Non-Intrusive Distributed Tracing of Wireless IoT Devices with the FlockLab 2 Testbed

ROMAN TRÜB, RETO DA FORNO, LUKAS DASCHINGER, and ANDREAS BIRI,
ETH Zurich, Switzerland
JAN BEUTEL, University of Innsbruck, Austria
LOTHAR THIELE, ETH Zurich, Switzerland

Testbeds for wireless IoT devices facilitate testing and validation of distributed target nodes. A testbed usually provides methods to control, observe, and log the execution of the software. However, most of the methods used for tracing the execution require code instrumentation and change essential properties of the observed system. Methods that are non-intrusive are typically not applicable in a distributed fashion due to a lack of time synchronization or necessary hardware/software support. In this article, we present a tracing system for validating time-critical software running on multiple distributed wireless devices that does not require code instrumentation, is non-intrusive and is designed to trace the distributed state of an entire network. For this purpose, we make use of the on-chip debug and trace hardware that is part of most modern microcontrollers. We introduce a testbed architecture as well as models and methods that accurately synchronize the timestamps of observations collected by distributed observers. In a case study, we demonstrate how the tracing system can be applied to observe the distributed state of a flooding-based low-power communication protocol for wireless sensor networks. The presented non-intrusive tracing system is implemented as a service of the publicly accessible open source FlockLab 2 testbed.

CCS Concepts: • **Networks** → **Protocol testing and verification**; • **Software and its engineering** → **Software testing and debugging**; • **Hardware** → *Wireless devices;*

Additional Key Words and Phrases: Distributed debugging, on-chip debug and trace, testbed, IoT, FlockLab, wireless sensor network

ACM Reference format:
Roman Trüb, Reto Da Forno, Lukas Daschinger, Andreas Biri, Jan Beutel, and Lothar Thiele. 2021. Non-Intrusive Distributed Tracing of Wireless IoT Devices with the FlockLab 2 Testbed. *ACM Trans. Internet Things* 3, 1, Article 5 (October 2021), 31 pages.
https://doi.org/10.1145/3480248

1 INTRODUCTION

The development and validation of networking protocol implementations for distributed IoT devices is challenging. Each target node contains its own local state that is influenced by

Authors' addresses: R Trüb, R. Da Forno, L. Daschinger, A. Biri, and L. Thiele, Computer Engineering and Networks Laboratory (TIK), ETH Zurich, Gloriastrasse 35, 8092 Zürich, Switzerland; emails: {rtrueb, rdaforno, ldaschinger, abiri, thiele}@ethz.ch; J. Beutel, Department of Computer Science, University of Innsbruck, Technikerstrasse 21a, 6020 Innsbruck, Austria; email: jan.beutel@uibk.ac.at.

2577-6207/2021/10-ART5
https://doi.org/10.1145/3480248

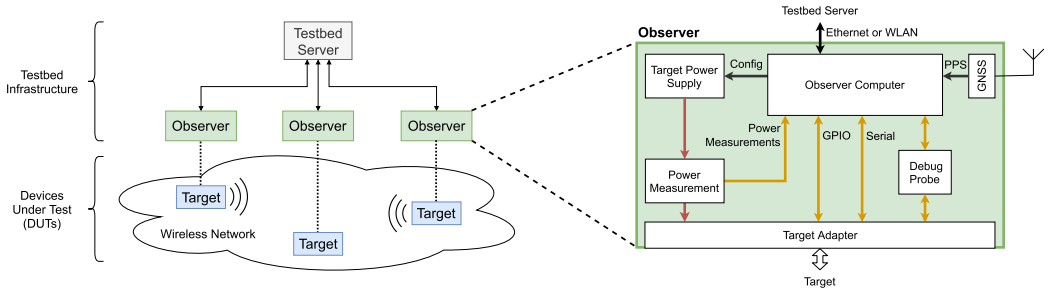

Fig. 1. A common testbed architecture includes observers and a testbed server that represent the testbed infrastructure. This infrastructure allows us to control and observe a set of target nodes that form a wireless network. The exemplary observer, based on the FlockLab 2 testbed [32], supports multi-modal actuation and tracing.

interactions with its environment through peripheral interfaces such as the wireless radio and sensors/actuators attached to the device. In a distributed context, these state changes are additionally based on interactions between the individual nodes via a wireless communication network.

In many cases, the design of target node software is first supported by a simulator or a handful of devices on the developer's desk. Simulations are valuable as a first step but cannot replace testing on actual hardware, since many details of real-world systems have a significant impact on the system operation. Tests with devices on the developer's desk often simplify important aspects as only a limited number of devices under test is feasible and as they are co-located and hence offer limited spacial diversity. These simplifications limit the validation of the concurrent aspects of distributed sensing systems under realistic conditions. For this reason, testbeds for wireless sensor networks are used to debug and validate target node hardware and software before the deployment in a targeted environment.

A basic structure for testbeds that uses a separate set of stateful observer nodes implemented as an infrastructure service on top of the actual target nodes, i.e., the network of IoT devices or sensor nodes, has proven to be useful [1, 7, 16, 26]. We describe the FlockLab 2 [32] testbed, which is a representative example of such a testbed architecture. A high-level overview is depicted in Figure 1. The distributed target nodes, which we call *targets*, are resource constrained devices that usually consist of a microcontroller for collecting and processing information and a low-power radio that is used to communicate with other targets over wireless channels. Each target is supplied, controlled, and monitored by a dedicated device. Since we focus on tracing in this article, we refer to this node as *observer*. An observer interfaces with the target node using one or multiple modalities. Nowadays, supporting multiple different modes of actuation and observation in a single testbed is common best-practice. The observers of the FlockLab 2 testbed provide: support for programming the targets, a configurable power supply, capabilities for logic actuation (writing GPIO signals), support for monitoring of power consumption, logic tracing (reading GPIO signals) as well as interactions via a serial or debug port. Observer nodes are connected to a central testbed server using an out-of-band channel, for example Ethernet. During the execution of a test, the stateful observers collect measurements and traces locally to avoid limitations by the shared network infrastructure. At the end of a test, the testbed server collects and processes all results that are then made available to the developer for analysis.

To monitor the state of software running on distributed target nodes, a variety of methods have been developed and applied, see for example References [18, 27, 29, 30]. The four most important classes of methods are as follows:

- **Serial Interface:** Methods that make use of a serial interface permit the to use `printf`-style logging to capture a diverse set of system states, see for example Reference [10]. Such methods are easy to use, since human-readable text is logged to inform about the system's state. However, the timing accuracy is usually low due to slow UART interfaces and the non-deterministic behavior of further components in the transmission chain. Furthermore, the methods require code instrumentation and are intrusive, thereby influencing the tight timing constraint of wireless protocols.
- **Buffer transfer:** Methods that use buffer transfers, e.g., via CoreSight [19], are a faster variant of `printf`-style logging, see for example Reference [2] where Segger RTT is used. Similarly to approaches using a serial interface, these methods are highly versatile but require code instrumentation and are intrusive. The timing accuracy is limited by delays in the debug channels.
- **Logic tracing:** Logging GPIO pin signals with a logic analyzer or similar hardware provides high timing accuracy and is minimally intrusive, see for example References [26, 32]. At the same time, such methods are limited to very few states (1 bit per GPIO pin). Of course, schemes to serialize more state information and send it over one or multiple GPIO lines can be applied. However, this would increase the code instrumentation and the intrusiveness of such methods significantly.
- **Hardware assisted debugging and tracing:** Tracing with dedicated debug and trace hardware (including built-in subsystems) allows for non-intrusive tracing and does not require code instrumentation, see for example References [14, 29]. However, this is usually only supported for a single device under test. Without sub-millisecond time synchronization, such methods are not suited for the tracing or debugging of distributed devices.

All these methods help to validate and confirm the correct behavior of the software running on distributed targets in a testbed. However, to the best of our knowledge, no testbed architecture provides a truly non-intrusive distributed tracing method without the need to instrument the code. However, this would be necessary to validate the software in the last stage before deployment [34]. As a consequence, the software running on a testbed is in almost all cases different from the software running in an actual deployment, rendering it almost impossible to draw conclusions from testbed tests that are also valid for a deployed system.

Adding or adapting code to trace the system state leads to changes in the functional behavior of the system and most importantly, it potentially changes the timing behavior. Communication protocol software for distributed wireless sensing devices is in many cases highly delay sensitive and even small changes in the code can significantly impact the overall system operation in terms of correctness, throughput, reaction time, reliability and energy consumption. Furthermore, additional instrumentation for tracing potentially influences the timing of the traced events. As a consequence, the correct ordering of events that are traced in the distributed system becomes even more challenging.

Based on our own needs to test timing-critical networking protocols in setups that are as close to the real-world deployment as possible, we worked out the following requirements that should be met by a testbed when debugging and validating target node software before deploying it:

- State and state transitions of the targets need to be observable networkwide. State transitions on different nodes should be chronologically ordered with an accuracy of at least the duration of a single radio message transmission, typically in the order of one millisecond or more.
- The supported event rate should be sufficiently high to detect state changes that only last for a short period of time, e.g., for a few tens of processor clock cycles.

- The method to collect observations should be non-intrusive, i.e., the program execution should not be influenced by the collection of observations. This is important for time-critical parts of the program code such as radio operations in wireless networking protocols.
- No code instrumentation should be required as code for verification is typically not included in the software for a deployed system to reduce power consumption and increase operational stability.

To fulfill these requirements, hardware support inside the microcontrollers is required. To observe the distributed state of all target nodes in a testbed, such hardware support and the corresponding required infrastructure is needed not only at one or a few central locations, but at all target nodes. Such a distributed tracing infrastructure requires testbedwide orchestration and the traces collected at different nodes need to be merged in a way that they allow statements about the networkwide state.

In this article, we present a system for testbedwide non-intrusive tracing of target nodes without requiring code instrumentation. To enable this, we propose the use of a dedicated debug and trace probe for each target node. This permits us to make use of the on-chip debug and trace hardware available in modern microcontrollers (see Figure 2 and Section 3.2) that enables data logging in parallel to the normal program execution without influencing the program execution in any way. The proposed tracing system records memory accesses (read/write) to existing variables, thus there is no need for manual code instrumentation or re-instrumentation if a different aspect of the software is investigated. In addition, the proposed tracing system does not require any debug information that can optionally be generated when compiling the executable binary. The same program image can be used in different test runs to trace different memory locations without the need to recompile the program code. In contrast to commonly used stop-start debugging, the proposed tracing method is completely free of any online interaction with the target nodes during the test execution. This facilitates the repeatability of tests and allows for unattended test execution. However, as described in Reference [32], the same testbed architecture allows the implementation of a stop-start-based online debugging service.

To order and align the traces from different observers on a common time base, all traced events are timestamped twice: (1) a local time is logged by the debug and trace hardware that is part of the microcontroller and (2) the synchronized testbed time is logged on the observer. In postprocessing on the testbed server, the collected data are used to apply time correction, i.e., to add an accurate global timestamp to each traced event. This also permits us to compare the traces with events from other tracing services such as logic or power tracing.

For every event, the resulting tracing data contain a synchronized timestamp, an identifier of the target node, accessed variable (or memory location), mode of access, value of the accessed variable and optionally the instruction memory address of the instruction responsible for the access. The data can be visualized for manual analysis and interpretation (see Section 6.4). Furthermore, a model of the system can be used to predict the output of the tracing. The prediction can then be automatically compared with the actual output of a test on the testbed. We demonstrate the applicability of this approach with a simple example including a networking protocol in Section 6.4.

In a previous paper [32], we described the FlockLab 2 testbed architecture in general. In this article, we focus on the aspect of non-intrusive distributed tracing using on-chip debug and trace hardware. In summary, this article contains the following additional contributions:

- A novel concept, design, and implementation of a non-intrusive distributed tracing system for a network of wireless IoT devices without the requirement for code instrumentation.
- Methodology to time synchronize traces from different independently collected data of the proposed tracing system with a sub-millisecond accuracy.

- Case-study that demonstrates how the proposed tracing system can be used for the validation of software used in distributed wireless IoT devices.

In Section 2, we discuss related tracing systems and compare their advantages and disadvantages. Section 3 introduces the proposed tracing system architecture and Section 4 describes the novel time synchronization methodology. In Section 5, we present a concrete implementation of the proposed tracing system based on the FlockLab 2 testbed and provide characterization values. In Section 6, we apply the tracing method to a timing critical flooding-based communication protocol and show how the resulting tracing data can be used to automatically detect deviations from the design and verify its correct behavior. Finally, we provide concluding remarks in Section 7.

2 RELATED WORK

Multiple works on testbeds for IoT and wireless sensor networks integrate debugging and tracing methods such as halting on break- or watchpoints and inspecting the internal state of the microcontroller when the program is halted. For example, in Clairvoyant [35] this set of methods is implemented by instrumenting the code and by transmitting the data via the wireless radio link. Marionette [33] also uses code instrumentation to integrate remote procedure calls that can be used for debugging. HATBED [36] makes use of the on-chip debug infrastructure for tracing. It supports low latency `printf` logging via **Instrumentation Trace Macrocell (ITM)** as well as **Data Watchpoint and Trace Unit–(DWT)** based watchpoints that are logged using a debug probe and collected via an out-of-band channel. Li et al. [14] also propose a testbed that supports collecting traces using the on-chip debug hardware. However, for both works, it is not clear how the traces of different nodes can be compared or aligned on a common time axis. Lim et al. [18] apply code instrumentation and uses `printf`-style logging combined with logic analyzer traces to reconstruct the program control flow. All of the above approaches require code instrumentation that influences the program execution and are therefore not well suited for tracing delay sensitive software.

