μ*Chaos*: Moving Chaos Engineering To IoT Devices

Wojciech Kalka[1[0009−0003−1802−692X]] and
Tomasz Szydlo[1,2[0000−0003−4101−2115]]

[1] Institute of Computer Science, AGH University of Krakow, Al. Mickiewicza 30, 30-059 Krakow, Poland `wkalka@agh.edu.pl`
[2] Newcastle University, Newcastle Upon Tyne, NE1 7RU, United Kingdom
`tomasz.szydlo@{newcastle.ac.uk|agh.edu.pl}`

Abstract. The concept of the Internet of Things (IoT) has been widely used in many applications. IoT devices can be exposed to various external factors, such as network congestion, signal interference, and limited network bandwidth. This paper proposes an open-source μ*Chaos* software tool for the ZephyrOS real-time operating system for embedded devices. The proposed tool intends to inject failures into device's applications in a controlled manner to improve their error-handling algorithms. The proposed novel framework fills the gap in the chaos engineering tools for edge devices in the cloud-edge continuum. In the paper, we also discuss the typical failures of IoT devices and the potential use cases of the solution.

Keywords: IoT · Chaos Engineering · fault injection · Zephyr-OS.

1 Introduction

According to predictions, the number of IoT devices is expected to cross 29 billion in 2030. This is an increase of about 300% compared to 2019[1]. Nowadays, IoT devices become ubiquitous in many areas of life so they can be exposed to various external factors. The spectrum of their operation covers from smart home applications, through transport or industrial areas, to extreme conditions, such as sensors working at fire sites to track the safety of firefighters[8]. Therefore, ensuring the work continuity of IoT devices is a significant challenge.

Nowadays, more and more companies are paying attention to the proper testing of products before they are launched into production. It is crucial to provide consumers with an appropriate quality and resilience of services[19]. This has given rise to the concept of *Chaos Engineering*, which Netflix introduced [3] in 2010. Before that, engineers were forced to introduce constant improvements and add new functionalities to handle noticed problems. The systems' complexity caused that there was not possible to predict all possible failures during the design time[7]. This led to the creation of Chaos Monkey, a tool for the unpredictable termination of system parts, including virtual machines and containers,

[2] Netflix, https://netflix.github.io/chaosmonkey/

ICCS Camera Ready Version 2024
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-63783-4_18

to test whether the system is fault-tolerant. However, chaos engineering could be deployed not only in IoT but also in e.g. blockchain technologies[20].

In the IoT domain, the failure resiliency problem is more complex due to its multilayered architecture. The outages and other disturbances could appear in many places along the sensor data processing paths. The IoT systems generally comprise three layers - perception, edge, and the cloud.

Each layer has its weak spots that can cause failures. In this paper, we will focus on the perception layer where IoT devices are prone to errors stemming from various factors. Since the most commonly used type of communication is wireless connectivity, network congestion, signal interference, and limited bandwidth can disrupt communication between devices and cause delays or failures in data transmission. Modern microcontrollers allow for more complex operations, enabling TinyML algorithms, which are susceptible to sensor data failures. These examples and many others conclude that chaos engineering has open challenges in IoT devices.

Several chaos engineering tools exist for cloud services, such as Chaos Twin [4], Chaos Monkey, CHAOSORCA[13], Chaos Recommendation Tool[18], or Chaos Mesh for Kubernetes. However, there are few known solutions for IoT end devices, but they mainly work in static software testing, e.g., Chaos Duck[5]. Other tools were also created for faults injection, but they are focused on the network layer using customized MQTT broker commonly used in IoT[6]. This paper proposes an open-source *uChaos* software component for the real-time operating system ZephyrOS for embedded devices supported by organizations like Google, Intel, and Nordic Semiconductor. By enabling *uChaos* component in the OS, a user can inject several types of faults, including the ones related to the sensors, hardware, network, and application. The *uChaos* exposes a flexible API that can be easily interacted with via serial port or console, enabling interaction during the tests. In the paper, we will discuss types of faults in IoT devices and show how the proposed novel tools and the software library work in the selected use cases.

The paper is organised as follows. Section 2 presents related work, and Section 3 introduces fault taxonomy in IoT and faults in IoT end devices. The next section describes the concept and the details of the designed tool. Section 5 presents the use case and experiments, while the last section gives conclusions.

