SPLIT: Smart Protocol Loading for the IoT

Torsten Zimmermann[†], Jens Hiller[†], Jens Helge Reelfs, Pascal Hein, Klaus Wehrle
Chair of Communication and Distributed Systems, RWTH Aachen University
{zimmermann, hiller, reelfs, hein, wehrle}@comsys.rwth-aachen.de
[†]**Equal Contribution**

Abstract

The Internet of Things (IoT) permeates our everyday life, e.g., in the area of health monitoring, wearables, industry, and home automation. It comprises devices that provide only limited resources, operate in challenging network conditions, and are often battery-powered. To embed these devices into the Internet, they are envisioned to operate standard protocols. Yet, these protocols occupy the majority of limited program memory resources. Thus, devices can neither add application logic nor apply security updates or adopt optimizations for efficiency. This problem will further exacerbate in the future as the further ongoing permeation of smart devices in our environment demands for more and more functionality.

To overcome limited functionality due to resource constraints, we show that not all functionality is required in parallel, and thus can be SPLIT in a feasible manner. This enables on-demand loading of functionality outsourced as (multiple) modules to the significantly lesser constrained flash storage of devices. We exemplify efficient modularization of DTLS and show that SPLIT enables operation of large protocol stacks while it incurs reasonable, tunable performance trade-offs. Our use case specific results show an initial runtime overhead of 23.34 % and 4.9 % for subsequent protocol executions.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design; C.2.2 [**Computer-Communication Networks**]: Network Protocols

General Terms

Design, Security

Keywords

Internet of Things, Networking, Protocols, Modularization, On-demand Loading, Sustainability

International Conference on Embedded Wireless Systems and Networks (EWSN) 2018
14–16 February, Madrid, Spain

ISBN: 978-0-9949886-2-1

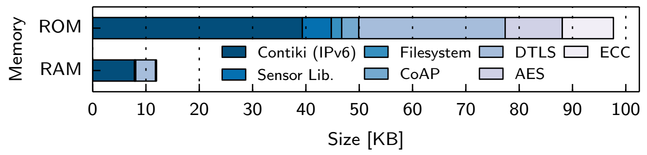


Figure 1. Memory requirements for a typical IoT stack.

1 Introduction

The ongoing permeation of smart devices in our environment, e.g., health monitoring, wearables, industry or home automation [1], provides the basis for the Internet of Things (IoT). As more and more users experience the benefits in these areas, the requirements regarding functionality provided by IoT devices also increase. Further demands for functionality emerge from the use of standard communication protocols to connect to the existing Internet infrastructure.

However, IoT devices are challenged by resource constraints especially facing limited processing power and tough memory boundaries, sparse energy provided by batteries, and lossy low-power wireless communication environments [3, 10]. These constraints lead to new, trimmed stacks with adapted protocols, e.g., 6LoWPAN [14]. Still, these protocols occupy the majority of memory resources, limiting capabilities for actual applications and further protocols.

Contrarily, we identify that functionality of many applications and protocols is separable into different phases, e.g., reading and processing a sensor value or connection setup and exchange of data. This especially holds for IoT security protocols such as DTLS [16], HIP DEX [15], and Minimal IKE [11]. Although intended for the IoT, they still occupy a non-negligible amount of memory (cf. Fig. 1 and analysis in Sec. 2). In addition, their optimizations trade off efficiency for higher memory usage [8, 10]. Consequently, memory for actual applications is scarce and thus significantly limits functionality, which is a crucial factor for envisioned IoT scenarios (Sec. 2). Simply increasing available device resources raises production costs and is thus undesirable, with regard to the tremendous amount of devices [1] and sustainability. However, the complete functionality, e.g., connection setup and exchange of data, is typically not required in parallel, which leaves potential for more efficient memory handling. Instead of keeping instructions for all functionality in program memory at all time, we propose to outsource functionality to the large flash storage and only load it on demand. Tapping this resource, we enable IoT devices to offer substantially more functionality. Existing approaches (Sec. 3) that exchange functionality at runtime mainly focus on reconfigurability and updatability after deployment [5, 17, 19]. Complementing these approaches, we analyze effect, feasibility, and applicability of on-demand loading for IoT stacks to efficiently use memory. Specifically, our contributions are as follows:

- We present **S**mart **P**rotocol **L**oading for the **IoT** (SPLIT), a mechanism for on-demand loading of functionality outsourced as (multiple) modules to the less constrained flash storage of devices. Outsourced functionality only employs resources when needed for execution (Sec. 4).
- We implement and evaluate SPLIT and highlight its applicability by employing the de-facto standard IoT security protocol DTLS (Sec. 5).

