Paper published at the 1st Workshop on Benchmarking Cyber-Physical Networks and Systems. Porto, Portugal. April 2018.

Moving Beyond Competitions: Extending D-Cube to Seamlessly Benchmark Low-Power Wireless Systems

Markus Schuß, Carlo Alberto Boano, and Kay Römer
Institute of Technical Informatics, Graz University of Technology, Austria
{markus.schuss, cboano, roemer}@tugraz.at

***Abstract*—Performance comparisons of low-power wireless systems are often not substantiated by accurate and realistic evaluations, which raises the need of a proper benchmark. In a first attempt towards a rigorous comparison of protocol performance under the exact same settings, we have developed in 2016 a prototype benchmarking infrastructure called D-Cube, and used it to run the first of a series of competitions aiming to quantitatively assess the performance of low-power wireless protocols in specific scenarios. Given the success of the competition among both academia and industry, we have significantly extended the benchmarking infrastructure in the following two editions: D-Cube now also supports, among others, remote experimentation, multiple traffic patterns and loads, a custom description of how to derive performance metrics, and is further able to control the network density as well as the harshness of the RF environment. In this paper we perform a critical analysis of the current capabilities of D-Cube and argue that its main limiting factor is that the traffic patterns and node identities are manually embedded in the source code by developers and cannot be changed automatically. We show that we can overcome this limitation by utilizing a well-known data structure and by having developers describe its memory address using a configuration file that is passed to the benchmarking infrastructure. Following this concept, we extend D-Cube with the ability of building and applying patches to binary files and show that this allows not only to automatically change traffic patterns and node identities, but to also change user-defined protocol parameters. We believe that this extension is one of the last missing stepping stones to make D-Cube a full-fledged benchmarking infrastructure for low-power wireless systems.**

***Index Terms*—benchmarking; competition; dependability; IoT; low-power wireless; measurement; performance; testbeds.**

I. Introduction

An increasing number of Internet of Things applications impose strict dependability requirements on network performance and require the employed communication protocols to deliver information in a reliable, efficient, and timely manner. In response to this need, many low-power wireless protocols have been proposed by industry and academia over the last decade.

The problem. Most of these protocols are validated by simulation or using large-scale public testbeds such as Indriya and FlockLab. In all these settings, developers typically define an ad-hoc evaluation scenario, extract the metrics of interest, and draw conclusions about the performance of their solution. However, the lack of a standardized methodology to evaluate protocol performance often leads to a high divergence across experimental setups, which makes it impossible to compare results obtained by different authors [7]. As a consequence, there is an increasing need to rigorously *benchmark* low-power wireless systems under the exact same settings.

Benchmarking requirements. A low-power wireless benchmark ideally consists of an experimental setup enabling the automated, seamless, and repeatable execution of experiments on real hardware [3]. Such a benchmarking infrastructure should allow a user to define *parameters* that directly characterize the system behavior, e.g., (i) traffic parameters such as pattern and load; (ii) system parameters such as network density; (iii) experiment parameters such as duration and number of runs; as well as (iv) environmental parameters, such as the amount of RF interference that should be generated in the surroundings of the wireless nodes. The benchmarking infrastructure should also allow a user to define *metrics* quantifying the performance of the system under test, such as the packet delivery rate, the energy consumption, and the end-to-end latency [3]. In the ideal case, the user just needs to provide the firmware to be tested and to specify concrete values for the set of parameters, as well as a description of how the performance metrics should be measured. The benchmarking infrastructure then autonomously executes multiple runs and returns a report containing the performance metrics of interest.

Our first attempt. As no benchmarking infrastructure satisfying these requirements existed, we have started in 2016 the EWSN Dependability Competition Series [2] as a first attempt to rigorously benchmark the performance of low-power wireless systems in harsh RF environments. With this long-term goal in mind, we have created *D-Cube*, a low-cost tool that allows to accurately measure key dependability metrics such as end-to-end delay, reliability, and power consumption, as well as to graphically visualize their evolution in real-time [8]. Each competition defines an evaluation scenario emulating the operation of wireless networks monitoring discrete events in the presence of a crowded RF spectrum, where all protocol and system parameters are specified in advance and do not change over time. The competing teams carefully tune their system to the specific evaluation scenario at hand in a dedicated preparation phase, during which they are aware of each other's performance. At the end of this phase, each team provides a final firmware whose performance is benchmarked against all others in terms of reliability, latency, and energy consumption.

