An industry perspective," *IEEE Internet of Things Magazine*, vol. 1, no. 2, pp. 12–17, 2018.

[4] P. Ruckebusch, E. D. Poorter, C. Fortuna, and I. Moerman, "Gitar: Generic extension for internet-of-things architectures enabling dynamic updates of network and application modules," *Ad Hoc Networks*, vol. 36, pp. 127–151, 2016.

[5] P. Brada and J. Bauml, "Automated versioning in osgi: A mechanism for component software consistency guarantee," in *Euromicro Conf. on Software Engineering and Advanced Applications*, 08 2009, pp. 428–435.

[6] S. De *et al.*, "Test-enabled architecture for iot service creation and provisioning," in *The Future Internet Assembly*. Springer, 2013, pp. 233–245.

[7] M. Chernyshev *et al.*, "Internet of things: Research, simulators, and testbeds," *Internet of Things Journal*, pp. 1637–1647, 2017.

[8] W. Kritzinger, M. Karner, G. Traar, J. Henjes, and W. Sihn, "Digital twin in manufacturing: A categorical literature review and classification," *IFAC-PapersOnLine*, vol. 51, no. 11, pp. 1016–1022, 2018.

[9] F. Doroodgar, M. A. Razzaque, and I. F. Isnin, "Seluge++: A secure over-the-air programming scheme in wireless sensor networks," *Sensors*, vol. 14, no. 3, pp. 5004–5040, 2014.

[10] S. L. Keoh, S. S. Kumar, and H. Tschofenig, "Securing the internet of things: A standardization perspective," *IEEE Internet of Things Journal*, vol. 1, no. 3, pp. 265–275, 2014.

[11] X.-L. Zheng and M. Wan, "A survey on data dissemination in wireless sensor networks," *Journal of Computer Science and Technology*, vol. 29, no. 3, pp. 470–486, 2014.

[12] L. Cheng *et al.*, "Towards minimum-delay and energy-efficient flooding in low-duty-cycle wireless sensor networks," *Computer Networks*, vol. 134, pp. 66–77, 2018.

[13] B. Kim and K.-i. Hwang, "Cooperative downlink listening for low-power long-range wide-area network," *Sustainability*, vol. 9(4), p. 627, 2017.

[14] J. Toussaint, N. El Rachkidy, and A. Guitton, "Performance analysis of the on-the-air activation in lorawan," in *Information Technology, Electronics and Mobile Communication Conf.* IEEE, 2016, pp. 1–7.

Jan Bauwens (jan.bauwens2@ugent.be) received his B.Sc. and M.Sc. Engineering in Computer Science from Ghent University. Since 2015 he has been a Ph.D. student at Ghent University as part of the Internet Technology and Data Science Lab (IDLAB) research group. He has been participating in several European and national research projects. His research topic is concentrated around flexible MAC development in Internet-of-Things networks.

Peter Ruckebusch (peter.ruckebusch@ugent.be) received his M.Sc. in computer science from Hogeschool Ghent Faculty Engineering, Belgium. Since 2011 he has been a Ph.D. student at the University of Ghent, IMEC, IDLab, in the Department of Information Technology (INTEC). He has been collaborating in several national and European projects. His research topics are situated in the low end of IoT, mainly focusing on reconfigurability and reprogrammability aspects of protocol stacks for constrained devices in IoT networks.

Spilios Giannoulis (spilios.giannoulis@ugent.be) received his M.Sc in electrical and computer engineering (2001) and Ph.D. (2010) from the University of Patras. Since 2015 he has been a postdoctoral researcher at the University of Ghent, IMEC, IDLab. He is involved in several EU projects. His main research interests are mobile ad hoc networks, wireless sensor networks, especially flexible and adaptive MAC and routing protocols, QoS provisioning, and cross layer and power aware architecture design as well as dynamic spectrum access techniques.

Ingrid Moerman (ingrid.moeman@ugent.be) received her M.Sc. in electrical engineering (1987) and Ph.D. (1992) from the University of Ghent, where she became a part time professor in 2000. She is a member of IDLab-UGent-IMEC, where she coordinates research on mobile and wireless networking. Her research interests include IoT, LP-WAN, cooperative networks, cognitive radio networks and flexible architectures for radio/network control and management. She has long experience in coordinating national and EU research projects.

Eli De Poorter (eli.depoorter@ugent.be) received his M.Sc (2006) in computer science engineering and Ph.D. (2011) from the University of Ghent. He is now a professor at INTEC, University of Ghent. He is currently also coordinating several national and international projects. His main research interests include wireless network protocols, network architectures, wireless sensor and ad hoc networks, future Internet, self learning networks, and next generation network architectures.