2 Memory Constraints Limit Functionality

We first analyze required functionality in typical IoT environments to highlight today's problem of limited functionality on memory-constrained devices: Typical tasks in the IoT involve communication with multiple entities locally or via the Internet. Humans may retrieve sensor readings or trigger actions [20]. Additionally, data is obtained from Internet services to make informed decisions. Finally, gathered data can be transmitted to the cloud for further processing [7]. Thereby, security and privacy of sensitive data, especially when sent over the Internet, must be maintained [1,7,10]. To realize this IoT vision, devices must cover a wide range of functionality from secure communication protocols to processing logic for sensor data, and interaction with the environment.

Still, they commonly must have low production costs, need to be compact, and are often battery-powered to be mobile. Thus, they are realized using specialized hardware at the price of restricted performance and capabilities in regard to processing and memory. Furthermore, to save power, they use low-power, low-rate wireless interfaces. We thus call these devices *constrained* [3]. Finally, they typically employ persistent storage, with a capacity of up to several MB. Exemplary devices are the Zolertia Z1 or the Wismote platform.

We illustrate the impact of the required functionality by an example and show the resulting memory footprint in Fig. 1. As target, we select the Wismote and rely on the Contiki OS [6]. To provide Internet connectivity IP is required, provided in form of IPv6. Further, to gather environmental data, libraries to operate and process sensor values are used. In addition, a filesystem stores data, e.g., in monitoring scenarios. To allow to act both as server and client, an application layer protocol (CoAP) is added. Depending on the use case and underlying network, this communication needs to be secured. Thus, we utilize DTLS along with its cryptographic functionality for AES and Elliptic Curve Cryptograhpy (ECC). In total, this results in 97.68 KB of ROM and 11.97 KB of RAM (cf. Fig. 1). Though this codebase already provides a basic set of functionality, adding features is challenging depending on the platform [3]. In the worst case, devices can neither add application logic nor apply security updates or optimizations, e.g., compression [8] or Denial of Service (DoS) protection.

Hitherto, increasing memory demands frequently resulted in the replacement of existing devices with less constrained devices. Given the sheer number of envisioned devices [1], this introduces unnecessary costs and following this procedure only forces to steadily exchange deployed devices, wasting resources and prohibiting sustainable deployments.

3 Related Work

Based on the mentioned problems (cf. Sec. 2), we focus on related work, that adapts or realizes protocol stacks for the IoT, or targets updates and reconfiguration after deployment.

Adapting Protocol Stacks: A key enabler for the IoT is the use of standardized protocols to allow a global interconnection over the Internet. Due to resource constraints [3], adaptations were proposed by academia and standardization organizations. To achieve IP end-to-end connectivity, 6LoWPAN [14] defines an adaptation layer to realize IPv6 over low-power wireless links. Typical deployments utilize leaner, less feature rich transport protocols, e.g., UDP, as TCP is rather heavyweight, e.g., due to accounting and retransmissions [1]. To further connect constrained and traditional networks, protocols like CoAP [18] simplify the mapping to HTTP. Though not originally intended for the IoT, DTLS [16] emerged as a lightweight security protocol [10, 13]. Optimizations allow protocols like DTLS, HIP DEX [15] and minimal IKE [11] to efficiently operate, though cryptographic operations challenge limited processing powers [10]. Other efforts delegate resource heavy operations to more powerful devices [9].

Still, further efforts are required to support a multitude of protocols, upcoming optimizations, and updates. Yet, not all functionality is needed in parallel all the time, an untapped potential we target with SPLIT to realize a sustainable IoT.