Other work is focused on distributed debugging. For example, in CD-TRS [8], Gong et al. propose to use code instrumentation and intrusive `printf`-style logging of events and device status to reconstruct a networkwide trace of the network's state over time. Events are sorted using the time slot numbers of the wireless protocol. However, this work assumes that the target nodes run a real-time wireless networking protocol where all nodes are fully synchronized. In many cases, full time synchronization is not assumed to be given but is part of the functionality to be tested on a testbed.

The Aveksha [29] system uses on-chip debug and trace hardware to trace internal state such as variables and the **program counter (PC)** during program execution in a non-intrusive way, i.e., without the need to halt the execution. The monitoring of the internal state is implemented by periodic memory read accesses (polling) via the JTAG interface. The smallest polling period is larger than 30 µs, which limits the the observation of state changes lasting a shorter period of time. Furthermore, the presented system is not suited for debugging the distributed program execution on a set of devices in a network, since Aveksha does not provide synchronized timestamps for the traced events.

The closest related work is Minerva [27] by Sommer et al., which proposes a system similar to Aveksha. The presented testbed architecture equips each observer with a debug probe and uses an on-chip debug and trace circuit of modern microcontrollers. Similar to Aveksha, Minerva supports non-intrusive tracing of variables and the PC by memory polling as well as stop and start debugging. In addition, it allows collecting memory snapshots without halting the program execution and supports networkwide assertions that are based on the traced data. Sommer et al. analyze the

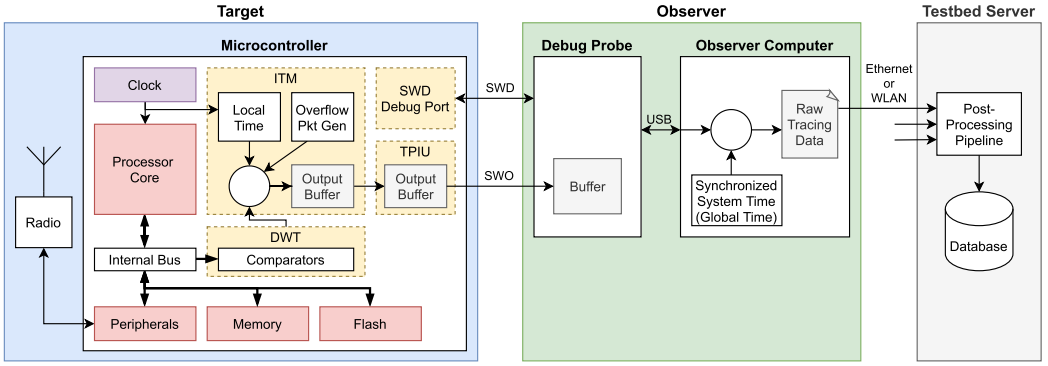

Fig. 2. Architecture for the distributed non-intrusive tracing system proposed in this article. Each observer has a dedicated debug probe to interface with the on-chip debug and trace sub-system (dashed yellow boxes). Other tracing and actuation functionalities, for example available in the FlockLab 2 testbed [32], are omitted in this figure.

overhead for a single memory poll (more than 600 µs on average) and the ordering of traced events and state that their method is not suited for high data rates and precise timing that is required according to Section 1.

In contrast to the discussed approaches, our solution does not use polling but uses the *data trace* feature of the DWT hardware where data are only generated if a change is detected by the on-chip debug hardware. This event-driven approach supports the tracing of higher event rates for a short period of time, e.g., during a burst of events. In addition, the traces of different observers are time synchronized by leveraging timestamps from the on-chip debug hardware during postprocessing. This allows for distributed tracing, i.e., the tracing data of different target nodes can be aligned and compared on a single time axis.

3 SYSTEM ARCHITECTURE

In this section, we describe the system architecture for the proposed distributed tracing method as depicted in Figure 2. We explain the components that enable hardware assisted tracing and discuss the infrastructure that is necessary in a testbed with distributed nodes.

3.1 Overview

The proposed tracing system requires an underlying testbed architecture that is similar to the one of FlockLab 2 [32]. As depicted in Figure 1, the testbed consists of three layers: (1) the target nodes, (2) the observer, and (3) a testbed server. The relevant components and the workflow for collecting tracing data with the proposed scheme are depicted in Figure 2.

The targets are resource constrained devices containing a microcontroller and a low-power radio for wireless communication. For the proposed tracing system, microcontrollers on the target must feature an ARM CoreSight [19] on-chip debug and trace sub-system (dashed yellow blocks in Figure 2) with **Serial Wire Debug (SWD)** and **Serial Wire Output (SWO)** ports. This is prevalent in modern microcontrollers. Of course, microcontrollers or processors with a similar interface and comparable services could be integrated as well.

The observer consists of an observer computer with connections to the testbed server and a debug probe (see Section 3.2.2) that connects the observer computer to the target via the SWD and SWO debug ports. By using the debug probe, the observer controls the target and collects tracing

data from the target. The observer computer requires enough processing and storage resources to perform the necessary tracing activities independently from the testbed server during the test execution.

All observers are globally time synchronized to accurately timestamp the traced data close to the target where the data are generated. For observers with satellite reception, the time synchronization is achieved by synchronizing the system clock of the observer computer to the accurately time synchronized **pulse-per-second (PPS)** signal generated by a dedicated **Global Naviation Satellite System (GNSS)** receiver on the observer. For observers without GNSS reception, accurate time synchronization is achieved via the **Precision Time Protocol (PTP)** protocol and a PTP master device equipped with a GNSS receiver that is accessible by all observers via an Ethernet connection. In addition, the observer computer provides local buffering of traced data, which ensures that enough logging capacity is available for each node, independent of potential congestion of the network between observers and testbed server. The testbed server consists of a computer that is connected to all observers via Ethernet or WLAN.

When a test is run on the testbed, the program is flashed onto the microcontroller of the distributed targets using the debug probe and the SWD port. In addition, the debug and trace hardware that is built into the target's microcontroller is configured for collecting tracing data. During the test execution, the target application is executed on the microcontrollers, and the targets may interact with each other via wireless links. In parallel, the debug and trace sub-systems on the microcontrollers generate tracing events and output tracing data via the SWO ports. Due to the dedicated on-chip debug and trace hardware inside the microcontrollers, the tracing does not influence the program execution and is therefore non-intrusive. The traced data are forwarded to the observer computer via the debug probe. Once the test has ended, the testbed server collects the tracing data from all observers and performs the postprocessing steps. Finally, the aggregated and processed tracing data can be accessed by downloading them from the testbed server.

In this article, we present the proposed non-intrusive distributed tracing system as an extension of the FlockLab 2 testbed architecture. However, the basic scheme of the tracing system is generally applicable and could be integrated in any testbed that satisfies the following requirements:

- The target nodes in the testbed feature the required support for hardware-assisted tracing as described in Section 3.2.1.
- The testbed infrastructure contains an accurately time synchronized and stateful observer node that includes a debug probe such that the traced data can be accurately timestamped and logged close to the target.
- The testbed infrastructure contains a computing device that can collect and post-process all tracing data.

3.2 Hardware Background

The proposed tracing system builds on top of the ARM CoreSight debug and trace hardware that is integrated in many commercially available modern microcontrollers. This on-chip debug and trace sub-system provides hardware assisted tracing functionalities. A debug probe is required to transfer the generated data to the observer computer. In this section, we describe aspects of the ARM CoreSight system and the debug probe that are relevant for the proposed system.

3.2.1 Target. For our tracing system, the targets are required to include the ARM CoreSight debug and trace system (or system with comparable functionalities and a compatible interface). The ARM CoreSight debug and trace hardware [19] is provided by ARM as modular IP cores that are integrated by many manufacturers in modern microcontrollers. A minimal set of

Table 1. Overview of a Selection of Commercially Available Microcontrollers and SoCs That Are Suitable for the Proposed Distributed Tracing System (Non-exhaustive List)

Manufacturer	Product	Core Architecture	Num. DWT Comparators
Atmel	SAM4	ARM Cortex-M4	4
Nordic Semiconductor	nRF52840	ARM Cortex-M4	4
NXP Semiconductors	LPC176x	ARM Cortex-M3	4
Renesas	RA6M1/M4	ARM Cortex-M33/M4	4
Silicon Labs	EFM32WG	ARM Cortex-M4	4
Silicon Labs	EFM32PG12/22	ARM Cortex-M33/M4	4
STMicroelectronics	STM32L1	ARM Cortex-M3	4
STMicroelectronics	STM32L4/F4	ARM Cortex-M4	4
STMicroelectronics	STM32L5	ARM Cortex-M33	4
STMicroelectronics	STM32F7/H7	ARM Cortex-M7	4
STMicroelectronics	STM32WB/WL	ARM Cortex-M4	4
Texas Instruments	MSP432P4xx	ARM Cortex-M4	4

functionalities of the CoreSight system is guaranteed to be available, with many more additional or higher-performance variants and features that are optional. Table 1 contains a non-exhaustive list of commercially available microcontrollers that fulfill the requirements for the proposed distributed tracing system.

The important blocks of the proposed tracing system are the DWT, the ITM, and the **Trace Port Interface Unit (TPIU)**:

- **DWT:** The DWT is a debug block that provides data tracing among other functionalities. It includes a fixed number of hardware comparators that can be configured to trigger events based on data on the internal bus or the PC of the microcontroller. In the proposed system, we use the DWT to collect the so called *data trace*, i.e., recording the accessed memory address, the data value, and the address of the instruction that caused the memory access (PC value) [20]. Traceable memory addresses cover the whole memory address space that is accessible by load and store instructions, including addresses of memory-mapped peripheral devices. Apart from the PC value, no other register values from inside the processor core are included in the data trace.
- **ITM:** Among other functionalities, the ITM arbitrates debug and trace packets from different sources of the debug and trace sub-system. For the tracing system proposed in this article, the ITM is only used to forward the data stream from the DWT to the TPIU output buffer and to add local timestamp packets to this stream.
- **TPIU:** The TPIU provides an interface to output the generated debug and trace stream to an external device, the so-called **Trace Port Analyzer (TPA)**. For the proposed tracing system, we require it to support the SWO that is a serial port similar to traditional UART.

3.2.2 Observer. To (1) allow the observer to instruct the debug and trace sub-system inside the microcontroller that events to trace and (2) to transfer generated tracing data to the observer computer, a commercially available debug probe is used. On ARM-based microcontrollers, a debug probe allows to interact and access the CoreSight components in the microarchitecture.

To control and configure the microcontroller, the debug probe connects to a debug port on the target device, in our case an SWD debug port. To receive tracing data output from the microcontroller, the debug probe acts as a TPA and reads from the SWO port. As depicted in Figure 2, in our testbed architecture, the debug probe is integrated into the observer.

4 TRACING AND TIME SYNCHRONIZATION

The idea of our proposed tracing system architecture is to provide an out-of-band infrastructure to allow for observations that do not influence the system under test. Tracing events on distributed target nodes is meaningful only if the traces from different nodes have the same time base such that the events can be compared and ordered testbedwide. The targets, on which the events are triggered and logged, have independent clock systems. The targets are resource constrained and their clocks usually exhibit non-negligible drift over time. But according to the requirements, we are not allowed to change the target implementation and add time synchronization just for the purpose of testing. Finally, the traced data pass through different buffers on the microcontroller and on the debug probe, adding non-deterministic and non-observable delays.

In the remainder of this section, we explain our approach for synchronizing the time of events traced on different target nodes and discuss the potential use of the obtained data.

4.1 Overall Time Synchronization Architecture

There are different methods to time synchronize event observations in a network of distributed wireless devices. One approach is to use a wireless time synchronization protocol, e.g., PulseSync [13] or Glossy [6], which distributes radio messages precisely at known time instants such that the nodes can reconstruct the reference time. Another approach is the use of dedicated hardware at the target such as a GNSS receiver [17] or a wired synchronization network based on PTP [15], which provides an accurate time reference at the target. However, these methods do not meet the requirements of our proposed tracing system (see Section 1), since they are intrusive. In addition, we do not assume that the system under test provides the infrastructure for accurate time synchronization, but rather we assume that the synchronization of the system is potentially unreliable and thus itself represents a reason for testing and validation. Therefore, we propose to use an out-of-band time synchronization based on two layers. In a first layer, the system time of each observer computer is synchronized to the accurate absolute GNSS-derived time as described in Section 3.1. In a second layer, the tracing data obtained from the target are synchronized to the system time of the observer computer. In the following, we explain the method for the second-layer synchronization that is applied offline using a regression.