1.1 Challenges

Performing chaos engineering techniques for IoT devices requires facing the following challenges: (i)identification of the common faults related to the IoT devices, (ii)introduction of the failure scenarios mimicking real failures, (iii)seamless integration of the chaos engineering tools with the operating systems for embedded devices.

ICCS Camera Ready Version 2024
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-63783-4_18

1.2 Contribution

This paper proposes a novel chaos engineering tool for IoT devices encompassing several types of failures and failure scenarios. The following are the paper's key contributions: (i) analysis of the common failures targeting IoT devices and the mimicking scenarios, (ii) implementation of the *uChaos* tool for Zephyr OS, (iii) evaluation of the solution in typical IoT use cases.

2 Related work

Due to the complexity resulting from the multi-layer architecture of IoT systems, it is challenging to have a holistic tool to test them through the full architecture stack. Therefore, different testing methods depend on which part or layer of the system is analysed. Properly chosen testing strategy and selected tools make the debugging and development process more efficient. Finally, the chaos engineering tools should allow for finding the bugs early in development, saving time and reducing project costs. This section presents a selected tool related to chaos engineering, listed in Table 1.

Chaos Monkey[3], developed by Netflix, aims to terminate in a controllable manner production instances of one or more virtual machines. These machines can run some microservices, e.g. video transcoding or streaming. It is possible to configure the scheduler to trigger failures at times when they can be better monitored. Another tool, *Gremlin*[2] is provided as a SaaS (Software as a Service) platform[4]. It allows testing cloud-based applications and infrastructure. The tool works with the most popular platforms like AWS or Azure. It has an agent that should be installed on a user container or virtual machine. Gremlin provides different possibilities for failure introduction, including computational resources, network, and system states by killing processes.

Chaos Mesh[5] is an open-source tool for cloud-based solutions, widely used with Kubernetes. It supports many types of faults, i.e. network latency, packet loss, HTTP communication latency, CPU race, system time changes, and platform faults like AWS node restart. It supports losing packets, pressure on the CPU, increasing physical disk load, filling disks, and killing or stopping processes. *Chaos Twin*[4] is a simulation tool created to work with cloud-based systems. It allows the creation of the digital twin of a system under test and then evaluates its performance on a business level by finding the most optimized architecture. Digital twin is a concept which occurs in more research[17]. The tool focuses on three types of errors, including partial or complete data centre failures, incorrect operation of virtual machines, and delays in communication between data centres. Simulations are run until the specified number of iterations is reached.

Chaos Duck[6] is a tool for testing IoT devices that emulates fault injection attacks. Injecting faults via hardware is difficult and expensive, so *Chaos Duck*

[3] Netflix, https://github.com/Netflix/chaosmonkey
[4] Gremlin, https://github.com/gremlin/chaos-engineering-tools
[5] The Linux Foundation, https://chaos-mesh.org/
[6] Igor Zavalyshyn, https://github.com/zavalyshyn/chaosduck

ICCS Camera Ready Version 2024
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-63783-4_18

uses a tested program binary file. It disassembles a chosen file and collects information about the code, e.g. branch instructions, static variables, and address space. *Chaos Duck* produces many faulted binaries and executes each by collecting statistics about the impact of each fault. The tool supports x86 and ARM architectures. The following error types are available - *bit flip (FLP)*, *byte zeroing*, and adding *conditional, and unconditional branches*.

Custom MQTT Broker[7] is a tool created for testing systems that use communication via MQTT. The authors have chosen a broker as a crucial element in the distributed systems, independent from factors such as complexity, and focused on the network layer as a place to inject failures. Messages are gathered and modified before sending to the clients. Each rule consists of the *topic* and an array of filters named *operators*. Operators transform the messages and pass them to the next one as an additional parameter. The tool has four operators - *map*, *randomDelay*, *message buffering*, and *randomDrop*.

The discussed tools do not focus on the internal peripherals of the IoT devices, which are essential from a data processing perspective. They are fairly directed to cloud layer of the IoT systems. Therefore, the proposed in this work *uChaos* tool is designed to fit into the embedded operating system, acting as the intermediate layer between the sensors and the application.

Table 1: Chaos Engineering testing tools for IoT systems.