Adaptivity, Updates and Reconfiguration: A major challenge in the IoT is to enable adaptivity *after* deployment. This includes reconfiguration or complete exchanges of functionality, which was not provided in parallel due to constraints. Thus, approaches that utilize script interpreters, virtual machines [12, 22], or image based updates [4, 21] emerged to realize this flexibility. An advantage of the first approaches is their run-time interpretation of scripts or intermediate code. However, they exhibit increased memory requirements and execution times than native code [5]. The latter targets to update the image in situ, introducing time and communication overhead for complete binaries [17]. Thus, state-of-the-art distributes only deltas between versions and optimizes compilation to produce minimal differences [4,21]. Still, this targets long term changes, e.g., bug-fixing, as it involves patching and rebooting, and not changing to new functionality due to a new, maybe transient, environmental situation.

In contrast, run-time dynamic linking approaches divide the firmware into a static and dynamic part. This is similar to approaches in less-constrained computing environments with feature-rich OSs, that enable loading of Dynamic Libraries to add functionality *after* the program start or add Kernel modules without rebooting. A method to apply these features to the IoT is proposed in [5] for Contiki [6]. Building upon this, REMOWARE [19] and GITAR [17] optimize the module handling, in terms of memory or easier module exchange.

Despite the ability to dynamically update and reconfigure code, these approaches focus on support for updates rather than reducing the overall memory requirements. In the next section, we discuss how these systems influence our approach.

4 On-demand Loading of Functionality

As illustrated in Sec. 2, realizing an IoT stack and adding necessary application code can easily exceed the available memory of many constrained IoT devices, limiting the overall functionality that can be realized. We thus propose Smart Protocol Loading for the IoT (SPLIT), to enable *on-demand* loading of functionality. By that, we target to enable resource constrained devices to use a variety of functionality, without the need to consider ROM limitations. To achieve this, we propose to realize *base* functionality within the ROM itself, but outsource further functionality split into (multiple) modules to the less constrained flash storage of devices. In the following, we motivate the applicability of this approach.

Many use-cases, e.g., industry or home automation require IoT devices to communicate sensitive data over the Internet. Such communication needs to be secured, preferably utilizing standardized protocols like DTLS. However, *after* the handshake of a security protocol, which contributes the major part of the memory requirements, this functionality is not required for following data transmissions. Similarly, protocols above the transport layer[1] are not required for sensor readout or application processing. The key observation is that we can split functionality into smaller *modules* that do not need to be present in memory in parallel, due to the general workflow.

At the same time, IoT devices possess a comparably large flash storage, i.e., MB vs. KB, which is typically only marginally occupied by sensed data or device specific configuration. This provides a natural location to store currently not required modules, enabling us to increase the usable code base, i.e., functionality, only limited by the size of the flash storage, which is at least an order of magnitude larger.

We discuss our design (cf. Fig. 2) along an exemplary use case. An application on a device needs to establish a secure communication with an Internet-based service. The respective protocol is not included in the device's ROM, but present in form of modules (m_1-m_3) in the flash storage. We assume these to represent the *handshake*, the *secure transfer*, and the *teardown* of the protocol. When an application now wants to utilize the provided functionality, it triggers the *Loader*.

To simplify this process for developers, we propose to employ approaches as in [19]. Accordingly, a developer defines a task, e.g., *initialize communication*, that is interpreted by the Loader, e.g., provided to the *application* via an API, which in turn executes the required modules in order. Alternatively, the modules could each link to another, such that a successful return triggers to load and execute the following module.

The Loader copies the required code of m_1 from the device's flash storage to RAM. To always have sufficient memory available for this copy operation, SPLIT statically reserves memory. Specific values are discussed in Sec. 5. Before execution of the module, the Loader has to check if all symbols are resolved. These include symbols that link to basic firmware (global) functionality, e.g., memory allocation or file access, and to intra-module (local) functionality, e.g., multiple functions in one module that call each other [5, 17, 19]. After this, the module is ready and the entry point is called.

[1]UDP, 6LoWPAN, and core functionality are key requirements for the envisioned IoT and, thus, we decided to not split below the transport layer.

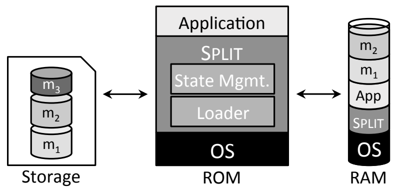


Figure 2. Functionality with a high memory footprint is split into modules and stored on the external storage. When needed, these are loaded, prepared, and executed.