An evolving infrastructure. Following the needs of the contestants and in order to improve the competition format, we have significantly extended the capabilities of D-Cube over the years. First, as it is hard to properly optimize a system in a few days only, we have supported remote experimentation.

Second, as the competition scenario originally supported only very specific parameters (e.g., only point-to-point traffic) and the competition results could hence not be easily generalized, we have implemented support for multiple traffic patterns and the ability to disable specific nodes on-demand (e.g., to vary network density). Furthermore, we have added the ability to input events via multiple GPIO pins instead of relying on platform-specific sensors (e.g., light sensors). Since the 2018 edition of the competition, we have also introduced the possibility to input to D-Cube the description of how to derive the performance metrics from the measurement traces. This allows our benchmarking infrastructure to automatically compute the results of a run and to list them in a *leaderboard* summarizing all results for a specific set of parameters. Therefore, as of today, D-Cube can support many of the features that an ideal low-power wireless benchmark should offer, although some of them are intertwined with the specific application scenario used in the competition and cannot be easily generalized.

Moving forward. In order to use D-Cube as a generic benchmarking infrastructure, a necessary step is to move beyond the static competition scenarios encompassing a single set of parameters (i.e., just a small instance of the design space). We hence need to provide D-Cube with the ability to (i) support *multiple sets of* parameters (e.g., different traffic patterns and loads) as well as (ii) to automatically instantiate several runs with a diverse combination of such parameters (i.e., a different benchmark *profile*). Towards this goal, the main challenge is that traffic patterns and node identities are currently manually embedded in the source code by developers and cannot be changed automatically. Therefore, we investigate how to overcome this limitation without constraining developers to a specific application scenario and provide a proof-of-concept solution in which D-Cube can automatically change traffic patterns and node identities by building and applying patches to binary files. We believe that this extension is one of the last missing stepping stones to make D-Cube a full-fledged benchmarking infrastructure for low-power wireless systems.

Contributions. In this paper we outline how the benchmarking infrastructure created to support the EWSN dependability competition series has significantly evolved in the past three years and show that it now supports – among others – remote experimentation, a set of configurable parameters, as well as a custom description of performance metrics (Sect. II). We then argue that, to seamlessly benchmark low-power wireless systems, the main limiting factor of D-Cube is that the traffic patterns and node identities are embedded in the source code by developers and cannot be changed automatically (Sect. III). We hence describe how we can overcome this limitation by making use of a well-known data structure and by extending D-Cube with the ability of building and applying patches to binary files (Sect. IV). With this extension, D-Cube is able to automatically change not only the traffic patterns and node identities, but any user-defined protocol parameter, such as the node initiating a Glossy flood. We finally conclude the paper with a summary and an outlook on future activities (Sect. V).

II. Three Years of Dependability Competition

Since 2016, we organize every year a dependability competition co-located with the International Conference on Embedded Wireless Systems and Networks (EWSN) [2].

In the first two editions, the task of the contestants was to design a system in which a source node monitors the brightness of a light source in close proximity using its embedded light sensors. Any sudden change in the lighting condition had to be promptly reported to a sink node by communicating over a wireless mesh network, such that the sink could replicate the state of the lighting source (on/off) using a GPIO pin. To emulate a congested RF environment, repeatable interference was generated using Jamlab [4].

The competing solutions were evaluated using three metrics describing the system's end-to-end performance: (i) reliability (i.e., the number of changes in lighting condition correctly relayed from the source to the destination); (ii) end-to-end latency (i.e., the time between the change in lighting condition and the instant in which the GPIO pin of the destination node reflected this change); as well as (iii) energy consumption (i.e., the amount of energy consumed by all nodes in the network).