Tool name	Layer	CPU	Memory	Peripheral Device	Virtual Machines	Network	Sensor	Application
Chaos Monkey	Could	○	○	○	●	○	○	○
Gremlin	Cloud	●	●	●	○	●	○	●
Chaos Mesh	Cloud	●	●	○	○	●	○	●
Chaos Twin	Cloud	○	○	○	●	●	○	○
Chaos Duck	End device	○	○	○	○	○	○	●
Custom MQTT Broker	Edge	○	○	○	○	●	●	○
uChaos	IoT	●	●	○	○	○	●	●

● - Supported
○ - Not supported

2.1 IoT faults taxonomy

Due to the complexity of IoT systems, errors can occur at different layers - cloud (e.g. server failures), edge (e.g. package loss, incorrect data processing) and end devices (e.g. sensor damage). This makes it challenging to create a very detailed IoT failure taxonomy. However, referring to works describing faults in WSN (Wireless Sensors Networks)[12], combined with the information regarding IoT faults[10], it is possible to introduce a general classification, divided into different categories. Fig. 1 shows the taxonomy of IoT faults proposed in this paper and divided into four categories of faults affiliation, bahaviour, time, system layer, and location.

From the system's operational perspective, the main distinctive factor is the type of failure, which might be either hard or soft. A soft fault is when a device

[7] SIGNEXT, https://github.com/SIGNEXT/instrumentable-aedes

ICCS Camera Ready Version 2024
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-63783-4_18

can respond and communicate, but the system knows it is damaged, e.g. it sends invalid data. On the contrary, a hard error is when the component or the device is not detectable or completely broken. Low battery and no power are good examples[15].

Another issue is related to the time domain during which that failure might appear. This category covers permanent, transient, or intermittent failures. Long-term faults are usually permanent. An example of such an error may be the previously mentioned power supply problem or a damaged communication module. Intermittent faults may be caused by changes in the operating environment, such as sensors covered by, e.g. worms, dust, plants, shadows, or objects placed by humans. Intermittent faults last longer than transient faults. The interval between their appearance might vary and be unpredictable[11].

The next group of faults is related to the system layers. End devices are very susceptible to errors, which may come from the outside environment they are working in or might result from their invalid operation. Each IoT device can be described from two perspectives - hardware and software. The hardware is mainly CPU, communication module, power module and all kinds of sensors. Any of these items may be damaged or contain manufacturing errors. The most common software fault is a problem with memory management. This can lead to memory leaks and device malfunction. Other common issues are sensor calibration, data processing, and packet forming. Then, the edge layer might be susceptible to failures such as insufficient network bandwidth, resulting in packet loss or incorrect processing/filtering of received data. Finally, in the cloud layer, the failure might appear with service configurations, database failures, errors in the application or the wrong selection of algorithms for processing the results.

The last group considered in the taxonomy is location-based faults. These can be internal system faults, such as connectivity or infrastructure problems. Defects from the external environment are important, as they significantly impact end devices. Changes such as temperature spikes can interfere with measurements, and electromagnetic interference or discharges can damage the device.

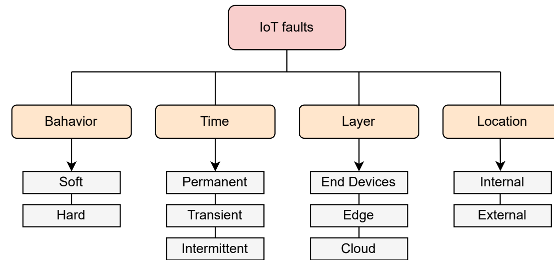

Fig. 1: IoT faults taxonomy.

2.2 Operating Systems for IoT

There are many operating systems for embedded systems on the market. Most are open-source, and choosing the right system depends on the project's require-

ICCS Camera Ready Version 2024
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-63783-4_18

ments. Some big tech companies have invested in developing Real-time Operating Systems (RTOS), for example, Amazon or Microsoft. One of the most popular is FreeRTOS maintained mainly by Amazon. It provides the RTOS API, with no support for other functionality such as GPIO or serial drivers, which are left to the developer to create.

The Zephyr OS is governed by the Linux Foundation. Its architecture is similar to Linux, e.g. the kernel is configured similarly. It is intended for complex and high-level embedded applications. It has ready-to-use libraries and supports various hardware target processors and development boards. It is more portable than other operating systems but could be more complicated for less experienced users. It has more than 450 officially supported boards and a lot of ready-to-use examples of system features.

Mbed OS is an RTOS for IoT solutions based on Cortex-M boards. It provides an abstraction layer for C/C++ applications that ensures portability between platforms supporting this operating system. There are all necessary operating system components e.g. Semaphores, Queues, Threads, Mutexes and Scheduler for switching between application activities.