During execution, m_1 needs to allocate memory for a data structure that is used by multiple modules, e.g., to store peer information. Thus, we add a *State Management* component that handles memory allocations for state required across modules. Other memory management tasks, e.g., for transient operations inside the modules, are still handled by the OS. After *handshake* completion, the Loader continues the protocol by loading and executing m_2. Depending on the available memory, m_1 can either stay active or has to be *unloaded* to free up space. If memory permits and m_1 is frequently used, the former reduces the overall loading time. After successful execution, the Loader loads m_3. In case of an error, it depends on the protocol or application if re-running the current module is sufficient or a rollback to a previous module is required.

With this approach, we reduce the overall ROM utilization and simultaneously increase the available functionality on the device, in principle only limited by the available flash storage.

4.1 Splitting Protocols into Modules

In the following, we show the modularization of DTLS as a representative security protocol. Especially security protocols require manifold functionality ranging from *large state machines* to handle various packet types during connection setup over *symmetric* to *public key cryptography* [10, 11, 13, 15, 16]. Furthermore, mechanisms that tailor these protocols to efficiently operate despite limited processing power and lossy, low-power wireless networks in the IoT, trade computation speedups against increased memory requirements [8, 10].

Fig. 3 shows the DTLS handshake and subsequent application data exchange. The handshake consists of several packets divided into *Flights*. Flights 1-2 implement a return-routability test to detect spoofed IP addresses. The *Hello* messages in Flights 3-4 negotiate DTLS parameters, e.g., the cipher. Furthermore, Flights 4-5 establish cryptographic keys (*KeyExchange*s) and authenticate the server to the client (optionally also vice versa) with *Certificate*s. *ChangeCipherSpec* and *Finished* in Flights 5-6 validate and finish the handshake.

This deterministic handshake procedure allows for a natural modularization of DTLS, e.g., along the flights or messages. However, flight-based modules are too large for our target devices. Thus, in a simple approach, we would split DTLS functionality into two modules for each message: One for processing and another for creating and sending of the message. Thus, we achieve efficient memory usage as modules only occupy memory when needed for the processing of a respective packet. Furthermore, we reduce incurred overheads as a handshake uses each module only once (two times for *ClientHello* modules). The same approach can be applied to application layer protocols, e.g., CoAP [18] which pro-

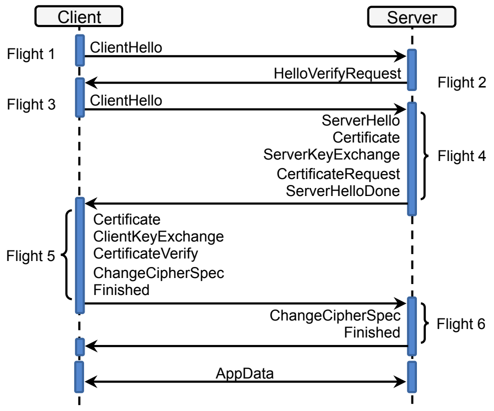


Figure 3. We separate DTLS messages into modules (individual messages). The granularity is tunable and protocol determinism allows for mitigating latency overheads.

cesses *GET*, *PUT*, *POST*, and *DELETE* requests. As SPLIT does not define the granularity of modules a priori, also alternative modularizations are possible. Moreover, even if CoAP fits into a single module, offloading it can still be beneficial as it frees resources for other resource-intensive protocols.

5 Implementation and Evaluation of SPLIT

Next, we describe our prototypical implementation of SPLIT's architecture and analyze its applicability on constrained devices. Moreover, we illustrate how we prepared the protocols for SPLIT. Subsequently, we evaluate the runtime overhead in comparison to a default DTLS implementation.