A. Key changes introduced by the 2018 edition

The dependability competition has evolved over the years and several key changes were introduced by its 2018 edition.

i) Multiple events. The first two competitions focused on the detection of a single type of event (change in lighting condition) using the embedded sensors of a source node. In the 2018 edition, up to 8 types of events had to be detected and reported using the GPIO pins available on the source nodes.

ii) Multiple traffic patterns. In the last competition, the wireless system to be designed had to support not only point-to-point (P2P) traffic (i.e., from a single source to a single destination), but also point-to-multipoint (P2MP) and multipoint-to-point (MP2P). In the case of P2P traffic, all changes of a specific GPIO pin on a source node had to be transmitted to a pre-defined destination node, which would toggle one of its GPIO pins accordingly. P2MP traffic relayed state changes in a specific GPIO pin of a source node to multiple destination nodes. A state change was considered to be reported correctly only if all destination nodes did reproduce it. MP2P traffic relayed changes in the state of a GPIO in multiple source nodes to a single destination, which had to logically OR all these state changes and reproduce the resulting pattern on a pre-defined GPIO pin. Multiple sets of source and destination nodes for each of the three patterns were supported, for a total of 11 source nodes, 13 destination nodes, and 27 nodes available as a forwarder.

iii) Remote experimentation. In the first two editions, the benchmarking infrastructure was installed at the venue hosting the EWSN conference. The competitors physically gathered there and had approximately 48 hours of preparation time to study the evaluation scenario and parametrize their solutions accordingly. The last competition, instead, was run remotely, and the competitors had more than two months time to experiment and optimize their systems.

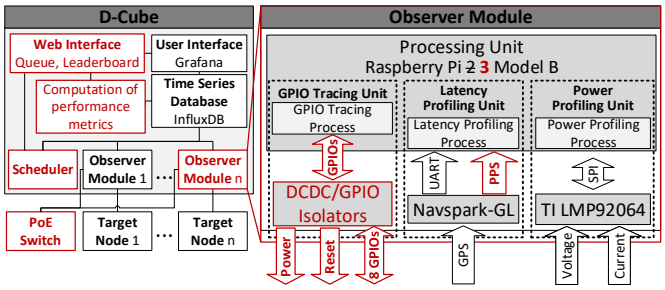
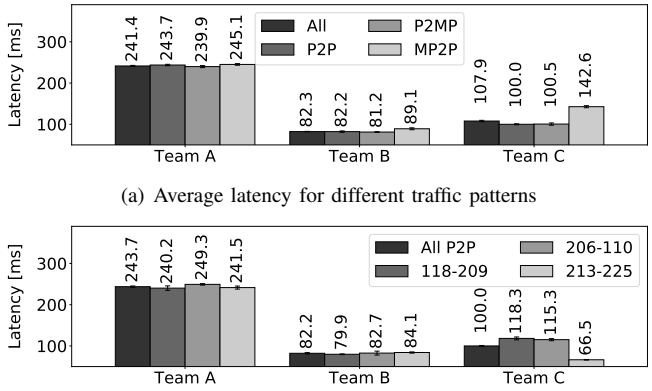


Fig. 1. D-Cube testbed infrastructure (updated components are shown in red).



(a) Average latency for different traffic patterns

(b) Average latency for different P2P source-destination pairs

Fig. 2. Average latency of different teams during the last EWSN'18 dependability competition. D-Cube was used to benchmark the performance of the different teams as a function of various traffic patterns and node identities.

iv) More challenging RF environment. In the 2018 edition of the competition, we no longer made use of JamLab to emulate a congested RF environment, but used instead many Raspberry Pi 3 nodes generating Wi-Fi traffic with diverse characteristics.

B. Evolution of D-Cube

In order to support the aforementioned changes, many new features were added to the competition's benchmarking infrastructure (called D-Cube[1] [8]). The latter has significantly evolved over the years: whilst in the first edition of the competition, experiments were scheduled manually, the entire execution, evaluation, and ranking of the submitted solutions can now be performed automatically, and the evaluation results are published on a leaderboard visible by all contestants.