Since 2019, Microsoft has supported the development of Azure RTOS ThreadX. One of the most important features is an advanced multitasking solution named *preemption-threshold scheduling*. It allows a sub-microsecond context switching and was a topic of academic research[16]. Manufacturers emphasize that performance is the feature that distinguishes ThreadX from other RTOSs. One of the solutions is the *Picocernel* design, where services don't have many layers, so additional overhead in function call is removed. Moreover, interrupt handling is also optimized because only scratch registers are processed.

3 Chaos tool concept

This section presents the concept of chaos engineering tool and its working principles. We will analyze and model faults introduced by the tool and discuss its integration with the resource-constrained Zephyr operating system.

3.1 IoT faults modeling

In this paper, we will mainly focus on IoT end device faults. For this group of devices, we are dividing them based on three criteria - hardware, software, and data. IoT devices are often specialized to work in specific environments and are most likely optimized for low energy consumption. Based on [14], we have collected selected faults in the Table 2, Table 3, and Table 4 along with their descriptions.

The first group of failures is associated with data processing. Data failures might result in the inability to access or utilize it effectively. These failures can occur due to various reasons, including technical issues, human error, software bugs, cybersecurity breaches, natural disasters, and hardware malfunctions. The selected failures are presented in Table 2.

ICCS Camera Ready Version 2024
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-63783-4_18

The significant challenge is the proper handling of software failures, as it can lead to the dysfunction of the entire device or its most important parts, e.g. communication with the rest of the system. The selected failures are presented in Table 4. Dealing with these failures in microcontroller-based devices with limited memory and computing resources is demanding. Every software failure, which forces the service staff to go and recheck the device state or program, generates additional costs that should be avoided.

The last group of failures covers hardware ones. Depending on the conditions, it is critical to properly secure the device to protect it from external conditions and possible mechanical damage because it can manifest itself in many ways and be challenging to detect. Selected failures are summarized in Table 3.

Table 2: Data faults

Fault	**Classification**	**Descripiton**	**Inject**
Outlier	Soft, Intermittent	Single, unexpected value that comes a lot over normal measurements. It could be caused by some hardware problems like unstable sensor connection or external factors e.g. electromagnetic pulse.	Pseudo-random value, chosen by the user and added to the original measurement, occurs with pseudo-random frequency.
Spike	Soft, Transient	Set of data with a value different than expected, often as a result of supply or connection failures.	The peak in read sensor data, which rises and slopes symmetrically in a number of samples set by the user.
Offset/ Rotation	Soft, Transient	Some value constantly added to the output. If the sensor position was changed due to some case or mounting system damage, it may produce different values.	Sample is increased every time by a constant percentage part of measurement, chosen by the user.
Stuck at value	Soft, Permanent	Sensor readings stay the same for a period of time. The reason may be a sensor malfunction.	Value returned by a sensor is constant and taken from the first sample after command execution.
Noise	Soft, Permanent	An additional, approximately constant value changing the read results may be related to a change in the environment in which the sensor operates, e.g. covering by another object, dirt, or change in ambience temperature.	Similar as in 'Outlier' however a sample is increased by value in every measurement.

3.2 μChaos Tool Design

Among the open-source Real-time Operating Systems (RTOS) solutions available on the market and discussed in Section 2.2, we have chosen Zephyr OS. The most important aspect is that it supports sensors, memory management and protocols used in IoT and provides modularity. Noteworthy is also the fact that it has a significant and growing community size. Another aspect was to ensure

ICCS Camera Ready Version 2024
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-63783-4_18

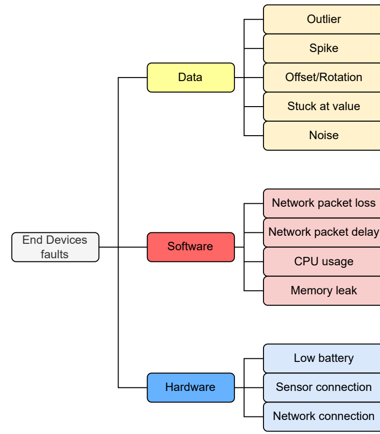

Fig. 2: IoT end devices faults.

