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 IoT is evolving, 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 overhead, especially in terms of energy consumption, thereby impacting the operational lifetime of the battery constrained devices. Until now, a comprehensive overview describing the different update steps and quantifying the impact of each step is missing in the scientific literature, making it hard to assess the overall feasibility of an over-the-air update. To remedy this, our article analyzes which parts of an IoT operating system are most updated after device deployment, proposes a step-by-step approach to integrate software updates in IoT solutions, and quantifies the energy cost of each of the involved steps. 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 135.026 mJ, of which the main portions are data dissemination (63.11 percent) and encryption (5.29 percent). However, when modular updates are used instead, the energy cost (e.g., for a MAC update) is reduced to 26.743 mJ (48.69 percent for data dissemination and 26.47 percent for encryption).

Introduction

The Internet of Things (IoT) refers to the trend to include small, cheap, and/or energy-efficient wireless radios in everyday objects. IoT solutions are already digitizing an increasing amount of functionalities of modern-day society, impacting application areas such as healthcare, surveillance, agriculture, personal fitness, and home and industry automation. This trend will lead to a further increase in the number of devices per person and 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 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 operational IoT networks. 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 coming years, even with the possibility of monthly software updates [1].

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

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

- An analysis concerning the distribution of software development effort in different parts of widely used IoT operating systems
- A comprehensive overview of the key steps in an over-the-air update process
- A quantification of the energy overhead per deployment phase, showing the relative energy impact
- A discussion on the impact of updates on operational processes, such as the versioning approach used for software modules
- A list of future research directions that could enhance the potential of over-the-air updates

Analysing Update Requirements in IoT Operating Systems

It is important to recognize parts of IoT solutions that 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:

- Application software
- Network protocol stack software
- Operating system (OS) core software
- Platform hardware driver software

Contrary to traditional software systems, IoT applications are relatively simple (i.e., sense or actuate) and therefore small in size. Most logic resides in

The authors analyze which parts of an IoT operating system are most updated after device deployment. They propose a step-by-step approach to integrate software updates in IoT solutions, and quantify the energy cost of each of the involved steps. The results show that besides the obvious dissemination cost, other phases such as security also introduce significant overhead.

The authors are with Ghent University - IMEC.