5.1 Implementation & Protocol Splitting

SPLIT Prototype: Our prototypical implementation of SPLIT is based on Contiki 2.7 [6]. We adapt and extend the Default Loader [5] with respect to our design in Sec. 4. Following our scenario, we choose Contiki with integrated IPv6 support over IEEE 802.15.4 links, i.e., 6LoWPAN, as our base OS. As modules are stored on the flash storage, we also include a file system. Modules use the Executable and Linkable Format (ELF), also used by the Default Loader. As target platform, we select the MSP430X-based Wismote which provides 16 KB of RAM and a minimum of 128 KB ROM (up to 256 KB). Although we target to improve the protocol handling of constrained devices that may expose less than the aforementioned ROM sizes [3], the remaining *headroom* alleviates the implementation, debugging and evaluation process.

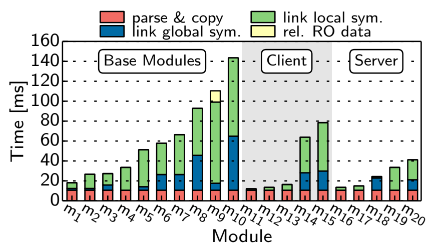
To trigger the *Loader* to execute a protocol or application, the developer calls a defined entry point, e.g., instructs the Loader to start a *handshake* or *measurement*. As a first step, the Loader locates the respective initial module on the file system, parses respective header information, e.g., offsets for symbols or string tables, and copies the binary code to the preallocated memory. Using the stored information, the Loader checks the *relocation* table. This contains symbols that cannot be resolved at compile time, either because they are not provided by the module itself (base firmware functionality), or their location is not known before the actual copy process. For base firmware functionality (global symbols), the Loader retrieves the name, e.g., `printf`, looks up the address in the symbol table of the firmware and writes this address to the respective location. Depending on the use-case and deployment, the modules can be pre-linked to the core firmware [5, 17], which allows to remove the necessary code for the global symbol linking as well as the symbol table itself, thus saving space. On the downside, this limits flexibility, as the modules cannot be used across platforms with different base firmware. In case of functionality provided by the module itself, the Loader calculates the final addresses after copying the code to the memory, using the starting address and offsets contained in the header. The standard loader employs linear search on the symbol's name to find this information in the ELF file. To obtain the data with a single flash operation, we instead read it from an ordered list distributed with the module. This process is repeated for read-only data and initialized variables.

Table 1. Size in KB when compiled for the Wismote. *Values obtained by measurements during evaluation.

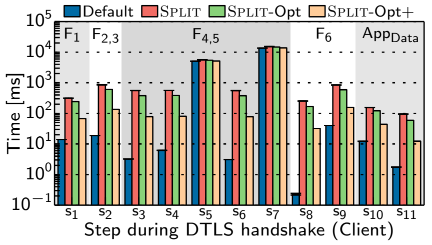
SPLIT	ROM	RAM
Contiki with IPv6	39.34	7.85
+ Loader + State Mgmt.	+ 2.57	+ 0.82 + 4.25*
+ SHA256 + μECC + AES	+ 3.35 + 9.59 + 10.72	+ 0.18
+ Symbol Table	+ 9.97	
Σ	**75.54**	**13.1**

Preparing Protocol Modules: With the SPLIT-architecture in place, the next step is to provide loadable modules. Following Sec. 4.1, we split DTLS along its handshake and secure exchange phases. We choose tinyDTLS 0.8.2 and replaced its ECC implementation with μECC (micro-ECC) as it exhibits faster processing at a minor size overhead. In this process, two requirements have to be tackled: *i)* We need to minimize the module interdependencies, i.e., extract self-contained modules if possible. Moreover, *ii)* we need to stay within platform dependent memory boundaries, i.e., use the available RAM efficiently, such that other operations are still possible. Given these requirements, we obtain 20 modules for tinyDTLS. Thereof client and server each have 5 individual modules, e.g., handling a client or server key exchange, and 10 common modules, e.g., initialization, retransmit handling or handshake finishing. Moreover, we decouple stand-alone functionality, i.e., SHA-256 hashing, μECC as well as AES, and place it in the ROM. We argue that this can also be utilized when disseminating modules, e.g., checking signatures, and thus save loading steps. In addition, we decouple per-connection state to be stored by the *State Management*.