Let us denote the time derived from the microcontroller's clock in the ITM as *local time*. For example, an event i on a target node s happens at local time $t_s[i]$. Such an event i as logged on target s is then automatically timestamped in the ITM. Note that the timestamp may not be taken at the very moment of the event i as it may be added some time later to the data trace due to the necessary processing and buffering. We refer to this timestamp as $t_s^{\mathrm{ITM}}[i]$ as it is derived from the local clock of target s.

Due to the high precision of the synchronization of the system time on the observer computer, we can assume that all observers use the same time base and refer to this testbedwide synchronized time as *global time*. When a traced event i arrives on the observer computer, a global timestamp $t^{\mathrm{OBS}}[i]$ derived from the global time is added. More details on the collection of events in the data trace as well as on the format of the data trace are provided in Section 4.2 and Section 4.3, respectively.

In summary, an event i that happens at target s at global time $t[i]$ and local time $t_s[i]$, gets a local timestamp $t_s^{\mathrm{ITM}}[i]$ in the ITM of the microcontroller and a global timestamp $t^{\mathrm{OBS}}[i]$ on the observer. We are now interested to get an estimate of $t[i]$ from $t_s^{\mathrm{ITM}}[i]$ and $t^{\mathrm{OBS}}[i]$. This is necessary to at least partially order the distributed events occurring at the targets.

To simplify the explanation, we first assume that the relation between the local time on the microcontroller and the global time on the observer can be approximated well by a linear relation.

This holds if the drift of the two time bases is constant for the duration of the test. More general schemes, e.g., for cases with clock drifts due to changing temperatures, are discussed at the end of this section. With the assumption of a linear relation, the relation can be expressed as

$$t[i] = t_s[i] \cdot m_s + q_s. \tag{1}$$

In the relation, m_s denotes the slope and q_s denotes the intercept of node s. The idea of this approach is to compensate the drift and offset between the local and global time. The parameters m_s and q_s will be determined by linear regression as described below. Since the reference of the local time changes with a reset of the target, the regression parameters need to be recalculated. Special synchronization packets allow us to partition the test time into time periods between resets (*synchronization epochs*) as described in Section 4.3. To further simplify the discussion, we will at first neglect all possible error terms in the relations.

We now describe how we can determine the global time of an event occurrence $t[i]$ from the timestamp $t_s^{\mathrm{ITM}}[i]$ contained in the data trace. We suppose that we empirically determined an estimate $E(\cdot)$ of the delay $E(\Delta^{\mathrm{ITM}})$ between the occurrence $t[i]$ of event i and the added timestamp $t_s^{\mathrm{ITM}}[i]$. Therefore, we can write $t_s^{\mathrm{ITM}}[i] = t_s[i] + E(\Delta_s^{\mathrm{ITM}})$ where $E(\Delta^{\mathrm{ITM}}) = E(\Delta_s^{\mathrm{ITM}}) \cdot m_s$ relates the timestamping delay in local and global time. We can now determine $t[i] = (t_s^{\mathrm{ITM}}[i] - E(\Delta_s^{\mathrm{ITM}})) \cdot m_s + q_s$, and therefore

$$t[i] = t_s^{\mathrm{ITM}}[i] \cdot m_s + (q_s - E(\Delta^{\mathrm{ITM}})). \tag{2}$$

In other words, given the timestamp $t_s^{\mathrm{ITM}}[i]$ of an event i, we can determine the global time of its occurrence, provided that we know an estimate of the delay $E(\Delta^{\mathrm{ITM}})$ as well as the parameters m_s and q_s. As discussed in Section 4.2, the delay $E(\Delta^{\mathrm{ITM}})$ can be neglected in many cases.

These parameters will be determined using linear regression based on the set of pairs of timestamps $(t_s^{\mathrm{ITM}}[i], t^{\mathrm{OBS}}[i])$ in the data trace for the received events i. To this end, we suppose that we also have available an estimate of the delay $E(\Delta^{\mathrm{TRF}})$ between the timestamp of an event by the ITM and the timestamp of the observer (**transfer (TRF)** time). Then we can derive $t^{\mathrm{OBS}}[i] = t^{\mathrm{ITM}}[i] + E(\Delta^{\mathrm{TRF}})$ where $t^{\mathrm{ITM}}[i]$ denotes the global time of the timestamp added to event i by the ITM with $t^{\mathrm{ITM}}[i] = t_s^{\mathrm{ITM}}[i] \cdot m_s + q_s$. As a result, we obtain

$$t^{\mathrm{OBS}}[i] - E(\Delta^{\mathrm{TRF}}) = t_s^{\mathrm{ITM}}[i] \cdot m_s + q_s. \tag{3}$$

Therefore, given the pairs of timestamps $(t_s^{\mathrm{ITM}}[i], t^{\mathrm{OBS}}[i])$ as well as an estimate $E(\Delta^{\mathrm{TRF}})$, estimations of the relation between local time at target s and the global time according to Equation (1) can be determined by regression of $t^{\mathrm{OBS}}[i] - E(\Delta^{\mathrm{TRF}})$ on $t_s^{\mathrm{ITM}}[i]$.

There are multiple delays between adding the local and adding the global timestamp to the tracing data that need to be considered. A detailed list and discussion of the aggregated delays Δ^{ITM} and Δ^{TRF} is given at the end of Section 4.2. Typically, the expected values of the delays need to be determined experimentally, since the exact models of all delay components are not known in general. In an actual implementation, Δ^{TRF} can reach values in the order of 200 ms.

Due to the uncertainty in the delays Δ^{ITM}, Δ^{TRF} as well as in the stability of the local and global clocks, Equation (2) cannot be satisfied with equality. To take this into account, we introduce an error term $\epsilon[i]$ that leads to

$$t[i] = t_s^{\mathrm{ITM}}[i] \cdot m_s + (q_s - E(\Delta^{\mathrm{ITM}})) + \epsilon[i]. \tag{4}$$

In Section 5.2.1, we experimentally determine an estimate of the error $\epsilon[i]$ for a concrete implementation of the tracing system.

So far, we assumed that the clock source drifts are constant and therefore a single linear regression is used to estimate the relation between local and global timestamps for the time period

between two target resets, i.e., the test duration if there is no user-triggered reset. The drift variations of clocks in microcontrollers are highly dependent on the quality of the clock source and temperature influence [5, 31]. In the following, we determine an estimate of the timestamping error of the data trace system that is caused by the drift of the clock source. For the worst-case, we assume that there are only two clock source frequencies, the minimum and the maximum, and that one is active in the first half and the other in the second half of the test duration. If we assume the target uses a high-quality clock source such as an external crystal oscillator with a deviation of ±25 ppm over the operating temperature range on the target and a test duration of 10 minutes, then we estimate the worst-case deviation of the global timestamp to be 15 ms. In case of a low-quality clock source, such as an internal RC oscillator with a deviation in the order of ±1% over the operating temperature range and the same test duration, we estimate a deviation of 6 s. To reduce the duration during which the clock source can drift away from global time, we apply a piecewise linear regression. Of course more sophisticated regression models, such as cubic regression splines could be applied analogously. For regression intervals of 30 s, the worst-case estimation of the previous example would result in a deviation of 0.75 ms for the external crystal and 300 ms for the internal RC oscillator.

4.2 Collection of Data Trace and Time Synchronization Data

To collect data trace data from the microcontroller on the target, the debug and trace sub-system needs to be configured and the generated data need to be fetched and forwarded to the observer computer. To reconstruct the accurate global time of the events, appropriate timestamps need to be collected as well, see Section 4.1. In the following, we will describe the corresponding configurations and mechanisms as far as they are relevant for the tracing architecture.

The different blocks of the debug and tracing system are configured using the SWD debug port, see Figure 2. The debug probe translates instructions from the observer computer into SWD protocol operations. For every global variable or data memory location that shall be traced, we configure one DWT comparator to trigger whenever the corresponding address in the data memory is accessed. We use the symbol table generated by the compiler to translate global variable names into memory addresses. The type of access (read, write, or both) is configured as a condition for the triggering of events.

Whenever the comparator triggers an event, a data trace packet that contains the comparator **identification (ID)**, the value at the address location in the data memory, and the access type is generated. Optionally, if requested by configuring the debug sub-system, a second data trace packet is generated that contains the current PC. An example trace of generated packets and timestamps is depicted in Figure 3. The data trace packets are then forwarded to the ITM.

The ITM adds additional packets to the output packet stream and forwards the stream to the TPIU, see Figure 2. After every reset of the microcontroller, the ITM adds a synchronization packet. In addition, the ITM adds local timestamp packets containing the current value of the local timestamp counter. To save output channel bandwidth, the ITM only adds a local timestamp packet if the value in the timestamp counter differs from the timestamp generated for the previous data trace packet, or if the timestamp counter reaches its maximum value. The local timestamps are differential, i.e., the timestamp counter is reset to zero whenever a timestamp packet has been generated. The timestamp counter is incremented based on the clock signal configured for the processor of the microcontroller. In addition, a prescaler that is specific for the local timestamp counter can be configured. As a result, for a prescaler larger than 1, events generated by different load or store instructions (at different processor clock cycles) can fall into the same local timestamp epoch.

The TPIU outputs the data as a byte stream via the SWO port, see Figure 2. The data are then received and buffered locally by the debug probe. The observer computer periodically polls the

debug probe and fetches the data from the buffer in the debug probe. For each fetch action, the observer computer takes a global timestamp from the synchronized system time and adds it to the fetched data trace. At the end of the testbed test, the testbed server fetches all raw tracing data files from all observers.

The chain from generating data trace packets in the DWT unit to storing the traced data in a file on the observer computer contains different buffers and additional data can increase the amount of data packets in the stream on the way. Whenever one of the buffers overflows, tracing data is delayed or discarded. To detect such incidents, the ITM can add corresponding information to the tracing stream. The local timestamp packets contain delay markers that indicate that either the timestamp or the data or both have been delayed relative to the actual event triggered in the DWT. In addition, if packets are discarded due to a buffer overflow, the ITM can generate and insert a special overflow packet that consists of a header only. We consider these special situations and omit delayed data when selecting the pairs of timestamps that are used for the regression in Equation (3).

In the following, we list the relevant components of the delay between the occurrence $t[i]$ of an event i and the local timestamp $t_s^{\mathrm{ITM}}[i]$ denoted as $\Delta^{\mathrm{ITM}} = \Delta^{\mathrm{cap}} + \Delta^{\mathrm{local}}$ as well as the delay between the timestamp of an event by the ITM and the timestamping by the observer $\Delta^{\mathrm{TRF}} = \Delta^{\mathrm{MCU}} + \Delta^{\mathrm{SWO}} + \Delta^{\mathrm{FETCH}} + \Delta^{\mathrm{USB}} + \Delta^{\mathrm{GT}}$.

Δ^{cap} The delay for capturing the data trace event in the DWT and generating data trace packet. The actual delay is implementation defined. However, based on the specification of the hardware tracing system [20], we know that the delay for this action is negligible (a few clock cycles at most) compared to the delays in later stages of the trace pipeline.

Δ^{local} The delay for generating the local timestamp packet in the ITM. Again, according to the specifications [20], the delay for this action is negligible compared to the delays in later stages of the trace pipeline.

Δ^{MCU} The delay for adding packets to the ITM output buffer and to forward them to the TPIU output buffer. Again, based on the specifications [20], the delay for this action is negligible compared to the delays in later stages of the trace pipeline in case the delay marker in the corresponding local timestamp does not indicate a delay.

Δ^{SWO} The time for transferring all previous packets up to and including the local timestamp packet from the TPIU to the buffer of the debug probe via SWO. This delay depends on the state of the queues and is directly related to the speed of the SWO connection f_{SWO}. A typical maximum value for the SWO speed is $f_{\mathrm{SWO}} = 4$ MBaud.

Δ^{FETCH} The delay until the observer computer initiates the next fetching action to transfer data from the buffer of the debug probe to the observer computer. This delay is bounded by the configured loop period for fetching data. Feasible loop periods are in the order of 10 ms.

Δ^{USB} The time for transferring the data from the buffer of the debug probe to the observer computer. The variance of this delay is expected to be in the order of milliseconds due to the non-deterministic behavior of the USB protocol implementation.

Δ^{GT} The delay for obtaining a global timestamp on the observer computer. The variance (jitter) of this delay is expected to be in the order of 200 µs, since the timestamp is taken in software in the user space of the operating system.

As mentioned before, an estimated average of the accumulated delays $E(\Delta^{\mathrm{ITM}})$ and $E(\Delta^{\mathrm{TRF}})$ are determined as part of the system characterization (see Section 5.2.1). These values are used to determine the global time of event occurrences according to Equation (2) via the regression in Equation (3).

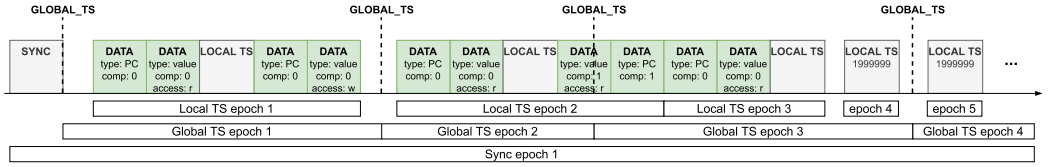

Fig. 3. Example trace of SWO packets with local and global timestamps.