Changes in D-Cube's hardware. As shown in Fig. 1, D-Cube no longer needs to rely on an existing testbed infrastructure, as both observer modules (upgraded to Raspberry Pi 3 devices) and target nodes (TelosB replicas) can be powered using Power over Ethernet (PoE). Target nodes can be individually powered on and off by software, hence allowing to control the network density and to emulate node failures (e.g., due to an early battery depletion). The GPIO tracing and latency profiling units were redesigned to allow simultaneous tracing and actuation of up to 8 GPIO pins, and nodes without GPS connectivity can synchronize to the rest of the network using NTP with an accuracy well below 10 μs.

Changes in D-Cube's software and back-end. As shown in Fig. 1, D-Cube features a new Web interface allowing contestants to upload their firmware (a single `.ihex` file) and to configure a number of parameters such as the generation of external interference, as well as the duration of an experiment. Experiments are automatically executed using a round-robin scheduling algorithm during night or public holidays, in order to keep uncontrolled interference at the minimum. The Web interface can also visualize raw measurement traces (i.e., voltage, current, and state of GPIO pins) and output a number of performance metrics (e.g., reliability, end-to-end latency and energy consumption). To compute these performance metrics, we have added the possibility to input into D-Cube the description of how to derive a set of metrics from the measurement traces. A description contains, for example, instructions on how to compute whether a destination node correctly ORed the state changes occurring in the GPIO pins of a set of sources nodes for MP2P scenarios. Such description is used to filter the measurements stored in a centralized database once a run has completed: thanks to the database search and filter operations, the performance metrics can be derived rather efficiently. With the description of how to derive performance metrics from measurements traces, D-Cube can then automatically compute the results of a run and list them on an online leaderboard. The latter summarizes all results obtained for a set of given parameters: a user can, for example, display the results obtained using specific levels of generated interference or traffic patterns. Fig. 2 shows how D-Cube allows to derive the end-to-end latency of different solutions for a specific traffic pattern (Fig. 2(a)), as well as to break down the performance obtained in P2P scenarios only, for different sets of source-destination pairs (Fig. 2(b)).

[1]D-Cube (D^3) takes his name from the three dependability metrics that are observed in the competition: reliability, timeliness, and availability.

III. From Competition to Benchmark

In each instance of the competition, a static setup with a fixed set of parameters was defined and communicated beforehand to all contestants. Although well-defined setups allow to clearly identify which systems are more suitable for a specific application and to push protocol performance to the limits [8], it is often not possible to generalize results. The solutions being benchmarked in competitions are indeed often designed for broader classes of applications (e.g., low-rate data collection with aperiodic traffic). When adapting a generic solution to the specific competition scenario at hand, contestants heavily optimize their code and strive for victory, ending up with their protocols being denatured (i.e., not resembling the original design) or with very customized solutions. As a result, one cannot argue whether the results obtained in the competition by a given solution are actually representative of its goodness and suitability for a generic class of applications or for a larger range of traffic parameters.

An example of this problem can be seen in Fig. 3, which shows the energy consumption of three solutions benchmarked

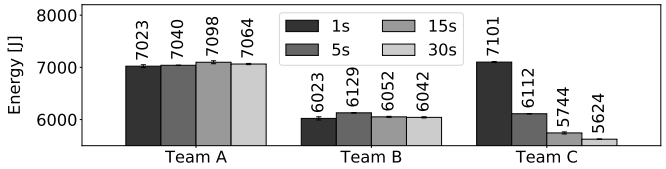


Fig. 3. Energy consumption of three solutions benchmarked at the EWSN'18 dependability competition in the presence of controlled interference. The performance of Team C largely varies as a function of traffic load.

at the EWSN'18 dependability competition as a function of traffic load in the presence of controlled interference. The 2018 edition of the competition made use of a fixed traffic load in which on/off events where generated probabilistically in the range $[1, 11]$ seconds (this choice was driven by the need to maximize the number of events generated in each of the slots allocated to the contestants for experimentation). Therefore, within the competition, the performance of the contestants as a function of traffic load was not captured nor evaluated. Whilst the performance of most solutions was relatively independent of the traffic load, Fig. 3 shows that this is not the case for one of the teams (Team C), whose energy consumption decreases when events are generated at a lower rate[2]. Indeed, the solution provided by Team C performs better than the other two if the application would generate events slightly less often (e.g., every 15 seconds). This aspect, however, could not be captured during the competition, due to its static setup.