Digital Object Identifier: 10.1109/MCOM.001.1900125

0163-6804/20/$25.00 © 2020 IEEE

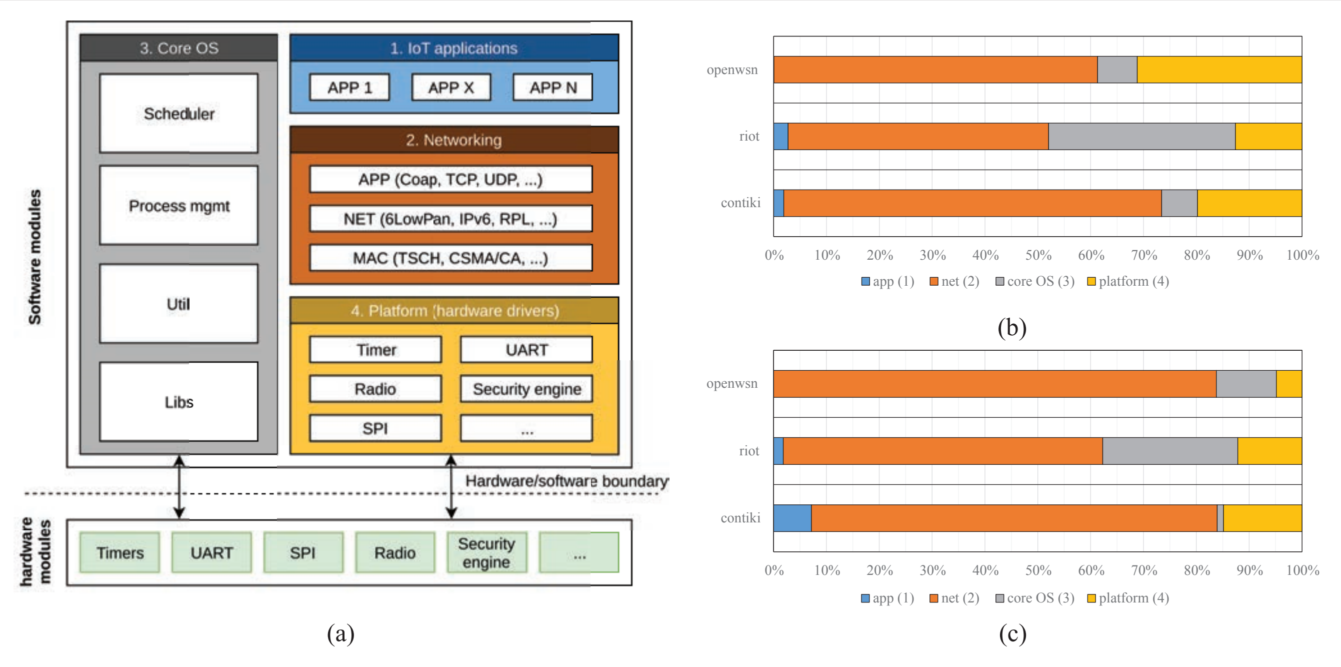


Figure 1. Git commit statistics and memory usage of three IoT operating systems: a) the typical software components IoT stack (and the hardware interface), divided in application code (1), networking functionality (2), OS core functionality (3), and platform specific code (4); b) relative memory size per component (percent); c) relative Git commit statistics of the different components, indicating the percentage of code lines changed between OS software releases (between V2018-01 and V2018.04 for RIoT, RB1.4 and RB1.8 for OpenWSN, and REL3.1 and the August 2018 master branch for Contiki).

the various network protocols, enabling end-to-end communication with IoT devices. 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 in Figs. 1b and 1c (in percent). *The network protocol stack comprises a significant portion of the firmware, occupying between 50 and 72 percent.* The complexity of the network stack is the main reason for the larger code size, which has a direct consequence on the development effort. Moreover, since standards are frequently updated, there is a continuous push to include the latest features in the code base. This, in contrast to the core OS and platform code, is illustrated in Fig. 1c, showing the Git commit statistics for two consecutive releases of three IoT operating systems. The statistics include all code lines changed in the software modules required to build a firmware on the Zolertia Remote. *Between 60 and 84 percent 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 in enabling application updates, Fig. 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 impossible to guarantee 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 at 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.

Software Update Process

This section provides the step-by-step process required to enable over-the-air software updates in a secure and reliable manner, visually represented in Fig. 2. The procedure is initiated by downloading an updated module from a software repository. First, during the "SW module management" phase, the code is verified offline. This phase includes:

- A compilation step, automatically adding linker metadata to the resulting binary module
- A compatibility analysis step, checking compatibility with the already deployed modules maintained in a binary module repository