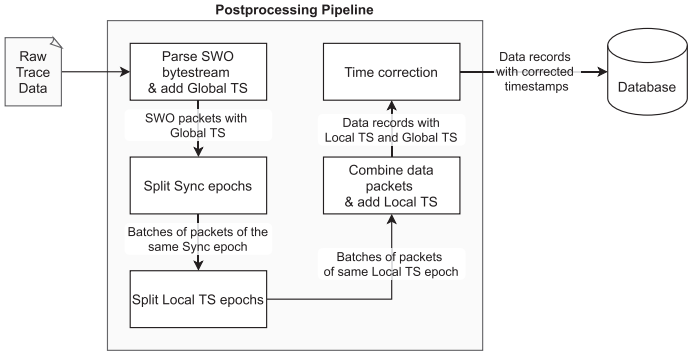

Fig. 4. Pipeline for processing raw tracing data into records of data trace events with time synchronized timestamps on the testbed server.

4.3 Time-Aware Parsing of Raw Tracing Data

After the execution of a test, the raw tracing data of each observer are available on the testbed server. In this section, we explain in detail how these data are parsed and how the global testbed time for each traced event is reconstructed in postprocessing.

The collected raw tracing data of an observer contains a stream of bytes received via the SWO port, interleaved with global timestamps $t^{\mathrm{OBS}}[i]$ derived from synchronized system time on the observer computer. An example is depicted in Figure 3. The SWO byte stream contains packets that consist of a one-byte header and a payload of 0 to 4 bytes. Since the observer computer fetches the available number of bytes from the debug probe buffer irrespective of SWO packet boundaries, the global timestamp values can be interleaved by individual bytes of a packet.

The server uses the multi-stage pipeline depicted in Figure 4 to parse and decode the raw tracing data of each observer. First, the SWO bytes are parsed into SWO packets. Each packet is annotated with the global timestamp corresponding to the first SWO byte from which the packet has been constructed. Then, the stream of packets is split into batches of packets that correspond to the same synchronization packet (`SYNC`), i.e., the period of time (synchronization epoch) between two microcontroller resets. If there was no microcontroller reset, then all packets belong to the same synchronization epoch. Packets in resulting synchronization epochs are further split into batches such that all packets in one batch correspond to the same local timestamp packet (`LOCAL TS`). This means that all data trace packets (`DATA`) in the same local timestamp epoch were triggered while the local timestamp counter in the ITM (see Figure 2) contained the same value. In a next step, different types of data trace packets that were triggered by the same event, e.g., a packet containing data and a packet containing the PC, are combined and the corresponding local timestamp from the local timestamp epoch is added. Each resulting data record contains a global timestamp, a local timestamp, and the traced values.

Table 2. Example Output from the Data Trace Service on FlockLab 2

Timestamp	Target ID	Variable	Value	Access	PC	Delayed
1602661351.1832957	11	tx_len	0	r	0×800bbfc	no
1602661351.1833105	11	tx_len	0	r	0×800b8ea	yes
1602661351.1833405	11	tx_len	1	w	0×800b8fa	yes
1602661351.1834369	4	tx_len	0	r	0×800bbfc	no
1602661351.1834520	4	tx_len	0	r	0×800b8ea	yes
1602661351.1834820	4	tx_len	1	w	0×800b8fa	yes
1602661366.1017075	4	tx_len	1	r	0×800be6c	no
1602661366.1017935	11	tx_len	1	r	0×800be6c	no
1602661366.1824489	11	tx_len	1	r	0×800be6c	no
1602661366.1825450	4	tx_len	1	r	0×800be6c	no
1602661366.1826136	11	tx_len	1	r	0×800bbfc	no
1602661366.1826262	11	tx_len	1	r	0×800b8ea	yes
1602661366.1826563	11	tx_len	2	w	0×800b8fa	yes
1602661366.1827123	4	tx_len	1	r	0×800bbfc	no

Timestamps are synchronized using the method described in Section 4.1.

In the final step, the time correction is applied to all data records, see Section 4.1. For this, high-quality timestamps, i.e., timestamps from local timestamp packets where the delay marker does not indicate a delay, are used to determine the regression parameters using Equation (3). Since the global timestamps taken on the observer $t^{\mathrm{OBS}}[i]$ contain significant outliers due to the non-deterministic delays Δ^{TRF}, we use two stages for the regression according to Equation (3). After a first regression, we remove all data points that deviate from the fitted line by more than 2 times the resulting standard-deviation. With the remaining synchronization pairs $(t_s^{\mathrm{ITM}}[i], t^{\mathrm{OBS}}[i])$, we perform another least-squares fit according to Equation (3). According to Equation (2), we use the resulting regression parameters to map the local timestamps $t_s^{\mathrm{ITM}}[i]$ to the synchronized global time $t[i]$ of the events i.

In the very last step, the comparator IDs are mapped back to the name of the variable for which the tracing was originally requested.

When the comparators in the DWT unit do not trigger the generation of data trace packets before the local timestamp counter reaches its maximum value (1999999 in the example in Figure 3) the ITM outputs a local timestamp packet with the maximum value. These packets form their own local timestamp epoch without data trace packets. This provides time synchronization points for the regression even for time periods where there are no data trace packets generated.

4.4 Potential of the Proposed Tracing System

In this section, we describe the resulting tracing data, highlight use cases such as validation using a simulation, and discuss tradeoffs and limitations of the proposed tracing system.

4.4.1 Resulting Trace Data. The resulting tracing data from a test execution with the proposed tracing system contains one record for every memory access of the chosen variables and access types configured on each observer. As in the example depicted in Table 2, the tracing data contain the corrected timestamp according to Equation (2), ID of the target that generated the event, the variable name, the value in the data memory after the access, the access mode (read (r) or write (w)), the PC (only if enabled) at the time of the memory access, and the delay marker. The delay marker is contained in the local timestamps and indicates whether a delay occurred when adding the data

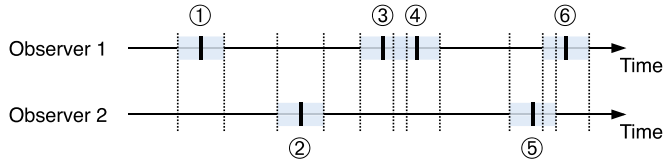

Fig. 5. The ordering of events for the same observer (e.g., events 3 and 4) is guaranteed by the on-chip debug and trace hardware, but statements about the ordering of events of different observers (e.g., events 5 and 6) are limited by the time synchronization accuracy of the data trace service.

trace or local timestamp packets to the output buffer inside the debug and trace sub-system on the microcontroller, as discussed in Section 4.3.

4.4.2 Use Cases. The obtained tracing data can be used to visualize and manually analyze the evolution of state changes of the distributed system over time. Examples of such visualizations are provided in Section 6.4. The authors of this article provide a visualization tool for data collected with FlockLab 2 as open source.[1]

The tracing data also allow the systematic and automated validation of the behavior of the software running on a set of distributed wireless target nodes. The basic concept is to implement the distributed system as a simulation (e.g., Reference [25]) or as timed automata (e.g., Reference [12]) and also as software running on the target nodes on the testbed. The output of the simulation or timed automata and output of the testbed can then be analyzed and compared. Related work has shown different approaches to perform such validations. Examples are conformance tests [23, 34], unit tests [11, 21], or distributed assertions [24, 27].

The proposed tracing system provides an out-of-band infrastructure to time synchronize the traced events. However, it is possible that the system under test implements a reliable time synchronization with even better temporal accuracy, e.g., based on Glossy [6]. For this case, it is possible to combine the more accurate time synchronization of the wireless protocol with the non-intrusive distributed tracing. The only requirement would be memory accesses at the critical time instants, namely at the synchronization reference points and when an event is to be logged. By using the tracing data, the offset of the event occurrence relative to the synchronization reference points can be determined with high accuracy as shown in Section 5.2.2. In many cases, memory accesses are required at the corresponding locations in the target code anyway and are therefore already present.

In addition, the tracing system proposed in this article allows to trace the program counter (PC) values, i.e., the instruction memory addresses at which the traced memory accesses are generated. One possibility to leverage this information is to reconstruct a partial program flow trace.

There is extensive previous work available that investigates methods to assess the chronological ordering of observed events. For example, Römer et al. [24] introduce flags to indicate whether the ordering can be determined and Hahner et al. [9] use a graph to determine the temporal order of events.

Detailed investigations of methods that make use of the traced data are not the focus of this article, as extensive related work that can be applied to analyze and visualize the timed traces from the distributed target nodes is available.

4.4.3 Limitations. Because the proposed tracing system is based on on-chip debug hardware, it is non-intrusive and efficient, but at the same time it has some limitations that need to be considered when using the testbed infrastructure. In the following, we list and discuss some noteworthy limitations of the proposed distributed tracing system.

[1] https://flocklab.ethz.ch.

Timestamping accuracy and event ordering: In general, total ordering with distributed independent loggers and non-zero time synchronization accuracy is not possible. The amount of events that can be ordered depends on the time synchronization accuracy and the time distance between event occurrences. Figure 5 depicts an example with two observers with data trace events (black bars) and their temporal uncertainties (transparent boxes mark the upper and lower bound of the timestamp). The uncertainty regions directly relate to the range of $\epsilon[i]$ from Equation (4). The ordering of events for the same observer is guaranteed by the on-chip debug and trace hardware. In the example this means that the ordering of events 3 and 4 is unambiguous even though the uncertainty regions overlap. The ordering of events of different observers is limited by the time synchronization accuracy of the tracing system, see Equation (4). The ordering is unambiguous if the events are further apart than twice the upper bound of the uncertainty of a single timestamp. In the example, this means that the ordering of events 1 and 2 can be clearly determined, whereas no statement on the ordering of events 5 and 6 can be made. By using an upper bound of the temporal error based on the characterization in Section 5.2.1 as well as the timestamps and target IDs from the output of the data trace service, a user can identify pairs of timestamps on different targets without sufficient spacing for which the ordering is ambiguous.

The use of additional hardware, such as an FPGA or a dedicated microcontroller, for timestamping the SWO output could potentially improve the timestamping accuracy as the delays Δ^{FETCH} and Δ^{USB} could be eliminated and the relatively large variance of Δ^{GT} could potentially be reduced. We did not investigate such an improvement as the obtained timestamping accuracy is sufficiently high as shown in Section 5.2.1 and Section 5.2.2 and additional hardware would increase the observer complexity. However, it could be an interesting approach for future work.

Maximum Event Rate: The maximum supported frequency with which a variable can be accessed such that the data trace service can still trace the accesses without issues, i.e., without delay markers indicating a delay or overflow packets being generated, is limited by the following factors:

- The maximum supported event repetition rate of the event triggering hardware inside the DWT unit.
- The size of the buffers in the debug and trace sub-system and the debug probe.
- The method and data transfer protocol that is used to offload the tracing data from the target to an external device (debug probe and observer computer).
- The amount of data that are generated by every event.

For example, enabling the tracing of PC values increases the number of data trace packets that are generated. Configuring the service to only trace write accesses instead of read and write accesses reduces the number of packets in general.

Amount of state that can be monitored: The number of variables/memory addresses that can be monitored is limited by the number of available comparators in the on-chip debugging hardware. The feature set of the debug sub-system is determined by the manufacturer of the microcontroller and cannot be extended. An overview of a selection of microcontrollers and **systems on a chip (SoCs)** with the number of comparators is provided in Table 1.

Complete sequence of local timestamp packets: The timestamps in local timestamp packets are differential to save tracing bandwidth. As a consequence, the timing is only correct if all local timestamp packets can be successfully forwarded to the observer computer. In case a buffer inside the on-chip debug and trace sub-system is full, an overflow of the buffer can occur and local timestamp packets may be discarded. As a result the synchronized timestamps cannot be reconstructed

correctly in postprocessing. However, such incidents can be detected based on the overflow packets in the SWO byte stream. System resets, however, do not lead to problems with reconstructing the synchronized timestamps, since synchronization packets indicate a restart of the SWO stream.

Power profiling and low-power modes: The proposed non-intrusive distributed tracing system focuses on the tracing and validation of the functional behavior of the system under test. Testbeds usually feature separate services dedicated to perform detailed power profiling. For example the FlockLab 2 testbed [32] supports accurate high-dynamic range power profiling to accurately measure the power consumption during radio transmissions as well as during low-power modes. It is possible to use both services, the distributed data tracing and the power profiling, simultaneously. However, in this case the significance of the power measurement is limited, since the power consumption of the debug and trace sub-system is also measured, as it cannot be isolated and thus supplied with power separately. In the case of a pure power measurement, the debug and trace sub-system can be switched off and therefore does not influence the measurement.

Modern microcontrollers feature many different power modes and allow to turn on/off separate components individually. The available options of course depend on the actual implementation. Usually, the debug and trace sub-system requires a high-frequency clock source. For ultra-low-power modes, usually only low-frequency clock sources are enabled. By default, a microcontroller with data trace output enabled enters a low-power mode with a low-frequency clock source when instructed to do so but does not disable the high-frequency clock source as it is still required by the debug and trace sub-system. As a consequence, the power consumption is increased but the functional behavior, including timing, is very close to the behavior without debug output enabled. Certain microcontroller implementations support the automated disabling of the debug and trace sub-system in case a low-power mode with low-frequency clock source is entered. However, the ARM CoreSight architecture does not specify this mode of operation and some microcontroller implementations do no generate synchronization packets upon wake-up from low-power mode. As a consequence, the proposed distributed tracing system does not support time correction of such traces.