A. Moving to a generic benchmarking infrastructure

To generalize the results obtained in the competition, one needs to benchmark the performance of a system under varying settings or for a whole class of applications. Towards this goal, the benchmarking infrastructure needs the ability of supporting multiple profiles and of automatically iterating through them, as discussed next.

Supporting a set of profiles. Each dependability competition corresponds to a single benchmark *profile*, i.e., a concrete set of parameters and performance metrics mapping to a specific real-world application [3]. Ideally, one would test a solution on multiple instances of a competition, i.e., using multiple profiles. For example, one can define four different data collection profiles where the data is periodically generated every 1, 5, 15, and 30 seconds, as shown in Fig. 3, or a profile in which data is sent aperiodically. Similarly, one can specify individual profiles capturing different traffic patterns, e.g., three profiles supporting only P2P, P2MP, and MP2P traffic, respectively. The ability of doing so would allow to better characterize protocol performance and to generalize the obtained results, e.g., one could show that the solution provided by Team C is suitable for applications with variable data rates (see Fig. 3). One would also be able to better assess the performance when using specific traffic patterns. Using the current competition setup this is possible (see Fig. 2), but the benchmarked solution needs to support multiple traffic patterns at the same time. The performance of the tested solution would likely be different when tailored to a single type of traffic only.

Automatic iteration through a set of profiles. Given a set of profiles, a generic benchmarking infrastructure needs to automatically execute different runs for each profile and return a report containing the desired performance metrics. Besides automatically scheduling experiments with different parameters (e.g., setting up runs using different traffic loads), an important feature of a benchmarking infrastructure is the ability to vary the *identity of the nodes* involved in the communication. In the competition setup, this is not the case, as the identities of source and destinations nodes are known beforehand. The ability to automatically shuffle the identity of transmitters and receivers would allow to better cover the space of possible link qualities and hop distance between nodes, hence allowing a more comprehensive characterization of protocol performance.

Static application specifications. The competition setup currently interacts with the firmware under test using GPIO pins. The latter are toggled in order to trigger the transmission of *binary* events from selected source nodes to a (set of) destination node(s). As detailed in [8], this choice allows an unobtrusive interaction with the firmware under test without the need for the developer to use additional libraries and without significantly affecting performance metrics such as latency and energy consumption. A generic benchmarking infrastructure should, however, embed the ability to pass *arbitrary commands* to the firmware under test in an unobtrusive way. For example, one should have the ability to instruct node 14 to send one packet with a payload length of 45 bytes to nodes 26, 31, and 45, or to enter a low-power state.

B. Which features is D-Cube missing?

We analyze next the current capabilities of D-Cube and point out which limitations should be addressed in order to make the benchmarking infrastructure more generic.

Already supported features. Fig. 4 shows the current architecture of D-Cube, including the new features added to support the EWSN'18 dependability competition (see Sect. II-B). Most of the inputted parameters can be changed without the need to interact with the developers. The traffic load can be provided to D-Cube as a description of the number of events to be generated (either an interval or a probability distribution). The harshness of the RF environment can be given as input to D-Cube by providing a description of the characteristics of the interference generated by the Raspberry Pi 3 devices. Experiment parameters such as the duration of a run and the number of repetitions can also be passed to the benchmarking infrastructure as input. D-Cube also has the ability to control the density of the nodes in the network by individually powering devices on and off. Finally, as discussed in Sect. II-B, a scheduler able to process the inputted parameters, the provided firmware, and a description of the performance metrics already exists. This scheduler automatically executes and evaluates the benchmarked firmware, generating a human-readable report

[2]The reliability of all three teams did not exhibit major differences when changing the traffic load, with $\geq 99\%$ of the events being correctly reported.