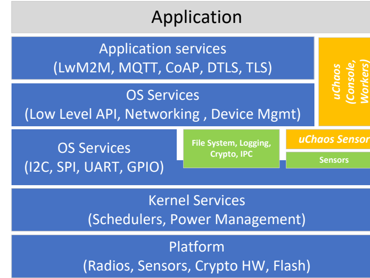

Fig. 3: ZephyrOS high-level architecture with μ*Chaos* components marked orange.

Table 3: Hardware faults

Fault	**Classification**	**Descripiton**	**Inject**
Low battery	Hard, Permanent	The amount of energy drawn from the battery prevents proper operation.	Device battery measured voltage value is reduced by the value and with timestamp selected by the user.
Sensor connection	Soft, Transient	IoT devices usually have sensor connected. Every mechanical connection has limits to its strength. Prolonged exposure to external conditions may cause some damages. There are also many other situations that are not related to harsh environmental conditions.	Zephyr has a special value to signal, that the microcontroller is unable to communicate with the sensor. This value is returned with a pseudo-random range set by the user.

that μChaos did not take up too much RAM and ROM resources due to the fact that it is a library dedicated to embedded devices.

As shown in Fig. 3 the tool comprises two components - *uChaos Sensor* and *uChaos Console*. The first one, *uChaos Sensor*, is designed to manipulate the measurement values. The second one, *uChaos Console*, is implemented as a separate thread working in the background and manages the sensor intermediate layer and introduces failures. Its functionality is exposed via UART (Universal Asynchronous Receiver-Transmitter) and thus can be controlled by external physical components but also is exposed to Zephyr Console where failures might be introduced via interactive command line console. The tool is provided as an open-source library, available on GitHub [8].

[8] https://github.com/wkalka/uChaos

ICCS Camera Ready Version 2024
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-63783-4_18

Table 4: Software faults

Fault	**Classification**	**Descripiton**	**Inject**
CPU usage	Hard, Transient	When too many tasks in the operating system work concurrently and there are no more available resources to do other operations. There may also be a bug in the software in tasks switching and one or more still occupy the CPU.	Set of threads to possibly add to the application. Their number with all necessary parameters like priority, stack size, and area has to be determined by the user at program compilation time. At the application beginning all defined threads are ready to start and the user can freely start them and then suspend and resume.
Memory leak	Hard, Intermittent	One of the common mistakes, especially when it comes to IoT devices, as developers have very little available memory and computing resources. It is quite easy to come over the size of the array with e.g. measurements.	Blocks of dynamically allocated memory with the size chosen by the user. Each block has a name to handle and a pointer to allocated memory. Blocks could be created at any time the application is running.

3.3 Data faults injection

Data faults are the most common group of failures available in μ*Chaos Sensor* component of the library. μ*Chaos Sensor* has been designed as a wrapper on the sensor subsystem provided by Zephyr OS. After activating μChaos, each time the Sensor's methods are called, it redirects to the library function. It uses low-level functions to obtain the data, and then before passing the values to the user application code, failures are introduced, as discussed in the previous section. Every type of fault has its own method to manipulate the data or driver behaviour properly. Thanks to this, the applications' operations on the sensors remain unchanged.

One of the main goals of this project was to design the library coherently with the sensor drivers provided by the Zephyr OS. Therefore, the failure types are associated with the types of sensors and not the particular devices. To use μChaos in a project, the user has to initialize its components at the beginning. For the sensors part, e.g. there is a parameter that indicates the number of used sensors in the application. All settings could be changed in the configuration file, e.g. which types of failures will be used in the application.

3.4 Hardware faults injection

Hardware faults group is implemented by μ*Chaos Battery* component. This part of μ*Chaos* library is based on Zephyr OS Analog-to-Digital Converter (ADC) driver. Similarly to μ*Chaos Sensor* it uses low-level system functions to obtain the tested voltage. After every `adc_raw_to_millivolts_dt` ADC driver function call, which returns the measured value expressed in millivolts, the context

ICCS Camera Ready Version 2024
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-63783-4_18

is redirected to `uChaosBattery_RawToMillivoltsDt`. Inside this method, the data is manipulated to simulate malfunctions in a device's power supply. Voltage drops may occur for a certain number of measurements or continuously until the minimum permissible battery voltage is reached.