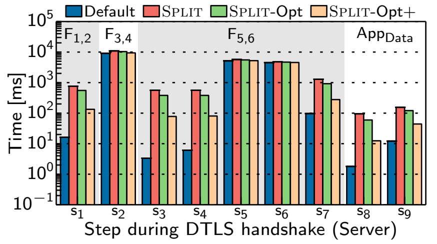
SPLIT Memory Breakdown: We summarize all sizes of the *static* base firmware in Table 1. The base firmware (Contiki + IPv6 support) occupies 39.34 KB ROM and 7.85 KB RAM. Adding the *Loader* (+ file system support) and the *State Management* component adds 2.57 KB to the needed ROM and 0.82 KB of RAM. To provide space for the modules, we reserve 1.32 KB of RAM to fit the largest module and further reserve 2.93 KB for protocol state (4.25 KB in total). The amount of reserved memory is empirically chosen and could be precomputed, e.g., using static code analysis and use-case knowledge, e.g., number of peers. Additionally, we move stand-alone functionality used by DTLS to the ROM, resulting in 3.35 KB for the SHA-256 hash implementation and 9.59 KB for μECC. Moreover, the AES implementation occupies 10.72 KB of ROM and 182 B of RAM. Finally, we



(a) Per module loading time for tinyDTLS.



(b) Default vs. SPLIT Client (default counterpart). Annotations correspond to DTLS flights.



(c) Default vs. SPLIT Server (default counterpart). Annotations correspond to DTLS flights.

Figure 4. Overhead introduced by SPLIT for individual modules and during operation.

include the symbol table of the complete firmware, which adds 9.97 KB to the needed ROM. In total, this SPLIT-enabled firmware occupies 75.54 KB of ROM and 13.1 KB of RAM. The tinyDTLS modules occupy 37.82 KB of flash memory, including *all* necessary ELF header information, e.g., string tables or relocation information. For actual code, 13.28 KB are necessary. This firmware fits the definition of constrained devices (specifically Class 1 devices) [3], but enables increased functionality that is in principle only limited by the storage.

5.2 Performance

SPLIT Base Time Overhead: Next, we evaluate SPLIT's processing time overhead based on the extracted tinyDTLS modules. To evaluate our prototype in a reproducible manner, we utilize the Contiki Cooja simulator. We program a client based on our SPLIT-firmware, populate the modules on the storage of the simulated device, subsequently load the modules, and measure the time for several steps during the loading process. Fig. 4(a) depicts these times for our 20 tinyDTLS modules. For better visibility, we sort them by their runtime and not in their order within the handshake process.

The first step consists of copying the relevant data from flash into RAM and parsing ELF headers, that contain information about symbols that have to be linked or read-only data that has to be copied. If not stated otherwise, the numbers provided in the following are the average and the standard deviation. This first step takes 10.64 ms $\pm$ 0.003 ms for the ELF files that have a size between 0.65 KB to 3.84 KB. Subsequently, the Loader links symbols, i.e., globally against the firmware and locally inside the module, which takes 0.8 ms $\pm$ 2.15 ms per global and 7.52 ms $\pm$ 6.57 ms per local symbol. The current prototype performs a rather naive approach, retrieving each global symbol individually from the table. Thus, this step increases with the amount of symbols. Optimizations include pre-linking global symbols, saving addresses in the ELF *after* the first load, or caching for repetitive lookups.

SPLIT Overhead during Execution: To analyze the impact of SPLIT during operation, we compare the *processing* time of a SPLIT-enabled client and server to execute a DTLS handshake with an *unaltered* counterpart over a single IEEE 802.15.4 hop against a *Default* client and server. The handshake is based on ECC utilizing the NIST P-256 curve [2], SHA-256 hashes, and mutual authentication.

We divide the handshake into steps and measure the individual processing times for the *Default* and SPLIT client. In this configuration, and with regard to the available RAM (cf. Table 1) and present modules, we are only able to load one module at a time. Thus, the following evaluation can be seen as a *worst-case* scenario. Based on the measurements, we further derive processing times for a SPLIT-Opt client that uses pre-linked modules (cf. Sec. 5.1). We observed retransmissions in the *Default* and SPLIT case, as DTLS does not consider the limited processing abilities of peers. However, approaches to drastically mitigate this effect exist [10]. We thus exclude retransmission influences in our further analysis. The results are summarized in Fig. 4(b). In the *Default* case, the whole processing time, including application data transmission (s_{11}), takes 18.75 s. When performing the handshake with a SPLIT-enabled client, the overall processing time increases by 33.74 % to 25.08 s. As summarized in Fig. 4(c), a SPLIT-enabled server that executes a handshake with an unaltered client takes 24.82 s, while the *Default* server requires 18.88 s. In this case, SPLIT adds 31.45 % of overhead compared to the *Default* implementation. The relative high overall processing is due to expensive ECC calculations and mutual authentication. Such a setup is required to ensure end-to-end security without dependence on a trusted delegation point [9]. Thus, we follow recommended standard settings [2].