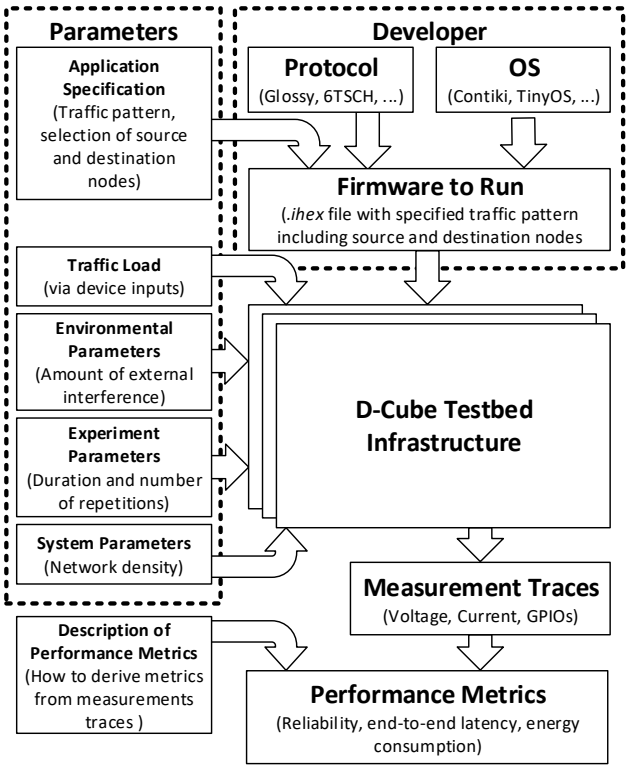


Fig. 4. D-Cube's architecture during the EWSN'18 dependability competition.

and storing persistently machine-readable performance metrics that can be used to rank the tested solutions.

Parameters intertwined with source code. As shown in Fig. 4, the application specification currently contains the traffic pattern as well as the identity of source and destination nodes. Node identities hence cannot be changed by D-Cube automatically without having access to the actual source code. In order to change traffic pattern and node identities without introducing significant overhead, a novel approach is required to alter their configuration directly in flash. This could be achieved by providing the developer with a well-known data structure in the form of a header file containing a C `struct`, as well as a textual description of its fields. By specifying the memory location of this `struct` in the `.ihex` file, the benchmarking infrastructure could automatically alter the values of all configuration variables using a binary patching framework. We provide a proof-of-concept of this approach in Sect. IV.

Support of binary events only. D-Cube does not currently support passing arbitrary commands to the benchmarked firmware. In principle, UART messages could be used to send specific commands to the tested application (e.g., following the principle of Contiki's shell), as they can embed a large amount of info. However, these messages are handled by a dedicated USB-to-UART converter embedded on the target node, which introduces uncontrollable latency and jitter to the measurements. For example, the FT232BM used by the TelosB nodes introduces at least a 1 ms latency as well as a non-constant overhead in current of $\approx$ 20 mA. Bypassing such a converter would require extensive modifications to the target node. Moreover, parsing UART messages introduces significant overhead in RAM and the necessary libraries limit the amount of flash memory available. In contrast to UART messages, the interaction with the benchmarked firmware using GPIO pins (as currently supported by D-Cube) ensures to not alter the behavior of the tested solution, as well as a minimal overhead. The proposed binary patching framework to separate traffic pattern and node identities from the application specifications in Sect. IV could also be used to send a predefined set of commands using the GPIO pins. A command can trigger an action upon reception of an event, which is well-suited for latency measurements. If a command requires more than one bit of data (e.g., to specify a payload or a destination address), information could be serialized over a GPIO pin. In absence of hardware support to automatically parse the data, the target node could emulate serialization in software[3].