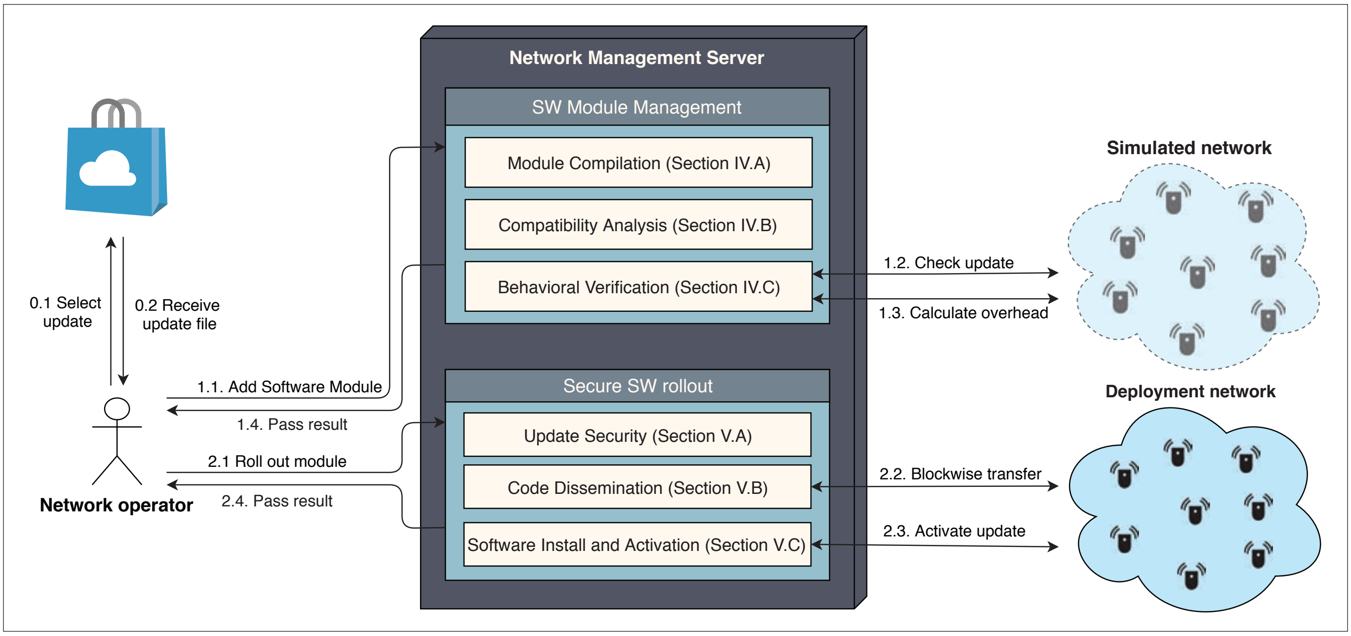


Figure 2. Workflow to perform over-the-air updates, split between the management of software modules and secure software rollout.

- A functional verification step, verifying the 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:

- A security step, encrypting and signing the binary module
- A dissemination step, transferring the binary module to the devices
- An activation step, installing and making the binary module operational

Each of these steps is discussed in more detail in the next sections.

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 have adverse effects on performance, 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, it is important to automatically assess the validity of the update up front, before actually performing the change and possibly wasting valuable resources. For this purpose, a network operator will first pass the software modules to the software module management services, which include three distinct steps, each explained in the remainder of this section: module compilation, compatibility analysis, and qualitative predeployment behavioral verification.

Software Module Compilation

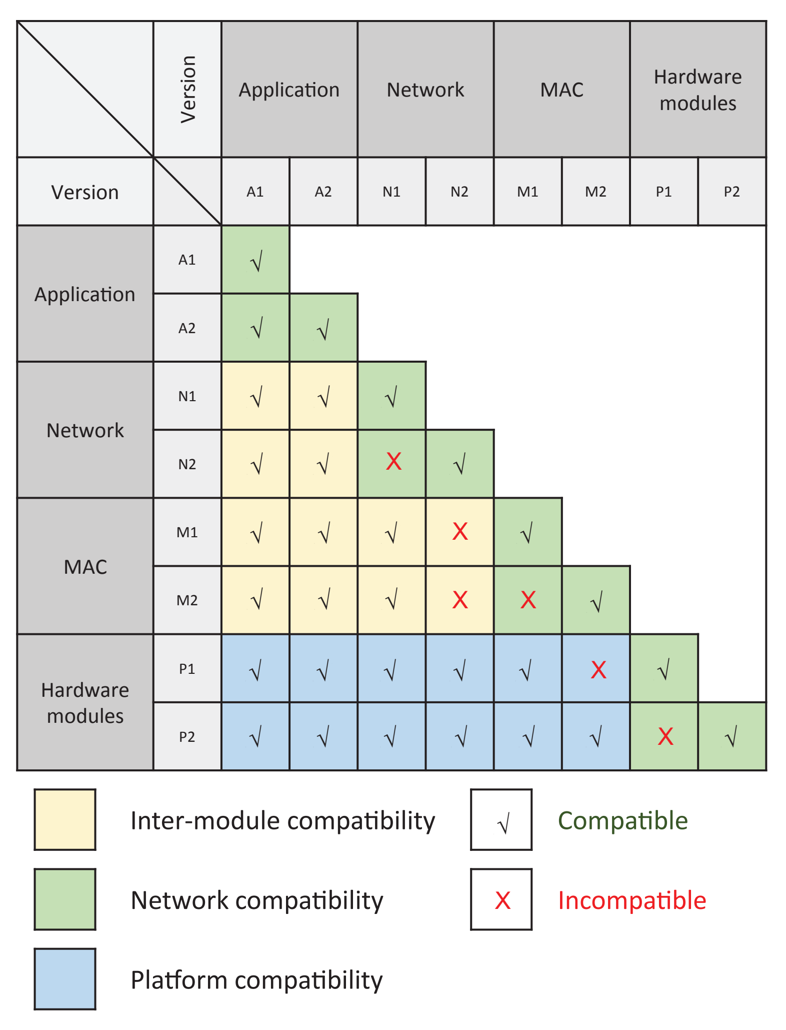
Module compilation ensures that the software is transformed from source code into an efficient binary format that can be distributed to a running system. Three different over-the-air software compilation methods are available for constrained devices, as discussed in [2]:

- Firmware-based approaches replace the entire image. All source code is compiled into a single image and installed on each device. Binary differential patching techniques (e.g., sending only the firmware differences) can be used in order to reduce the image size 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) sections 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 on each device.

The latter two approaches (i.e., 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 all of the firmware even after installation. This comes at the cost of increased jump table management complexity and extra memory usage.

The choice of the update method and binding model has a considerable impact on the possible



	Version	Application		Network		MAC		Hardware modules	
Version		A1	A2	N1	N2	M1	M2	P1	P2
Application	A1	√							
	A2	√	√						
Network	N1	√	√	√					
	N2	√	√	X	√				
MAC	M1	√	√	√	X	√			
	M2	√	√	√	X	X	√		
Hardware modules	P1	√	√	√	√	√	X	√	
	P2	√	√	√	√	√	√	X	√

🟨 Inter-module compatibility

🟩 Network compatibility

🟦 Platform compatibility

√ Compatible

X Incompatible

Figure 3. An example matrix showing the compatibility between devices and network layers.

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 replacing the entire code base but require the 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 updating/adding application-level code blocks (i.e., the blue part of Fig. 1a), while a loosely coupled binding model [4] can update all code blocks, for example, when including the network protocols.

Compatibility Analysis

Because software modules are often developed independent 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. If 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 (Fig. 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 coexist 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 partial update methods because network layers can be updated separately. For instance, an updated medium access control (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 physical layer, rendering the new MAC version incompatible. To counter this, the central management server keeps track of the software version(s) installed on each device and ensures compatibility. A possible approach utilizes a matrix for keeping track of compatibility (Fig. 3). This compatibility matrix can be updated by performing tests ranging from static code verification to simulations to analysis in a real-life deployment.

Pre-Deployment Behavioral Verification

While the previous verification step primarily focuses on compatibility of software versions, this section describes a 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 easily be 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. For instance, it checks if the throughput and latency requirements are still fulfilled.

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, NS-3 [7]), which provides a network-wide view of the overall QoS. However, 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, applied primarily in the context of manufacturing and warehousing [8]. The digital twin mimics the behavior of real physical objects, allowing the testing of 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.