3.5 Software faults injection

In μChaos, a software faults are realised by *μChaos CPU*, where a CPU is loaded by additional operations e.g. variable iteration or mathematical calculations, which don't have any impact on the tested application. The aim is to capture a free CPU computational resources, to keep a CPU as long as possible in non-idle state. This could lead to limiting some functions of threads critical for application like communication with a network, monitoring power supply and measuring some crucial metrics. This part of μChaos library doesn't redirect behaviour of custom system functions. User has to define a thread with selected priority and name to recognize the thread via *μChaos Console*. After the application begins running all μChaos threads are in *prestart* state and then the user can start them or suspend them via certain commands e.g. *load_ add LoadThread*.

4 Use Case and Evaluation

In the evaluation, we consider a typical IoT device used to monitor industrial machines. In the target machine, we monitor its temperature and vibrations using a temperature sensor and an accelerometer. The IoT device is battery-powered. In the evaluation, we are focusing only on the perception layer. Therefore, we are analysing only the raw values provided by the sensors. Experiments realise different scenarios, each representing a distinct type of data errors shown in Fig. 2.

We have prepared experiments with two types of sensors (Fig. 4) - accelerometer ADXL345 and temperature sensor DPS310. Sensors were placed on a separate, small development board with connectors[9][10].

The application was run on nRF52 DK, a development kit from Nordic Semiconductor with nRF52832 Soc (System on Chip)[11]. It supports wireless communication like Bluetooth LE, Bluetooth mesh ANT and NFC.

4.1 Temperature

Scenario for the temperature sensor contains producing anomaly. During the experiment, the sensor was subjected to a heat source after the first 10 samples

[9] OKYSTAR, ADXL345, https://www.okystar.com/product-item/adxl345-digital-3-axis-gravity-acceleration-sensor-oky3247/

[10] Seeed Studio, DPS310, https://wiki.seeedstudio.com/Grove-High-Precision-Barometric-Pressure-Sensor-DPS310/

[11] Nordic Semiconductor, nRF52 DK, https://www.nordicsemi.com/Products/Development-hardware/nrf52-dk

ICCS Camera Ready Version 2024
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-63783-4_18

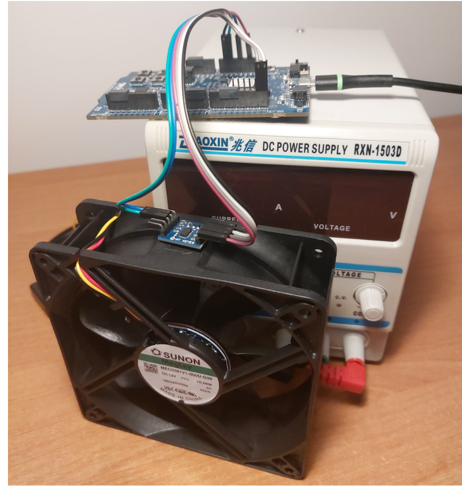

Fig. 4: Testbed setup.

were taken, and then a further 50 samples were captured. The heat source was then removed, and a final 150 measurements were taken to bring the sensor back to near its initial temperature. However, this took about 100 measurements due to the inertia of the sensor.

In the next experiment iteration, fault injection began after the first 5 samples (indicated by a black vertical line), before the heat source approach, and ended before the last 10 samples of the experiment. Regarding data anomaly, outliers appeared in the data every 6-10 measurements with a level between 50% and 70% of the original value. Separated points in Fig. 5 show data anomalies.

The temperature value chart is accompanied by a comparison of the average value and standard deviation across the entire measurement range. These re-

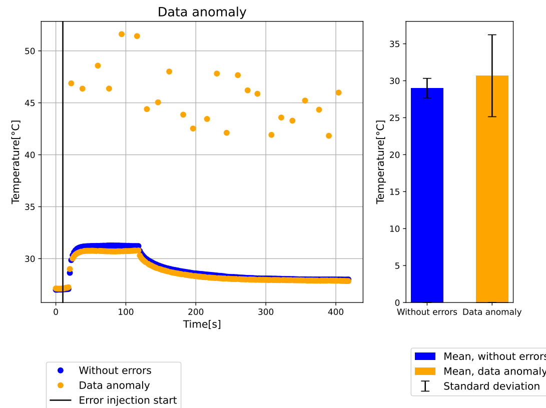

Fig. 5: Temperature sensor data anomaly.

ICCS Camera Ready Version 2024
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-63783-4_18

sult validate the experiment's assumption, that the mean is slightly higher than the clear signal, indicating that some changes have occurred in the data. This suggests that anomalies do not significantly impact the overall average value. However, they do contribute to an increased standard deviation due to the presence of scattered data points with anomalies.