Finally, we analyze *what-if* scenarios following our discussion about potential optimizations in Sec. 5.1. We point out that although the current version of SPLIT offers flexibility, the results can be seen as *worst-case* times, based on the required steps. To analyze the potential benefit of an optimized version using pre-linked modules, we calculate the processing time without the need to link *global* symbols. As illustrated in Fig. 4 (SPLIT-Opt), this could reduce the overhead to 23.34 % (23.13 s in total) for the client. Similarly, on the server, this can reduce the overhead to 21.92 % (19.25 s). Although these overheads are not negligible, this is a worst-case evaluation, as we can only load one module at a time. Thus, we further analyze a special version of SPLIT, called SPLIT-Opt+, where we also assume the *local* symbols to be set in the module *after* the first load. This is possible as only one module is active at a time in this setting and thus the memory location is always the same, i.e., intra-module dependencies do not change their address. Following this, the times for subsequent operations can be reduced, i.e., 19.67 s (+ 4.88 %) for the client and 19.81 s (+ 4.9 %) for the server.

6 Deployment Considerations

Although SPLIT allows to increase the available functionality, this comes at the price of induced overhead, with respect to time and energy. In the following, we discuss aspects that have to be considered and propose mitigation strategies.

Energy Overhead: To assess the energy overhead, we evaluated a toy example on a real Zolertia Z1. We measured execution time and power consumption for module loading utilizing a measurement platform[2]. Although tasks like wireless communication consume much more energy, excessively accessing the flash can add noticeable consumption and reduce battery lifetime. Thus, depending on the use case, a careful execution plan can limit this effect, e.g., a DTLS security association may stay active for a certain amount of connections, instead of requiring a handshake over and over again. We argue that this trade-off is acceptable to realize a flexible, extensible, and thus sustainable deployment.

Adapting to available resources: For a modularized protocol, each loading of a module adds overhead. Based on the available memory, the size of modules can be increased by adding functionality to save loading steps. Exemplary, the last message of a DTLS flight triggers the first transmission of the following flight (cf. Fig. 3). Thus, combining corresponding functionality saves one loading step. While this increases performance at the cost of higher memory usage, we can also pursue the opposite direction: Decreasing the size yields more loading steps, but decreases memory usage.

Reducing latency: For communication protocols, on-demand loading of modules upon packet reception increases latency. However, protocol determinism allows us to load modules in advance, e.g., while waiting for reception of an initial message or response, the respective modules can already be loaded. Devices that act as a server may receive an initial message from a remote peer at any time. To have modules available upon reception, they could preload modules for corresponding processing. Similarly, protocol determinism allows determining which type of message is expected next.

Mitigating DoS threats: Using SPLIT requires precaution to not make devices prone to DoS attacks. By alternating packet types, an attacker can force a device to load and unload modules. However, SPLIT only accounts for the on-demand loading overhead, i.e., forcing a device to process a packet is not specific to SPLIT. Thus, heavyweight protocols typically implement DoS protection (cf. Sec. 4.1). As such mechanisms are lightweight to not introduce new DoS potential, keeping them in memory is reasonable, and requires an attacker to pass them to trigger operations of SPLIT. Moreover, devices activate this protection only in case of a presumed attack [10], not occupying memory during typical operation.

7 Conclusion

In this paper, we enable IoT devices to support a broad set of functionality. To this end, we present SPLIT which enables on-demand loading of functionality outsourced as (multiple) modules to the significantly lesser constrained flash storage of devices. We explicitly target applications and protocols above the transport layer and adapted an implementation of the commonly used security protocol DTLS to support SPLIT, whereas the concept of SPLIT is not limited to this. Our worst-case evaluation, e.g., only *one* active module at a time, shows the principle applicability of SPLIT on resource constrained devices. Moreover, we identified aspects that need to be taken into account for actual deployment. We argue that with careful planning, a key aspect of future work, SPLIT's benefits can outweigh its overheads.