IV. Patching Binary Files: Proof of Concept

To separate traffic pattern and node identities from the application specification, we extend D-Cube with the ability of building and applying patches to binary files following the concept shown in Fig. 5. The application specification given to the developers now contains a header file embedding a well-known data structure in the form of a C `struct`, as well as a textual description of its fields. The developers need to first add a new section to their firmware containing an instance of the C `struct` and assign a default value for all its fields, e.g., traffic patterns and identity of source and destination nodes (`traffic_pattern`, `source_id`, and `destination_id` in the example[4] shown in Fig. 5). These default values are placeholders and will automatically be replaced by the binary patching framework. To this end, the developers also provide D-Cube with an `.xml` file containing the memory address at which the C `struct` was placed (e.g., `0xd400`). Using a Python script, the binary patching framework seamlessly modifies the base firmware's `.config` section and creates a firmware ready to run – without involving the developer, without requiring access to the source code, and without introducing any overhead at run-time.

We describe next a preliminary implementation of the binary patching framework (Sect. IV-A) and show that D-Cube can now modify not only traffic patterns and node identities, but even user-defined protocol parameters (Sect. IV-B).

A. Binary patching

We implemented a binary patching framework using Python that takes as input: (i) a base firmware as `.ihex` file, (ii) the description of traffic patterns, node identities, and user-defined variables in an `.xml` file, as well as (iii) the new

[3]An example of such emulation is the 1-Wire bus used by TelosB nodes to communicate with the embedded DS2411 ID chip.

[4]Please note that this is a simplified example supporting only one concurrent pattern with up to 8 source and 8 destination addresses. In principle, one can implement an array with any fixed number of source and destination nodes.

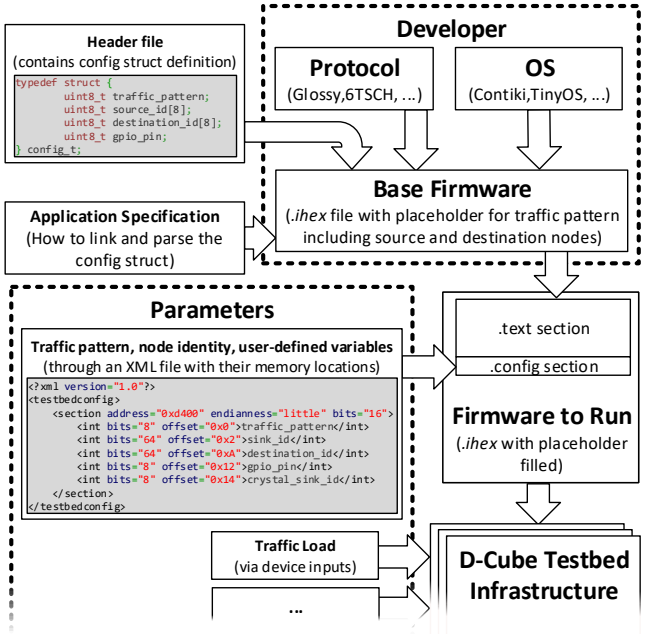


Fig. 5. Changes to D-Cube's architecture in order to support binary patching.

values for the parameters in the C `struct` described by the `.xml` file and produces a new firmware to run (see Fig. 5). As `.ihex` files are compressed ASCII encoded representations of binary information and include a checksum for each line, they cannot be modified directly. Therefore, we use a Python script to perform the patching operation in three steps. First, the script calls gcc's `obj-copy` to create an `.elf` file from the base firmware. Second, as the `.elf` format is still compressed, the script reads out the `.elf` file's headers. The latter describe at least two sections: `.text` (i.e., the program code of the base firmware) and `.config` (acting as placeholder configuration), and further contain the offset of each section in the uncompressed address space, as well as the offset in the compressed `.elf` file. The script then reads the offsets for all parameters in the C `struct` from the `.xml` file and replaces the variables in the `.config` section with the new values provided as arguments to the script. The last step consists in using `obj-copy` to generate a new `.ihex` file with filled placeholders from the modified `.elf`.