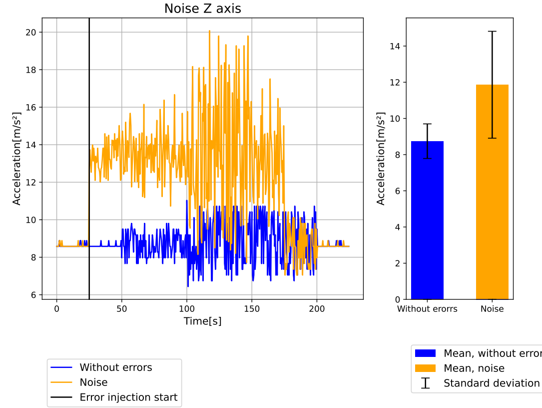

Fig. 6: Accelerometer Z axis noise.

4.2 Acceleration

In this case, we had a different scenario, where the noise error was injected. The acceleration was measured every 500ms and displayed on the device console. The sensor was mounted on top of a computer fan, while the fan speed was voltage-controlled by the laboratory power supply. Measurements were made at four fan speeds, including idle. Scenario consists of 450 samples. After every 100 samples, the speed was increased, and the last 50 measurements were at idle speed again. In case containing faults injection, it started after the first 50 samples of idle speed and finished after 50 measurements of the highest fan speed. The fault was applied to each accelerometer axis.

The black vertical line points to the moment of faults injection start. The axis Z showed the direction where the vibrations had the greatest impact including also the gravitational acceleration. The range of the added noise value was between 40% and 70% of the original values, as shown in Fig. 6. In analyzing the provided information, it is evident that the graph displaying the noisy data is noticeably shifted from the original graph. This observation is further supported by an approximate increase of 40%-45% in the average value. Additionally, the high standard deviation value indicates a substantial absolute value of noise, aligning with the assumptions made for the experiment, ranging from 40%-70%.

4.3 Battery measurement

As the majority of IoT devices are battery-powered, monitoring of energy source level is essential. Below some voltage value, typically 1.7V-1.8V, most microcontrollers cannot work properly. Information about the low battery state could

ICCS Camera Ready Version 2024
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-63783-4_18

prevent the device from being useless by enabling power-save modes. Remaining in the deep discharge state is also destructive for the battery itself. For this purpose, we have created μ*Chaos Battery*. This component functionality allows the simulation of an unexpected power consumption increase as a rapidly decreased battery voltage level. A sudden, increased energy consumption was simulated by reducing the read battery voltage value by 10mV, every 2 measurements, until the minimum voltage value was obtained, which for the type of battery used was approximately 1.8V.

4.4 CPU usage

To analyze the behaviour of the embedded firmware facing the high CPU utilisation in the tool, we have implemented a separate thread which, at regular intervals, performs a simple variable increment operation for approximately 300 milliseconds. Subsequently, the thread waits for the next operation for 1 second, effectively increasing the CPU load. The user can initiate and resume specific threads through the *uChaos Console* in *uChaos CPU* and has the flexibility to define the tasks of the load functions and the threads utilized. As subsequent threads loading the system were launched, the time between subsequent temperature measurements increased from 2s to 16s. The threads were then gradually disabled to decrease again the interval between measurements.

4.5 Memory consumption

Table 5 shows the size of each μChaos component. In accordance with the assumptions outlined in Section 3.2, one of the key design considerations for out library was its imperative to occupy minimal space in both ROM and RAM, given the constrained resources typically available in embedded systems. As it could be seen, due to the need to operate on strings and handle many commands, *uChaos Console* takes up most of the RAM and ROM. Following the tests conducted on the nRG52 DK board, it can be inferred that μChaos aligns with the initial assumptions. The proportion of RAM it utilizes amounts to about 2% of the total, while in terms of ROM, it accounts for below 1%.

Table 5: μChaos memory size

Component	**ROM [kB]**	**RAM [kB]**
μChaos Sensor	1,21	0,11
μChaos Battery	0,3	0,04
μChaos Console	1,58	1,32
μChaos CPU	0,27	0,05

5 Summary

The paper discusses the novel μ*Chaos* tool for Zephyr OS based embedded IoT devices. It enables chaos engineering to be applied directly to low-power battery-operated devices. In the paper we have also discussed the common types and

ICCS Camera Ready Version 2024
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-63783-4_18

sources of the failures of the devices. Finally, we have presented the scenarios showing the usage of the developed tool in sensor-based applications for IoT devices.