5 IMPLEMENTATION

To characterize the performance and to show the use of the system for validation of wireless IoT devices, we implemented the proposed tracing system for STM32L4- (STMicroelectronics) based targets as part of the FlockLab 2 testbed [32]. The implementation is open-source and the FlockLab 2 testbed with the proposed tracing service is publicly accessible.[2]

5.1 Implemented System

In this section, we briefly describe the specifications of the implemented system consisting of targets and the testbed infrastructure.

5.1.1 Target. We use the DPP2 LoRa target node architecture [32] that consists of the DPP2 LoRa communication board [4] and a simple target adapter that connects the pins of the communication board to the FlockLab 2 target interface. The DPP2 LoRa communication board consists of an STMicroelectronics STM32L433CC microcontroller and a Semtech SX1262 low-power long-range radio transceiver. The microcontroller features an ARM Cortex-M4 core that supports a clock frequency of up to 80 MHz. For this work, we use the external crystal as a clock source and run the microcontroller at 48 MHz. The radio transceiver provides up to +22 dBm transmit power, operates in the 868-MHz band, and supports both FSK and LoRa modulations.

[2] https://flocklab.ethz.ch.

Table 3. Specification of the Example Implementation of the Proposed Tracing System Consisting of an STM32L433 Microcontroller and the FlockLab 2 Testbed Infrastructure

Component	Value
Number of DWT comparators	4
ITM output buffer size	10 Bytes
TPIU output buffer size	128 Bytes
Max SWO transfer speed	4 MBaud
Debug probe buffer size	4 MB

In the ARM Cortex-M4 architecture, the IP cores for supporting CoreSight debug and trace functionalities are optional. The STM32L433 microcontroller implements a combined SWD and JTAG debug port and integrates the DWT, ITM, and TPIU blocks, see also Section 3.1 and Figure 2. The key specifications are listed in Table 3. The DWT contains 4 comparators that allow the parallel tracing of up to four variables or data memory addresses, respectively. The TPIU provides the SWO output port with a maximum speed of 4 MBaud.

5.1.2 Testbed Infrastructure. The implemented example system is part of the publicly available FlockLab 2 [32] testbed. The observers are equipped with a Segger J-Link OB (on-board) debug probe that features a 4-MB buffer. A BeagleBone Green single-board computer running a Linux operating system serves as the observer computer. It interfaces with the debug probe via USB. The time synchronization of the BeagleBone observer computer is based on the PPS signal generated by a u-blox M8 GNSS receiver, providing an accuracy of 60 ns according to the datasheet. Observers without GNSS signal reception are time synchronized with PTP through the BeagleBone's Ethernet interface to connect to another BeagleBone (PTP master) with a GNSS receiver. The `chrony` software is used to adjust the BeagleBone's system time based on the PPS signal or the PTP protocol. `chrony` statistics data from 9 PTP-synchronized observers collected over 24 hours indicate that the system time deviation is below 1 µs for 98.9% of the time, while the maximum logged deviation is 14 µs. On the observer computer, the J-Link library from Segger and the Python library `pylink` are used to interact with the debug probe. A Python script on the observer computer is used to configure the microcontroller on the target as well as to periodically check the buffer on the debug probe and to log available tracing data together with the global timestamps derived from the observer computer's system time to a file. The implemented system uses a poll period of 10 ms and a local timestamp prescaler value of 16. The chosen values represent a tradeoff between minimizing the delay jitter for forwarding the traced data and keeping the computational overhead on the observer computer low. For the SWO connection between the microcontroller and the debug probe, we use the maximum supported SWO speed of 4 MBaud.

On the testbed server, a set of Python scripts and libraries are used to schedule a test, distribute configuration information to the different observers, fetch tracing data collected by the observers, and postprocess the tracing data as described in Section 4.3.

5.2 Characterization

In this section, we investigate and measure the performance of the implemented tracing system. Important aspects that do not directly follow from the specifications of the used system components are the accuracy of the corrected timestamps and the maximum supported event rate.

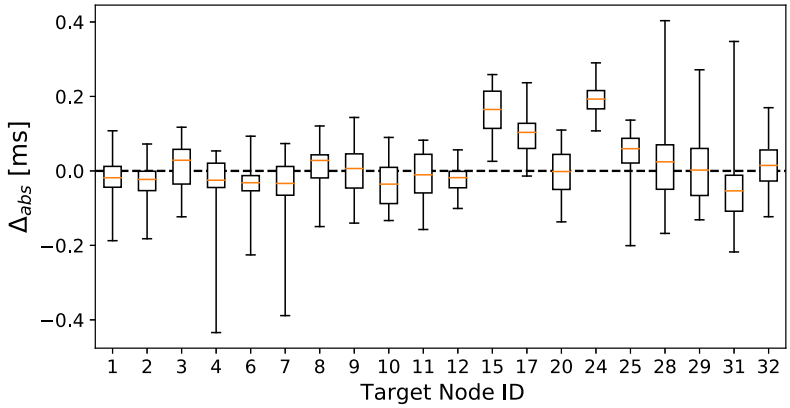

Fig. 6. Distribution of time difference between timestamps from implemented data trace systems and the logic tracing of the FlockLab 2 testbed. For each target, the test contains more than 40,000 events with random intervals between 20 µs and 71,980 µs distributed over 24 minutes. The box extends from the lower to the upper quartile values of the time difference values, with a line at the median. The whiskers indicate the minimum and the maximum of the time differences. The IDs of the target nodes are not sequential, because not all node IDs exist in the FlockLab 2 testbed and we only use observers with GNSS time synchronization for this measurement. The results show that the implemented system achieves an absolute time accuracy below 0.45 ms.

5.2.1 Absolute Time Synchronization Accuracy. To measure the absolute time accuracy of the implemented tracing system, we use the logic tracing service available on FlockLab 2 to obtain the ground truth. The logic tracing provides a high temporal accuracy of typically below 0.25 µs [32]. Logic tracing requires code instrumentation that is not a problem for this analysis, since the microcontroller does not perform time-critical tasks. We run a test program on the DPP2 LoRa target to generate events. Such an event consists of toggling a GPIO pin directly followed by writing to a global variable that is traced using the data trace service. For this test, we solely use FlockLab 2 observers that are synchronized using GNSS, since this provides the necessary accuracy and stability.

We perform the test on 20 targets with 20 observers. On each target, more than 40,000 events are generated with a randomized interval between 20 µs and 71,980 µs and distributed over a test duration of 24 minutes. For every event i, we calculate the difference of the timestamps provided by the two tracing services as $\Delta_{\text{abs}}[i] = t_{\text{datatrace}}[i] - t_{\text{gpio}}[i]$ where $t_{\text{datatrace}}[i]$ is derived from data trace data and therefore equal to $t[i]$ in Equation (4) in Section 4.1.

Figure 6 shows the resulting distribution of the differences for each target. These measurements allow us to estimate the magnitude of the error $\epsilon[i]$ of the time correction in Equation (4) in Section 4.1. The maximum offsets of different target nodes are within ±0.45 ms. The results of this test indicate that the implemented system achieves a time accuracy below 0.45 ms, which is significantly lower than typical packet on-air-times and satisfies our system requirements in Section 1.

5.2.2 Relative Time Accuracy. In this subsection, we measure the relative time accuracy of the implemented tracing system, i.e., the accuracy of measuring time intervals. This allows to compare the relative time accuracy of synchronized global timestamps to the unsynchronized local timestamps. As in Section 5.2.1, we use a test program on the DPP2 LoRa target to generate events and we use the logic tracing service available on FlockLab 2 to obtain the ground truth. Again, an event consists of toggling a GPIO pin directly followed by writing to a global variable that is traced using the data trace service.

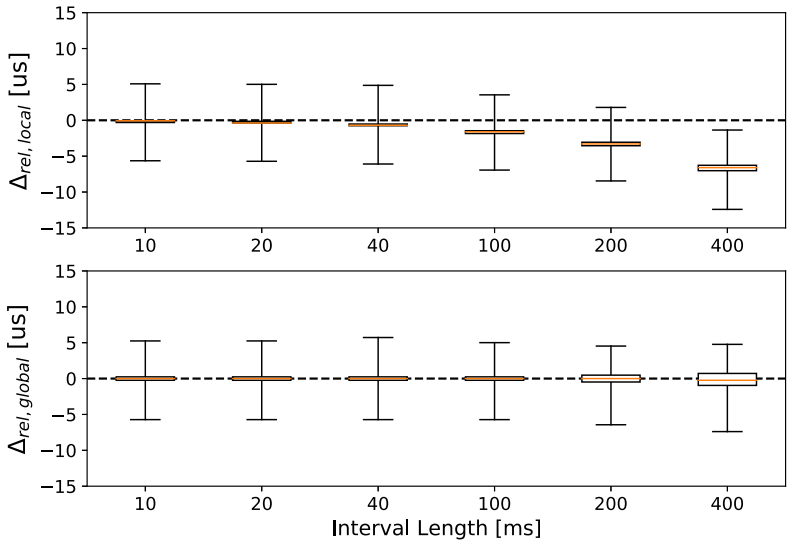

Fig. 7. Deviation of time intervals measured with local and global timestamps from data tracing compared to measurements from the logic tracing service of the FlockLab 2 testbed. The distribution for each interval length contains 50,000 samples. The box extends from the lower to the upper quartile values of the time deviation values, with a line at the median. The whiskers indicate the minimum and the maximum of the time deviation. The results show that the relative time accuracy when using the synchronized global timestamps instead of local timestamps is comparable for short intervals and significantly better for long intervals.

On the target, 60,000 events are generated with a fixed interval of 10 ms distributed over a test duration of 10 minutes. The events are used multiple times to form 50,000 different intervals with a length between 10 ms and 400 ms. For each pair of events (event i and $i+k$, with k corresponding to the interval length), we determine the interval I based on three different sources:

- Timestamps from the GPIO events from logic tracing (ground truth): $I_{\text{gpio}}[i,k] = t_{\text{gpio}}[i+k] - t_{\text{gpio}}[i]$.
- Unsynchronized local timestamps from data tracing: $I_{\text{local}}[i,k] = t_s^{\text{ITM}}[i+k] \cdot m - t_s^{\text{ITM}}[i] \cdot m$ As in Section 4.1, timestamp $t_s^{\text{ITM}}[i]$ is derived from the local clock on target s. Here, factor m is not obtained from regression but directly determined from the configuration (microcontroller clock frequency and prescaler of local timestamp counter) and is used to convert timer ticks to seconds.
- Synchronized global timestamps from data tracing: $I_{\text{global}}[i,k] = t_{\text{datatrace}}[i+k] - t_{\text{datatrace}}[i]$ Again, timestamp $t_{\text{datatrace}}[i]$ is equal to $t[i]$ in Equation (4) in Section 4.1.

Then the deviations of the interval based on the local timestamps $\Delta_{\text{rel,local}}[i,k] = I_{\text{local}}[i,k] - I_{\text{gpio}}[i,k]$ and the global timestamps $\Delta_{\text{rel,global}}[i,k] = I_{\text{global}}[i,k] - I_{\text{gpio}}[i,k]$ are calculated.

Figure 7 shows the resulting distribution of the deviations. For this experiment, we show the data from a single observer. We performed the same experiment on multiple GNSS synchronized observers and obtained comparable results. The results show that the relative time accuracy when using the synchronized global timestamps instead of local timestamps is comparable for short intervals but significantly better for long intervals. The measurements of the interval using the local timestamps deviate by 16.5 ppm (median). This can be explained by the fact that local timestamps are not corrected for clock drift. As a consequence, the absolute value of the deviation of local timestamps increases linearly with increasing interval length.

Table 4. Results of the Measurements to Determine the Maximum Supported Continuous Event Rate

Configured period [µs]	10	11	12	13	14	15	16	17	18	19	20
Actual average period (logic tracing) (µs)	10.3	11.3	12.3	13.3	14.3	15.2	16.2	17.2	18.2	19.2	20.2
Event rate (kHz)	97.0	88.5	81.4	75.3	70.1	65.6	61.6	58.1	54.9	52.1	49.6
Actual bytes per period (average)	4.09	4.52	4.86	5.31	5.71	6.10	6.49	6.73	7.00	7.00	7.00
Resulting SWO data rate (Mbit/s)	3.17	3.20	3.16	3.20	3.20	3.20	3.20	3.13	3.08	2.92	2.78
Buffer overflow indicated	yes	yes	yes	no	no	no	no	no	no	no	no
Delay indicated	yes	yes	yes	yes	yes	yes	yes	yes	no	no	no

The implemented system allows continuous tracing of periodic events with event rates up to 54.9 kHz.