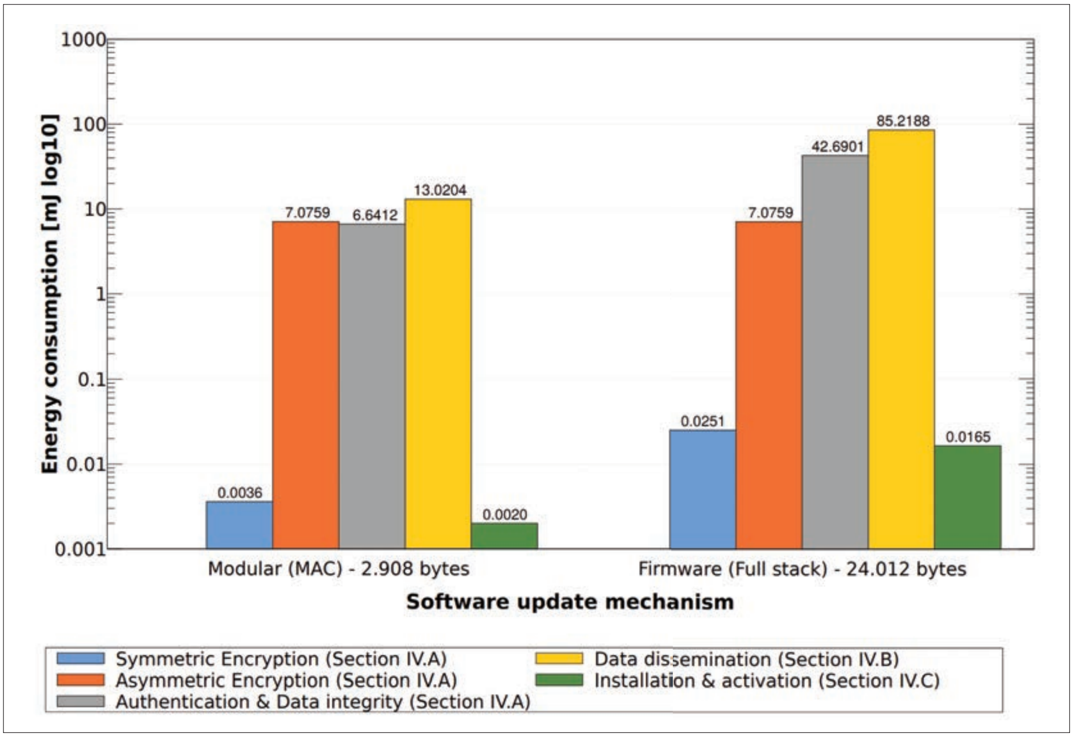


Figure 4. Energy consumption of the over-the-air update process for firmware updates (full stack) and partial updates (MAC layer), shown on a logarithmic scale.

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. For instance, it checks if the throughput and latency requirements are still fulfilled.

Moreover, the simulated environment is continuously updated with information from the actual network, improving the simulation model. If the digital twin network behaves as expected after the update, the actual software rollout can be initiated.

Phase 2: Secure Software Rollout

After validating an upcoming software update, the deployment sequence can commence, including:

- Securing and authenticating the data transfer
- Disseminating the software update module(s)
- Installing the software module(s) 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 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 for the Zolertia Remote device for both a modular as well as a full firmware update, as shown in Fig. 4. 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. *Overall, a full firmware update requires about 135.026 mJ, while a modular update only requires 26.743 mJ.*

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, 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 elaborates which security measures are feasible and will quantify the overhead for the constrained end devices. These end nodes typically verify the origin of the 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 B/packet or 25 percent of the 127-byte IEEE 802.15.4 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 percent (31.62 percent) 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 0.01 percent 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.

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 extend software systems beyond their originally intended scope. This will put forward many challenges that need to be tackled to achieve sustainable IoT networks.

Besides authentication and data integrity, confidentiality is an equally important security aspect. Two flavors 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 than its asymmetric counterpart. For instance, on the IEEE 802.15.4 CC2538 radio chip, it takes 3.4 ms to decrypt a full firmware upgrade of 24,012 bytes using AES-256, while the same decryption takes 12,606.3 ms 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 realize, requiring a key sharing protocol. Therefore, it is not advisable to solely rely on either symmetric or asymmetric encryption to guarantee confidentiality.

In order to achieve a high level of security with decreased energy cost and lower computational overhead, a combination of both methods can be used (e.g., the Datagram Transport Layer Security [DTLS] standardized protocol [10]). 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 minimizing the consequences of a compromised symmetric encryption key, enabling multicast during over-the-air update, and offering a good trade-off between performance and security. *When using DTLS-like approaches, encryption still requires a significant amount of energy: 7.080 mJ for a modular update and 7.101 mJ for a full firmware update. Compared to the total update cost, this accounts to 26.47 and 5.29 percent, respectively.*

Code Dissemination

Several techniques exist to disseminate an update to wireless devices, 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, and novel techniques [12] have been proposed to overcome the duty-cycle restrictions imposed by regulatory bodies such as the Federal Communications Commission (FCC) and European Telecommunications Standards Institute (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 an 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 with minimal impact on energy and latency, special measures such as block (N)acks and caching on intermediate devices are required. Especially when employing multicast dissemination, retransmissions should also be grouped and multicast 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 for by the dissemination, especially when considering that the HMAC message overhead 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 48.68 percent (63.11 percent).*

Software Module Installation and Activation

After update dissemination, the software must be installed and activated on all devices. The installation procedure is different for each of the update methods (i.e., firmware-based, dynamic linking, and prelinking), as discussed earlier. The installation starts as soon as the server reports that all devices have received and verified the complete update file. The result is reported back to the server after which the activation procedure can be initiated.