With SPLIT, we complement existing mechanisms that enable devices to cope with limited processing and energy resources, as well as low power wireless communication. By enabling devices to execute various functionality ranging from diverse secure communication protocols to interaction with their environment, we contribute to this active development of the IoT and its functionality-rich vision. Eliminating the need to replace constrained devices, SPLIT can help in realizing a more sustainable deployment.

8 Acknowledgments

We thank Matteo Ceriotti for the fruitful discussions. This work has been funded by the German Research Foundation (DFG) as part of the CRC 1053 MAKI. We would like to thank the DFG for the support within the Cluster of Excellence "Integrative Production Technology for High-Wage Countries".

9 References

[1] L. Atzori et al. The Internet of Things: A survey. *Computer Networks*, 54(15), 2010.
[2] E. Barker et al. NIST Special Publication 800-57 Recommendation for Key Management, Part 1, Rev 3: General. *NIST SP*, 2012.
[3] C. Bormann et al. Terminology for Constrained-Node Networks. RFC 7228 (Informational), 2014.
[4] W. Dong et al. Optimizing relocatable code for efficient software update in networked embedded systems. *ACM TOSN*, 11(2), 2014.
[5] A. Dunkels et al. Run-time dynamic linking for reprogramming wireless sensor networks. In *ACM SenSys*, 2006.
[6] A. Dunkels et al. Contiki-a lightweight and flexible operating system for tiny networked sensors. In *IEEE LCN*, 2004.
[7] R. Hummen et al. A Cloud design for user-controlled storage and processing of sensor data. In *IEEE CloudCom*, 2012.
[8] R. Hummen et al. Slimfit – A HIP DEX compression layer for the IP-based Internet of Things. In *IEEE WiMob*, 2013.
[9] R. Hummen et al. Delegation-based authentication and authorization for the IP-based Internet of Things. In *IEEE SECON*, 2014.
[10] R. Hummen et al. Tailoring end-to-end IP security protocols to the Internet of Things. In *IEEE ICNP*, 2013.
[11] T. Kivinen. Minimal Internet Key Exchange Version 2 (IKEv2) Initiator Implementation. RFC 7815, 2016.
[12] J. Koshy et al. VMSTAR: synthesizing scalable runtime environments for sensor networks. In *ACM SenSys*, 2005.
[13] H. Kwon et al. Challenges in Deploying CoAP Over DTLS in Resource Constrained Environments. In *WISA*. 2015.
[14] G. Montenegro et al. Transmission of IPv6 Packets over IEEE 802.15.4 Networks. RFC 4944 (Proposed Standard), 2007.
[15] R. Moskowitz et al. HIP Diet EXchange (DEX). IETF Internet-Draft draft-ietf-hip-dex-05, Feb. 2017. Work in Progress.
[16] E. Rescorla et al. Datagram Transport Layer Security Version 1.2. RFC 6347 (Proposed Standard), 2012.
[17] P. Ruckebusch et al. Gitar: Generic extension for internet-of-things architectures enabling dynamic updates of network and application modules. *Ad Hoc Networks*, 36, 2016.
[18] Z. Shelby et al. The Constrained Application Protocol (CoAP). RFC 7252 (Proposed Standard), 2014.
[19] A. Taherkordi et al. Optimizing Sensor Network Reprogramming via in Situ Reconfigurable Components. *ACM TOSN*, 9(2), 2013.
[20] H. Wirtz et al. Enabling ubiquitous interaction with smart things. In *IEEE SECON*, 2015.
[21] Z. Yang et al. R-code: Network coding based reliable broadcast in wireless mesh networks with unreliable links. In *GLOBECOM*, 2009.
[22] X. Zhu et al. ReLog: A systematic approach for supporting efficient reprogramming in wireless sensor networks. *Journal of Parallel and Distributed Computing*, 102, 2017.

[2]http://www.comsys.rwth-aachen.de/short/powergraph