5.2.3 Maximum Event Rate. As discussed in Section 4.4.3, different factors can limit the maximum event rate supported by the data tracing system. In this section, we experimentally determine the limit for the implementation of the tracing system in FlockLab 2 with the DPP2 LoRa target. We differentiate two corner cases for the maximum supported event rate: (1) continuous generation of periodic events and (2) a burst of consecutive events without delay between individual events.

Continuous Periodic Events: For measuring the maximum event rate for continuous generation of periodic events, we use the same program for the target as in Section 5.2.1 but provide a set of fixed periods in the range from 10 to 20 µs to generate the events. For each period, we generate 10,000 events where each event consists of writing a single variable into memory. Data tracing is configured not to trace the PC. With logic tracing, we measure the actual period between events. From the obtained data trace data, i.e., the SWO packets, we count the number of generated SWO bytes and determine whether a delay or overflow was indicated. The results for one execution on a single observer are listed in Table 4. The test was repeated multiple times on different observers and the result did not differ significantly.

The results show that buffer overflows occur for event rates of 81.4 kHz and higher (period below 13 µs). For event rates between 58.1 kHz and 75.3 kHz, no buffer overflows occur but delays are indicated. This means that all tracing data could successfully be forwarded but timing information might not be sufficiently accurate. For event rates of 54.9 kHz and lower, continuous data tracing works without issues. In this case, 7 bytes per event are generated. For every event, a full data trace packet (5 bytes) and a medium-sized local timestamp packet (2 bytes) are required. For the cases where no overflows but delays are present, the number of bytes per event is lower. This can be explained with delayed local timestamp packets, which lead to cases where multiple data trace packets share the same timestamp packet. The SWO data rate appears to be limited to 3.2 Mbit/s. This represents 80% of the configured SWO baud rate of 4 MBaud. This is consistent with the overhead of a UART connection with one start, eight data, and one stop bit. Therefore, it seems likely that the SWO connection represents the bottleneck for the maximum event rate in case of continuous periodic events.

Burst of Events: To measure the maximum event rate in case of a burst of events, we use a program that performs a specified number of consecutive write accesses. No explicit delay between write accesses is used, the delay is caused solely by the time it takes the microcontroller to execute the memory write instruction. The number of events within each burst is varied between 1 and 8. The data trace service is configured not to trace the PC. Again, the test was repeated multiple times on different observers and the results did not differ significantly.

The results, listed in Table 5, suggest that up to three consecutive events are supported without issues. Four consecutive events are supported but delays occur when packets are forwarded inside the debug and trace sub-system. For Five and more events, buffer overflows occurred and data are missing in the tracing data. The number of SWO bytes per burst in case of buffer overflows is not constant, which can be explained by a slightly varying timing behavior of different runs as the

Table 5. Results of the Measurements to Determine the Maximum Supported Burst Event Rate

Number of consecutive events	1	2	3	4	5	6	7	8
Data packets	1	2	3	4	4	4	4	4
Local TS packets	1	1	1	3	3	3	2	2
Number SWO bytes	9	14	19	27	28	28	27	27
Buffer overflow indicated	no	no	no	no	yes	yes	yes	yes
Delay indicated	no	no	no	yes	yes	yes	yes	yes

The implemented tracing system allows to trace up to 3 consecutive events in a burst without delay between the events.

interval between events has an influence on the length of the local timestamp packet size. It can be estimated that the buffer that represents the bottleneck for the maximum event rate in case of bursts has a size of around 28 bytes.

6 CASE STUDY

We demonstrate the use of the proposed non-intrusive tracing system by applying it to an implementation of the eLWB networking protocol [28] that is based on synchronous transmissions. In the first part (Section 6.3), we show the advantages of non-intrusive hardware assisted tracing by comparing the impact of the data tracing to traditional tracing via the serial interface. In the second part (Section 6.4), we demonstrate how our tracing system can be used for validating the eLWB networking protocol running on a set of distributed wireless IoT devices. Before presenting the two parts of the case study, we briefly discuss the relevant aspects of the eLWB protocol and the used system setup.

6.1 Data Collection with eLWB

In this section, we describe the scenario used for the case study and we discuss the used networking protocol layers.

6.1.1 Data Collection Scenario and Realization with the FlockLab 2 Testbed. The scenario comprises a set of battery-powered wireless sensor nodes and a single sink node. In contrast to the sensor nodes, the sink node has access to an unrestricted power supply (e.g., connected to mains or supported by energy harvesting). The sensor nodes periodically sense a physical property of the environment (e.g., temperature, acoustic or seismic emissions, etc.) and determine whether the measurement is relevant, i.e., whether an event has been detected. Multi-hop wireless transmissions are used to transfer the detected events from the sensor nodes to the sink node, as not all sensor nodes have a direct link to the sink node. The sink node forwards the data to a database via an Internet connection. The goal of the system is to transfer as many of the detected events to the database as possible while minimizing energy consumption on the sensor nodes such that the system's lifetime is maximized. As a secondary objective, the latency of transferring the events should be minimized.

To realize the scenario with the FlockLab 2 testbed (see Section 5.1.2), we use 20 DPP2 LoRa targets as used and described in Section 5.1.1. One of the targets (target 2) represents the sink, all other targets represent sensor nodes. Event value generation is emulated by obtaining a 2 byte value from a random number generator. Following each eLWB communication round, each node generates a random number of events that is uniformly distributed in the interval $[0, 2]$. Based on the FlockLab 2 topology, the network topology consists of 17 nodes located in an office building

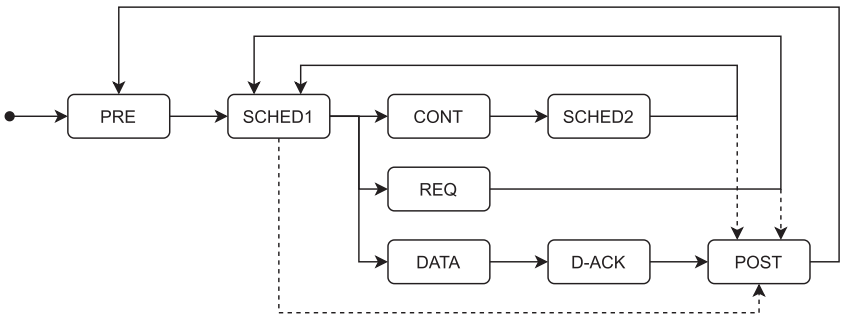

Fig. 8. The state diagram of the eLWB implementation used for the case study. Solid lines represent normal state transitions, dashed lines correspond to transitions where nodes did not successfully receive a message or do not want to further participate in the eLWB round.

and 3 outdoor rooftop nodes. The sink is one of the nodes inside the office building. As a result, the link distance is inhomogeneous and in the range of 4 m to 2 km. The time synchronization of the wireless protocol is considered as part of the system under test and is therefore not used to timestamp the traced events. Instead, the out-of-band time synchronization provided by the FlockLab 2 testbed infrastructure is used.

6.1.2 eLWB. For the case study, we use eLWB [28], which represents a state-of-the-art wireless networking protocol designed for the energy-efficient data collection of event-based sensor data at a central sink. With eLWB, the sink organizes the data transfers in a **time-division multiple access (TDMA)** fashion. The state diagram of the eLWB protocol implementation used for the case study is depicted in Figure 8. Due to the (TDMA)-based coordinated channel accesses, all participating nodes are supposed to be in the same state at the same time.

The communication is organized in communication rounds. In every eLWB round, the nodes prepare data to send in the `PRE` state and process potentially received information in the `POST` state. Each round consists of one or more sub-rounds that have a varying duration. Each sub-round consists of slots and is initiated by a schedule flood sent by the sink (`SCHED1` state). A single Gloria flood (see Section 6.1.3) is used as the underlying communication primitive to send and/or receive a message in a slot. All nodes that receive the schedule flood follow the sequence of states as defined in the schedule sent by the sink. An eLWB round starts with a contention sub-round. During the contention slot (`CONT` state), the sink listens and sensor nodes can indicate their demand to transfer data. Nodes that are not yet registered use the message sent during the contention slot to register themselves. In our example, we circumvent this bootstrapping mechanism and use a static registration for all sensor nodes. If the sink receives a transmission during the contention slot, then it announces the request sub-round in a second schedule flood (`SCHED2` state). In the request sub-round, a slot is exclusively assigned to each registered sensor node (`REQ` state). In every request slot, the corresponding sensor node can transfer its communication demand for the current eLWB round. In the following data sub-round, the sink node announces the data slots and the assigned sensor nodes (`DATA` state). Then, the sensor nodes use the assigned slots to transfer the data. Following the data sub-round, the sink confirms the receipt of the messages by sending an acknowledge flood (`D-ACK` state). If the sink does not receive any message during the contention slot or no request during the request sub-round, then the following sub-rounds are skipped (dashed arrows in Figure 8). Analogously, the sensor nodes skip sub-rounds if they do not receive schedules from the sink node.

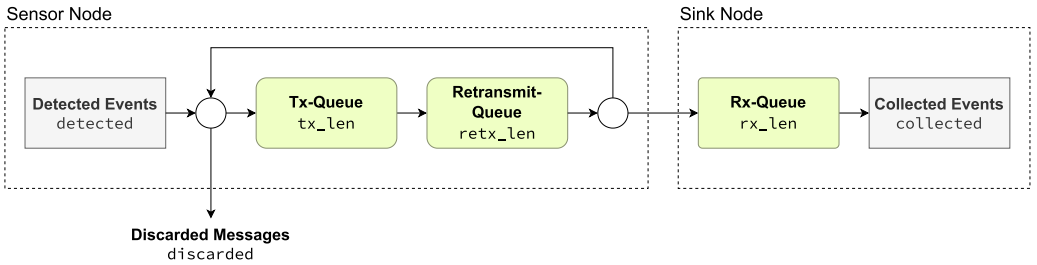

Fig. 9. The flow of messages between event generation on the sensor node and collection on the sink node as implemented by the used eLWB data collection protocol variant.

The message flow from the sensor node to the collection on the sink node is depicted in Figure 9. Each sensor node has its own Tx- and Retransmit-queues. Data that need to be forwarded to the sink is first added to the Tx-queue. When the data have been transmitted in a data slot, the data are moved from the Tx-queue to the Retransmit-queue. Based on the acknowledgement information received during the `D-ACK` state, the sensor node checks whether the data successfully arrived at the sink node. If this is the case, then it removes the corresponding packets from the Retransmit-queue. Otherwise, it moves the packets from the Retransmit-queue back to the transmit queue. The sink node has an Rx-queue that temporarily stores all received messages before they are processed and forwarded.

For the case study, we set the eLWB a round period to 10 s and the maximum number of data slots in one round to 20 slots.

6.1.3 Gloria Floods. The eLWB communication protocol requires a lower-layer communication primitive that allows flooding messages to all nodes in the network. For this, we use Gloria, an implementation of Glossy [6] ported to the DPP2 LoRa platform. Glossy allows to flood messages over multiple hops based on the concept of synchronous transmissions. In Glossy, a node alternately receives and sends a packet within a flood. The time to re-transmit a packet is determined by the arrival time of the immediately preceding received packet. In contrast to Glossy, in Gloria the timing of packet re-transmissions within a flood is based on timer events rather than the directly preceding reception of a packet. As a consequence, a node successfully receives a packet within a Gloria flood at most once and retransmits packets multiple times back-to-back, similar to BlueFlood [3]. For the main part of the case study, we use the LoRa modulation with spreading factor SF5 and send a maximum of 7 transmissions within a Gloria flood. This leads to a data message duration of 13.1 ms and a flood duration of 138.4 ms.

6.2 Workflow of the Data Trace Service on FlockLab 2

Along with other configurations, the experimenter uses an XML-formatted test configuration file to specify the variables and the corresponding access modes that shall be traced on each observer of the testbed. The mapping of variables to addresses is performed automatically by the testbed using the symbol table provided in the program image for the microcontroller. Alternatively, it is possible to directly specify an address of the data memory that shall be monitored. The interface for the experimenter is interaction-free during test execution that facilitates repeatability.

The testbed executes the test according to the configuration specified in the XML file, i.e., during the test it collects the data trace output and processes it at the end of the test as described in Section 4.3. After the completion of the test, the experimenter can download the resulting tracing data or inspect a web-browser-based visualization of the collected tracing data.

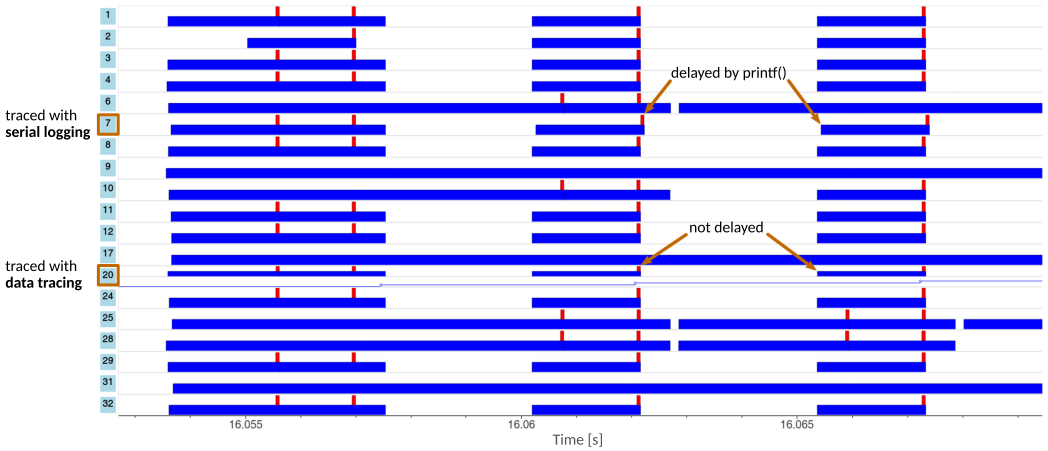

Fig. 10. Logic tracing of a single Gloria flood. Short (red) pulses are interrupts from the SX1262 radio chip, long (blue) pulses are Rx and Tx operations indicated by GPIO events on the microcontroller. Serial logging used on node 7 is intrusive and delays the radio operation by 70 µs while data tracing used on node 20 is non-intrusive.