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 to the update server is broken, it is impossible to fix the issues remotely, making automatic rollback mechanisms a necessity. The devices need 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 percent of the overall energy cost for both methods.*

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 extend software systems beyond their originally intended scope. This will put forward many challenges that need to be tackled to achieve sustainable IoT networks:

- The trustworthiness and origin of third-party code needs to be verified.
- The solutions offered by research 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.
- Code isolation techniques should be developed in order to prevent attacks from inserting malicious code.
- Recent software-defined radio (SDR) platforms also allow partial updates of field-programmable gate array (FPGA) functionality and thereby the entire network stack, including the physical layer, becomes upgradable.

- The recent trends toward software defined networking (SDN) approaches and virtualization allow networks to inject new network rules into the application layer, thereby influencing lower layer protocol behavior. These approaches could be extended by injecting not only rules, but even full software components or new network stacks at the application layer.
- Edge/fog-based architecture could be used to more efficiently disseminate update data to the end devices.

Conclusions

In the fast growing world of the Internet of Things, networks are deployed in increasingly diverse application domains. In order to make IoT solutions truly sustainable, it is necessary to periodically update (parts of) the software post-deployment. This article gives 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 summarized in Fig. 2.

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

To conclude, it is possible to improve the sustainability of IoT solutions and calculate the possible overhead up front. 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. Thus, a trade-off can be made between this cost and the performance increase.

It is possible to improve the sustainability of IoT solutions and calculate the possible overhead up front. 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. Thus, a trade-off can be made between this cost and the performance increase.

Acknowledgments

This work was supported by the FWO SBO IDEAL-IoT project (S004017N), the FWO EOS MUSEWINET 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," *Euromicro Conf. 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 J.*, vol. 3, 2018, pp. 104–19.

[3] J. C. Cano *et al.*, "Evolution of IoT: An Industry Perspective," *IEEE Internet of Things Mag.*, vol. 1, no. 2, 2018, pp. 12–17.

[4] P. Ruckebusch *et al.*, "Gitar: GekneriC Extension for Internet-of-Things Architectures Enabling Dynamic Updates of Network and Application Modules," *Ad Hoc Networks*, vol. 36, 2016, pp. 127–51.

[5] P. Brada and J. Bauml, "Automated Versioning in OSGI: A Mechanism for Component Software Consistency Guarantee," *Proc. Euromicro Conf. Software Engineering and Advanced Applications*, Aug. 2009, pp. 428–35.

[6] S. De *et al.*, "Test-Enabled Architecture for IoT Service Creation and Provisioning," *The Future Internet Assembly*, Springer, 2013, pp. 233–45.

[7] M. Chernyshev *et al.*, "Internet of Things: Research, Simulators, and Testbeds," *Internet of Things J.*, 2017, pp. 1637–47.

[8] W. Kritzinger *et al.*, "Digital Twin in Manufacturing: A Categorical Literature Review and Classication," *IFAC-PapersOnLine*, vol. 51, no. 11, 2018, pp. 1016–22.

[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, 2014, pp. 5004–40.

[10] S. L. Keoh, S. S. Kumar, and H. Tschofenig, "Securing the Internet of Things: A Standardization Perspective," *IEEE Internet of Things J.*, vol. 1, no. 3, 2014, pp. 265–75.

[11] X.-L. Zheng and M. Wan, "A Survey on Data Dissemination in Wireless Sensor Networks," *J. Computer Science and Technology*, vol. 29, no. 3, 2014, pp. 470–86.

[12] L. Cheng *et al.*, "Towards Minimum-Delay and Energy-Efficient Coding in Low-Duty-Cycle Wireless Sensor Networks," *Computer Networks*, vol. 134, 2018, pp. 66–77.

[13] B. Kim and K.-i. Hwang, "Cooperative Downlink Listening for Low-Power Long-Range Wide-Area Network," *Sustainability*, vol. 9, no. 4, 2017, p. 627.

[14] J. Toussaint, N. El Rachkidy, and A. Guitton, "Performance Analysis of the On-the-Air Activation in LoRaWAN," *Info. Tech., Electronics and Mobile Commun. Conf*. IEEE, 2016, pp. 1–7.

Biographies

Jan Bauwens (jan.bauwens2@ugent.be) received his B.Sc. and M.Sc. in engineering and 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 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. 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.

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.

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 IDLab, Ghent University-IMEC. 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.