B. Exposing user-defined protocols parameters

The binary patching framework described in Sect. IV-A can also be used to modify user-defined parameters. The developer may indeed choose to expose additional parameters in the provided `.xml` file to check whether the chosen protocol parameters actually deliver the best average performance.

We illustrate how this can be done with a running example making use of the publicly available source code of Crystal [5]. A key configuration parameter of Crystal is the node that is used as Glossy flood initiator [6]. We hence expose this protocol parameter to D-Cube by adding to the `.xml` file a user-defined configuration parameter named `crystal_sink_id`. The latter is the name of the variable used to identify the node initiating a flood in the original firmware. We then instruct

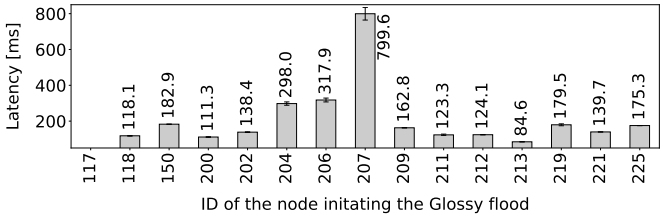


Fig. 6. Average latency of Crystal when using different Glossy flood initiators.

D-Cube to execute several runs using a set of different nodes as flood initiators and a setup similar to the one employed in the EWSN'18 dependability competition. Using the proposed binary patching framework, D-Cube automatically replaces the `crystal_sink_id` variable and allows to derive the results shown in Fig. 6. The latter shows that selecting node 213 as flood initiator allows to obtain the shortest average latency, and that when selecting node 117 as initiator, Crystal cannot successfully communicate. Whilst D-Cube can be used to automatically alter user-defined variables, understanding of their function relies on documentation provided by the developer. If such variables are exposed, reasonable default values and their range should be provided in textual form.

V. Conclusion and Future Work

In this paper we perform a critical analysis of D-Cube and argue that it already embeds many of the features desired by a generic benchmarking infrastructure. Its main limitation is that traffic patterns and node identities are currently embedded in the source code by developers and cannot be changed automatically. We show that we can overcome this limitation by extending D-Cube with the ability of building and applying patches to binary files: developers can now make use of well-known data structures and describe their location using a configuration file. In future work we will enrich D-Cube with the ability to record environmental effects (so to enable a fairer comparison between protocols [3]) and to control other environmental aspects (e.g., temperature [1]). We will further explore how to serialize commands over GPIO pins.

Acknowledgments

This work was performed within the LEAD-Project "Dependable IoT in Adverse Environments", funded by TU Graz.

References

[1] C. A. Boano et al. TempLab: A Testbed Infrastructure to Study the Impact of Temperature on Wireless Sensor Networks. In *Proc. of IPSN'14*.
[2] C. A. Boano et al. EWSN Dependability Competition: Experiences and Lessons Learned. In *IEEE Internet of Things Newsletter*, 2017.
[3] C. A. Boano et al. IoTBench: Towards a Benchmark for Low-power Wireless Networking. *Proc. of the* 1^{st} *CPSBench Workshop*, 2018.
[4] C. A. Boano, T. Voigt, C. Noda, K. Römer, and M. A. Zúñiga. JamLab: Augmenting Sensornet Testbeds with Realistic and Controlled Interference Generation. In *Proc. of the* 10^{th} *IPSN Conf.*, 2011.
[5] T. Istomin et al. https://github.com/d3s-trento/crystal/tree/depcomp18.
[6] T. Istomin, A. L. Murphy, G. P. Picco, and U. Raza. Data Prediction + Synchronous Transmissions = Ultra-low Power Wireless Sensor Networks. In *Proc. of the* 14^{th} *SenSys Conf.*, 2016.
[7] S. Duquennoy et al. A Benchmark for Low-power Wireless Networking. In *Proc. of the* 14^{th} *ACM SenSys Conference, poster session*, 2016.
[8] M. Schuß et al. A Competition to Push the Dependability of Low-Power Wireless Protocols to the Edge. In *Proc. of the* 14^{th} *EWSN Conf.*, 2017.