6.3 Non-Intrusive Tracing of a Time-Critical Synchronous Transmission Protocol

In the first part of the case study, we investigate the impact of serial logging compared to non-intrusive tracing using the data trace service. For this purpose, we log a variable in the time-critical part of the Gloria protocol implementation. The Gloria layer is responsible for the correct timing of transmit and receive radio operations such that transmissions of different nodes occur simultaneously with a precision of <1 µs [6]. If the execution on a subset of nodes is delayed, then the transmissions of different nodes are no longer synchronous and the performance of the communication protocol potentially decreases. For this reason, non-intrusive tracing is essential in the context of time-critical distributed communication systems.

We instrument the code of one sensor node (target 7) to print the current slot index (`slot_index` variable) right before the transmission is initiated. On a second node (target 20), we use the data trace service to trace the `slot_index` variable. To obtain a clearer visualization, we used an FSK modulation with a bitrate of 125 kbit/s and a packet duration of 2.0 ms instead of the LoRa modulation for this experiment.

In Figure 10, a visualization of the corresponding logic and data tracing is depicted. As expected, the transmissions of node 7 with `printf` code instrumentation are delayed. The delay is around 70 µs, as we use DMA transfer for serial printing. If blocking serial printing had been used, then the delay would have been even larger (around 1.060 µs for 44 characters at 460,800 Baud). In both cases, the delay would strongly impact the synchronous activities and the performance of the communication scheme would decrease. With the presented non-intrusive data tracing, the time-critical execution of the synchronous transmissions is unaffected. In addition, serial logging only provides a snapshot of the variable at a certain point in the code. Data tracing, however, allows us to capture all updates of the variable as long as the event rate remains below the supported limit (see Section 5.2.3).

In many cases, a large portion of the time used for developing a program for a system of distributed wireless IoT devices is dedicated to timing-critical low-level details such as to correctly handle interrupts corresponding to external or internal events, see for example Reference [22]. The exact timing behavior changes when code instrumentation is added. This means that for

Table 6. Overview of eLWB Variables That Are Traced in the Case Study Using the Data Trace Service of FlockLab 2

Variable	Description	Sensor Node	Sink Node
`elwb_state`	eLWB protocol state	✓	✓
`detected`	Detected data value	✓	
`tx_len`	#Msgs in Tx-queue	✓	
`retx_len`	#Msgs in Retransmit-queue	✓	
`rx_len`	#Msgs in Rx-queue		✓
`collected`	Collected data value		✓

the implementation of time-critical parts either no code instrumentation can be used or the time-critical parts need to be re-adjusted after the removal of the code instrumentation used to verify the correct behavior. Since our proposed solution for distributed tracing enables the monitoring of the internal state without influencing the execution behavior, it can be used even for time-critical parts of the code.

6.4 Tracing Distributed State of the eLWB Networking Protocol

In the second part of the case study, we demonstrate how the distributed state of the eLWB protocol can be traced. By visualizing the distributed state of the network and by applying sanity checks, it is possible to detect faulty and verify correct behavior.

The goal of the validation is to ensure that the implementation of the protocol behaves as intended. Since eLWB involves distributed state, the validation requires detailed observations of all nodes. To compare state transitions of different nodes, the observations need to be tightly time synchronized. Furthermore, the observations of the distributed state must not influence the execution behavior to obtain meaningful experimental results that can be applied to a future deployed system. All the mentioned requirements (also see Section 1) are satisfied by the tracing system that we propose in this article.

Concretely, we want to verify the following three aspects of the eLWB implementation:

(A1) All participating sensor nodes (nodes that received the previous eLWB schedule sent by the sink node) are in an allowed eLWB state (see Section 6.1.2). There are no lockups or illegal state transitions (e.g., `SCHED1` → `SCHED2`).
(A2) All events from the sensor nodes are correctly forwarded through the message queues on the sensor and sink nodes. Exceptions are the messages that need to be discarded due to queue overflows on the sensor node.
(A3) Events arriving on the sink node do not differ from the events generated on the sensor nodes.

For the validation, we trace the variables listed in Table 6 on the sensor node and the sink node, respectively. The protocol state (`elwb_state`) is traced on all 20 nodes and is used to verify that all sensor nodes are in the same eLWB protocol state as the sink node (aspect A1). An exemplary visualization of one eLWB round used to verify proper state transitions is depicted in Figure 11. Most sensor nodes follow the schedule of the sink node and are therefore in the same state as the sink node. In the example, sensor node 15 does not receive the second schedule and transitions from `SCHED2` directly to `POST`).

The variables `detected` and `collected` are used to verify that the events have been transferred unmodified (aspect A3). The sequence of values from events on the sensor nodes (`detected`)

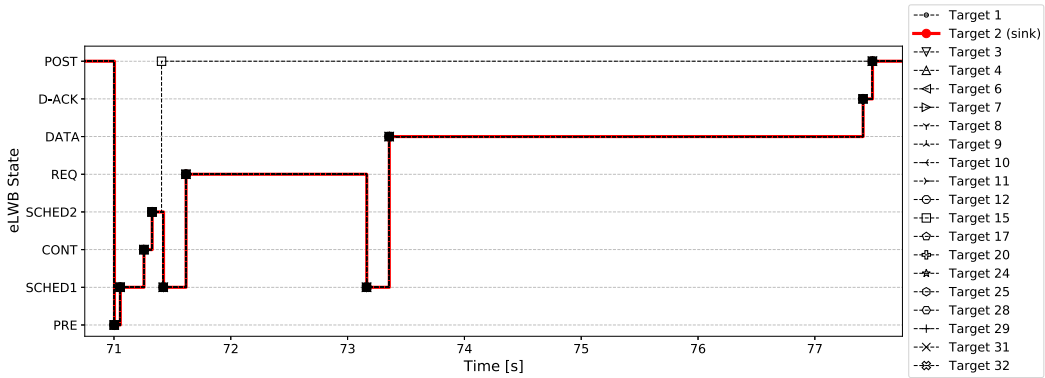

Fig. 11. Visualization of the data trace of the `elwb_state` variable for one exemplary eLWB round executed by 20 nodes. All nodes that successfully received the first schedule are synchronized and follow the eLWB states of the sink node. In this example, node 15 did not successfully receive the second schedule during the SCHED2 phase and stops participating before the eLWB round ends.

must match the sequence of messages with events received by the sink `collected` from the same node.

The three variables `rx_len`, `retx_len`, and `tx_len` indicate the fill level of the message queues and permit together with the variables `detected` and `collected` to keep track of the message flows (aspect A2). This involves the number of events that are detected, transmitted, received, acknowledged, and finally collected on the sink (see Figure 9). The relations that must hold for every eLWB round can be summarized by Equations (5)–(7). The arrows indicate whether the elements added (↑) or removed (↓) from a queue are counted. S corresponds to the set of all sensor nodes. For every detected event (`detected`), a corresponding Tx-queue (`tx_len`) element must be added, except if the event is discarded. For every Retransmit-queue element that is removed and not added to the Tx-queue (and not discarded), a corresponding Rx-queue (`rx_len`) element must be added on the sink node. These two relations are expressed in Equation (5). For every Tx-queue element that is removed, a corresponding Retransmit-queue element needs to be added (Equation (6)). Finally, all messages inserted into the Rx-queue must be included in the collected events (Equation (7)),

$$\underbrace{\sum_{k\in S}\left(\#\texttt{detected}_k + \texttt{retx\_len}_k\downarrow - \texttt{tx\_len}_k\uparrow - \#\texttt{discarded}_k\right)}_{=\texttt{rx\_len}'\uparrow} = \texttt{rx\_len}\uparrow, \tag{5}$$

$$\texttt{tx\_len}_k\downarrow = \texttt{retx\_len}_k\uparrow \quad \forall k \in S, \tag{6}$$

$$\texttt{rx\_len}\downarrow = \#\texttt{collected}. \tag{7}$$

In the example test, the code for the sensor nodes contains an (artificial) bug. As a result, the messages from the Retransmit-queue are not added to the Tx-queue but are discarded when the sink node indicates no acknowledgement for the corresponding message. Since the message flow extends over several nodes (see Equation (5)), it is necessary to be able to observe the change of the distributed state of the system to detect the problem. With the tracing system presented in this article, we can log all the associated distributed state changes in a non-intrusive fashion. The accurate time synchronization of the tracing system permits us to compare the state changes of different nodes.

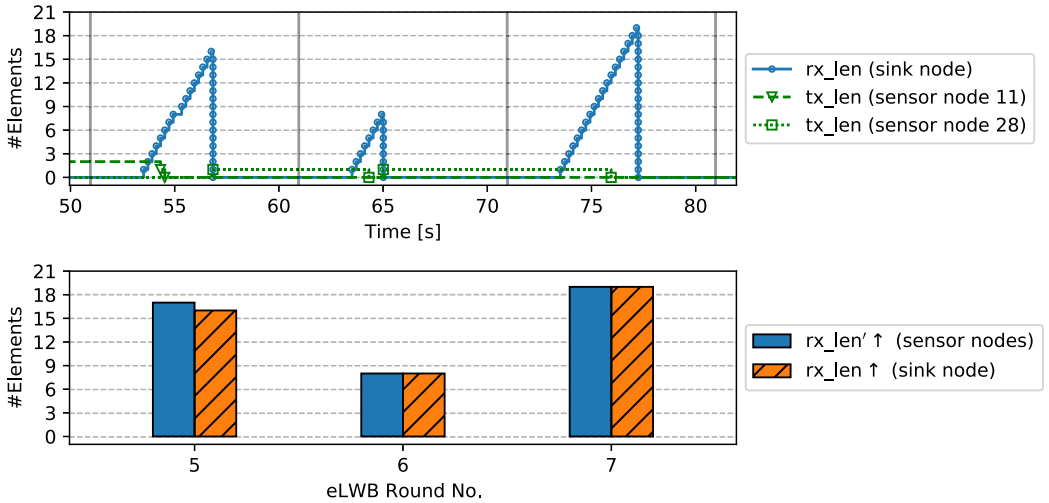

Fig. 12. The upper plot visualizes the queue fill levels. To improve readability, only the fill levels of the Rx-queue on the sink and the Tx-queues of two sensor nodes are plotted. The lower plot shows the change of the number of Rx-queue elements per round as traced on the sink node (`rx_len` ↑) and as derived based on the tracing of the sensor nodes (`rx_len′` ↑). The inequality between the two values in eLWB round 5 indicates that the behavior is erroneous.

An example visualization used for the validation of the message flow is depicted in Figure 12. The plots depict three eLWB round periods. The upper plot shows the number of messages contained in the queues. For better readability, only the fill levels of the Rx-queue on the sink and the Tx-queue of two sensor nodes are plotted. The lower plot of the figure shows the number of messages successfully transmitted from all sensor nodes to the sink node in each round. For every round, this value is determined twice: One time the value is calculated using the fill levels on all sensor nodes (left side of Equation (5)), a second time the value is determined based on the fill level of the Rx-queue (right side of Equation (5)). Under the assumption that messages are not discarded and acknowledgment messages are successfully received, both of which apply in the presented example, the two values must be equal, i.e., $\texttt{rx\_len}' \uparrow \stackrel{!}{=} \texttt{rx\_len} \uparrow$, in every round. Since $\texttt{rx\_len}' \uparrow > \texttt{rx\_len} \uparrow$ in round 5, we can conclude that messages were not properly forwarded to the sink node and that the implementation is erroneous in this aspect. The accurate time synchronization of the distributed observations allows to investigate the exact cause and timing of the error in more detail. The evaluation shown in this section is only a simple example. More sophisticated methods, as discussed in Section 4.4.2, can be used to perform a more in-depth evaluation and could help to automatically detect invalid state transitions and identify corner cases in state machines.

7 CONCLUSIONS

In this article, we present how native on-chip debugging can be put to use in the context of testbeds for wireless IoT devices. We demonstrate that the main advantages of on-chip debugging are that the behavior of the program is not altered by the tracing and that explicit instrumentation of the code is not necessary. To enable this in a distributed network, we propose an architecture based on commercially available debug probes. Because the distributed debug probes have different time bases, we design a methodology to temporally synchronize the traces collected with the tracing system. We implement the proposed tracing system in the FlockLab 2 testbed and show its viability for

the development of software for wireless IoT devices in a case study. We compare the proposed tracing method with traditional debugging via the serial interface and demonstrate how non-intrusive tracing can be successfully used for the verification of a networking protocol implementation.

Non-intrusive and instrumentation-free tracing allows investigations and observations that can be directly applied to an actually deployed system, since the same program code can be used without any modification. The presented tracing system allows automated test execution without requiring manual interaction during the test. This is advantageous in the context of a testbed with many nodes and simplifies repeatability. Time synchronization of the traces is important to meaningfully compare the distributed state of networking protocols at network scale. The achieved time synchronization accuracy of the example implementation is in the order of sub-milliseconds. This is sufficient for the tracing needs of most higher-layer protocols but should be complemented by traditional methods for precise tracing, such as logic tracing, for low-level hardware aspects. The fixed number of traceable variables or memory locations can be limiting for certain applications. However, the unmodified binary image can be reused to trace additional variables in a new execution. The presented system is well suited for an automated verification of an existing implementation as well as for the identification of misbehavior during the development process. The system is available as open-source to the public as part of the FlockLab 2 testbed and has shown to be of great use during the development of highly time-sensitive protocols based on synchronous transmissions.

REFERENCES

[1] Cedric Adjih, Emmanuel Baccelli, Eric Fleury, Gaetan Harter, Nathalie Mitton, Thomas Noel, Roger Pissard-Gibollet, Frederic Saint-Marcel, Guillaume Schreiner, et al. 2015. FIT IoT-LAB: A large scale open experimental IoT Testbed. In *Proceedings of the IEEE 2nd World Forum on Internet of Things (WF-IoT'15)*. IEEE, 459–464. https://doi.org/10.1109/WF-IoT.2015.7389098

[2] Akshit Akhoury, Krishna Birla, Rohit Sarkar, Arun Ravi, Shaleen Kalsi, and Subhojit Ghorai. 2019. Design and analysis of RTOS and interrupt based data handling system for nanosatellites. In *Proceedings of the IEEE Aerospace Conference*. 1–9. https://doi.org/10.1109/AERO.2019.8742184

[3] Beshr Al Nahas, Simon Duquennoy, and Olaf Landsiedel. 2019. Concurrent transmissions for multi-hop bluetooth 5. In *Proceedings of the 2019 International Conference on Embedded Wireless Systems and Networks (EWSN'19)*. Junction Publishing, 130–141. https://dl.acm.org/doi/10.5555/3324320.3324336.

[4] Jan Beutel, Roman Trüb, Reto Da Forno, Markus Wegmann, Tonio Gsell, Romain Jacob, Michael Keller, Felix Sutton, and Lothar Thiele. 2019. The dual processor platform architecture. In *Proceedings of the 18th International Conference Information Processing in Sensor Networks (IPSN'19)*. ACM, 335–336. https://doi.org/10.1145/3302506.3312481

[5] Atis Elsts, Xenofon Fafoutis, Simon Duquennoy, George Oikonomou, Robert Piechocki, and Ian Craddock. 2018. Temperature-resilient time synchronization for the internet of things. *IEEE Trans. Industr. Inf.* 14, 5 (2018), 2241–2250. https://doi.org/10.1109/TII.2017.2778746

[6] Federico Ferrari, Marco Zimmerling, Lothar Thiele, and Olga Saukh. 2011. Efficient network flooding and time synchronization with glossy. In *Proceedings of the 10th ACM/IEEE International Conference on Information Processing in Sensor Networks*. IEEE, 73–84. https://ieeexplore.ieee.org/abstract/document/5779066.

[7] Kai Geissdoerfer, Mikołaj Chwalisz, and Marco Zimmerling. 2019. Shepherd: A portable testbed for the batteryless IoT. In *Proceedings of the 17th Conference on Embedded Networked Sensor Systems (SenSys'19)*. ACM, 83–95. https://doi.org/10.1145/3356250.3360042

[8] Tao Gong, Huayi Ji, Song Han, Tianyu Zhang, Chuancai Gu, Xiaobo Sharon Hu, and Mark Nixon. 2017. Demo abstract: A cross-device testing and reporting system for large-scale real-time wireless networks. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'17)*. IEEE, 157–158. https://doi.org/10.1109/RTAS.2017.21

[9] Jorg Hahner, Kurt Rothermel, and Christian Becker. 2004. Update-linearizability: A consistency concept for the chronological ordering of events in MANETs. In *Proceedings of the IEEE International Conference on Mobile Ad-hoc and Sensor Systems*. IEEE, 1–10. https://doi.org/10.1109/MAHSS.2004.1392060

[10] Vlado Handziski, Andreas Köpke, Andreas Willig, and Adam Wolisz. 2006. Twist: A scalable and reconfigurable testbed for wireless indoor experiments with sensor networks. In *Proceedings of the 2nd International Workshop on Multi-hop Ad-hoc Networks (REALMAN'06)*. 63–70. https://doi.org/10.1145/1132983.1132995

[11] Konrad Iwanicki, Przemyslaw Horban, Piotr Glazar, and Karol Strzelecki. 2014. Bringing modern unit testing techniques to sensornets. *ACM Trans. Sens. Netw.* 11, 2, Article 25 (2014), 41 pages. https://doi.org/10.1145/2629422

[12] Yu Jiang, Houbing Song, Yixiao Yang, Han Liu, Ming Gu, Yong Guan, Jiaguang Sun, and Lui Sha. 2018. Dependable model-driven development of CPS: From stateflow simulation to verified implementation. *ACM Trans. Cyber-Phys. Syst.* 3, 1, Article 12 (2018), 31 pages. https://doi.org/10.1145/3078623

[13] Christoph Lenzen, Philipp Sommer, and Roger Wattenhofer. 2015. PulseSync: An efficient and scalable clock synchronization protocol. *IEEE/ACM Trans. Netw.* 23, 3 (2015), 717–727. https://doi.org/10.1109/TNET.2014.2309805

[14] Shangrong Li, Junyan Ma, and Yi Li. 2020. A testbed for hardware-assisted online profiling of IoT devices. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops (ICSEW'20)*. ACM, 627–630. https://doi.org/10.1145/3387940.3392247

[15] Antonio Libri, Andrea Bartolini, Michele Magno, and Luca Benini. 2016. Evaluation of synchronization protocols for fine-grain HPC sensor data time-stamping and collection. In *Proceedings of the International Conference on High Performance Computing & Simulation (HPCS'16)*. IEEE, 818–825. https://doi.org/10.1109/HPCSim.2016.7568419

[16] Roman Lim, Federico Ferrari, Marco Zimmerling, Christoph Walser, Philipp Sommer, and Jan Beutel. 2013. FlockLab: A testbed for distributed, synchronized tracing and profiling of wireless embedded systems. In *Proceedings of the 12th International Conference on Information Processing in Sensor Networks (IPSN'13)*. ACM, 153–166. https://doi.org/10.1145/2461381.2461402

[17] Roman Lim, Balz Maag, and Lothar Thiele. 2016. Time-of-flight aware time synchronization for wireless embedded systems. In *Proceedings of the 2016 International Conference on Embedded Wireless Systems and Networks (EWSN'16)*. Junction Publishing, 149–158. https://dl.acm.org/doi/10.5555/2893711.2893732.

[18] Roman Lim and Lothar Thiele. 2017. Testbed assisted control flow tracing for wireless embedded systems. In *Proceedings of the International Conference on Embedded Wireless Systems and Networks (EWSN'17)*. Junction Publishing, 180–191. https://dl.acm.org/doi/10.5555/3108009.3108033.

[19] Arm Limited. 2013. CoreSight Technical Introduction. Retrieved from https://developer.arm.com/documentation/epm039795/latest.

[20] Arm Limited. 2021. ARMv7-M Architecture Reference Manual. Retrieved May 27, 2021 from https://developer.arm.com/documentation/ddi0403/latest/.

[21] Michael Okola and Kamin Whitehouse. 2010. Unit testing for wireless sensor networks. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Sensor Network Applications (SESENA'10)*. ACM, 38–43. https://doi.org/10.1145/1809111.1809123

[22] Stefanos Peros, Stéphane Delbruel, Sam Michiels, Wouter Joosen, and Danny Hughes. 2020. Simplifying CPS application development through fine-grained, automatic timeout predictions. *ACM Trans. IoT* 1, 3, Article 18 (2020), 30 pages. https://doi.org/10.1145/3385960

[23] Hendrik Roehm, Jens Oehlerking, Matthias Woehrle, and Matthias Althoff. 2019. Model conformance for cyber-physical systems: A survey. *ACM Trans. Cyber-Phys. Syst.* 3, 3, Article 30 (2019), 26 pages. https://doi.org/10.1145/3306157

[24] Kay Römer and Junyan Ma. 2009. PDA: Passive distributed assertions for sensor networks. In *Proceedings of the International Conference on Information Processing in Sensor Networks*. IEEE, 337–348. https://ieeexplore.ieee.org/abstract/document/5211917/.

[25] Raimondas Sasnauskas, Olaf Landsiedel, Muhammad Hamad Alizai, Carsten Weise, Stefan Kowalewski, and Klaus Wehrle. 2010. KleeNet: Discovering insidious interaction bugs in wireless sensor networks before deployment. In *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN'10)*. ACM, 186–196. https://doi.org/10.1145/1791212.1791235

[26] Markus Schuß, Carlo Alberto Boano, Manuel Weber, and Kay Römer. 2017. A competition to push the dependability of low-power wireless protocols to the edge. In *Proceedings of the 14th International Conference on Embedded Wireless Systems and Networks (EWSN'17)*. Junction Publishing, 54–65. https://dl.acm.org/doi/10.5555/3108009.3108018.

[27] Philipp Sommer and Branislav Kusy. 2013. Minerva: Distributed tracing and debugging in wireless sensor networks. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems (SenSys'13)*. ACM, Article 12, 14 pages. https://doi.org/10.1145/2517351.2517355

[28] Felix Sutton, Reto Da Forno, David Gschwend, Tonio Gsell, Roman Lim, Jan Beutel, and Lothar Thiele. 2017. The design of a responsive and energy-efficient event-triggered wireless sensing system. In *Proceedings of the 2017 International Conference on Embedded Wireless Systems and Networks (EWSN'17)*. Junction Publishing, 144–155. https://dl.acm.org/doi/10.5555/3108009.3108028.

[29] Matthew Tancreti, Mohammad Sajjad Hossain, Saurabh Bagchi, and Vijay Raghunathan. 2011. Aveksha: A hardware-software approach for non-intrusive tracing and profiling of wireless embedded systems. In *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems (SenSys'11)*. ACM, 288–301. https://doi.org/10.1145/2070942.2070972

[30] Matthew Tancreti, Vinaitheerthan Sundaram, Saurabh Bagchi, and Patrick Eugster. 2015. TARDIS: Software-only system-level record and replay in wireless sensor networks. In *Proceedings of the 14th International Conference on Information Processing in Sensor Networks (IPSN'15)*. ACM, 286–297. https://doi.org/10.1145/2737095.2737096

[31] Francisco Tirado-Andrés and Alvaro Araujo. 2019. Performance of clock sources and their influence on time synchronization in wireless sensor networks. *Int. J. Distrib. Sens. Netw.* 15, 9 (2019). https://doi.org/10.1177/1550147719879372

[32] Roman Trüb, Reto Da Forno, Lukas Sigrist, Lorin Mühlebach, Andreas Biri, Jan Beutel, and Lothar Thiele. 2020. FlockLab 2: Multi-modal testing and validation for wireless IoT. In *Proceedings of the 3rd Workshop on Benchmarking Cyber-Physical Systems and Internet of Things (CPS-IoTBench'20)*. OpenReview.net. https://doi.org/10.3929/ethz-b-000442038

[33] Kamin Whitehouse, Gilman Tolle, Jay Taneja, Cory Sharp, Sukun Kim, Jaein Jeong, Jonathan Hui, Prabal Dutta, and David Culler. 2006. Marionette: Using RPC for interactive development and debugging of wireless embedded networks. In *Proceedings of the 5th International Conference on Information Processing in Sensor Networks (IPSN'06)*. ACM, 416–423. https://doi.org/10.1145/1127777.1127840

[34] Matthias Woehrle, Kai Lampka, and Lothar Thiele. 2013. Conformance testing for cyber-physical systems. *ACM Trans. Embed. Comput. Syst.* 11, 4, Article 84 (2013), 23 pages. https://doi.org/10.1145/2362336.2362351

[35] Jing Yang, Mary Lou Soffa, Leo Selavo, and Kamin Whitehouse. 2007. Clairvoyant: A comprehensive source-level debugger for wireless sensor networks. In *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems (SenSys'07)*. ACM, 189–203. https://doi.org/10.1145/1322263.1322282

[36] Li Yi, Junyan Ma, and Te Zhang. 2019. HATBED: A distributed hardware assisted testbed for non-invasive profiling of IoT devices. In *Proceedings of the 2nd Workshop on Benchmarking Cyber-Physical Systems and Internet of Things (CPS-IoTBench'19)*. ACM, 13–17. https://doi.org/10.1145/3312480.3313172

Received Februray 2021; revised June 2021; accepted August 2021