The proposed tool can be augmented by the edge and cloud chaos engineering tools composing the holistic failure injection solution for the IoT continuum. Further improvements should also concern other data source modalities, including audio data from the microphones for keyword spotting applications and predictive maintenance and video sources for tiny object detection and image recognition. Another possible direction of library improvement is exploring faults and disturbances in communication between devices in wireless networks like Wi-Fi, Matter, and Bluetooth. A diversity in types of networks and protocols creates interesting opportunities.

Acknowledgments. The research presented in this paper received funding from Polish Ministry of Science and Education assigned to AGH University.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Vailshery LS. Number of Internet of Things (IoT) connected devices worldwide from 2019 to 2021, with forecasts from 2022 to 2030. https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/; Noc 22, 2022, last accessed 2024/12/15
2. Butow T. Chaos Engineering: the history, principles, and practice. http://aiweb.techfak.uni-bielefeld.de/content/bworld-robot-control-software/; May, 2021, last accessed 2024/12/16
3. S. Sharieh and A. Ferworn, "Securing APIs and Chaos Engineering," 2021 IEEE Conference on Communications and Network Security (CNS), Tempe, AZ, USA, 2021, pp. 290-294, https://doi.org/10.1109/CNS53000.2021.9705049.
4. F. Poltronieri, M. Tortonesi and C. Stefanelli, "ChaosTwin: A Chaos Engineering and Digital Twin Approach for the Design of Resilient IT Services," 2021 17th International Conference on Network and Service Management (CNSM), Izmir, Turkey, 2021, pp. 234-238, https://doi.org/10.23919/CNSM52442.2021.9615519.
5. I. Zavalyshyn, T. Given–Wilson, A. Legay, R. Sadre and E. Rivière, "Chaos Duck: A Tool for Automatic IoT Software Fault-Tolerance Analysis," 2021 40th International Symposium on Reliable Distributed Systems (SRDS), Chicago, IL, USA, 2021, pp. 46-55, https://doi.org/10.1109/SRDS53918.2021.00014.
6. M. Duarte, J. P. Dias, H. S. Ferreira and A. Restivo, "Evaluation of IoT Self-healing Mechanisms using Fault-Injection in Message Brokers," 2022 IEEE/ACM 4th International Workshop on Software Engineering Research and Practices for the IoT (SERP4IoT), Pittsburgh, PA, USA, 2022, pp. 9-16, https://doi.org/10.1145/3528227.3528567.
7. A. Basiri et al., "Chaos Engineering," in IEEE Software, vol. 33, no. 3, pp. 35-41, May-June 2016, https://doi.org/10.1109/MS.2016.60.

ICCS Camera Ready Version 2024
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-63783-4_18

8. Abdelsalam Ahmed, Maher F. El-Kady, Islam Hassan, Ayman Negm, Amir Masoud Pourrahimi, Mit Muni, Ponnambalam Ravi Selvaganapathy, Richard B. Kaner, Fire-retardant, self-extinguishing triboelectric nanogenerators, Nano Energy, Volume 59, 2019, Pages 336-345, ISSN 2211-2855, https://doi.org/10.1016/j.nanoen.2019.02.026.
9. I. Priyadarshini, B. Bhola, R. Kumar and C. So-In, "A Novel Cloud Architecture for Internet of Space Things (IoST)," in IEEE Access, vol. 10, pp. 15118-15134, 2022, https://doi.org/10.1109/ACCESS.2022.3144137.
10. Dharun Anandayuvaraj and James C. Davis. 2023. Reflecting on Recurring Failures in IoT Development. In Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22). Association for Computing Machinery, New York, NY, USA, Article 185, 1–5. https://doi.org/10.1145/3551349.3559545.
11. Elham Moridi, Majid Haghparast, Mehdi Hosseinzadeh, Somaye Jafarali Jassbi, Fault management frameworks in wireless sensor networks: A survey, Computer Communications, Volume 155, 2020, Pages 205-226, ISSN 0140-3664, https://doi.org/10.1016/j.comcom.2020.03.011.
12. Adday GH, Subramaniam SK, Zukarnain ZA, Samian N. Fault Tolerance Structures in Wireless Sensor Networks (WSNs): Survey, Classification, and Future Directions. Sensors. 2022; 22(16):6041. https://doi.org/10.3390/s22166041.
13. Jesper Simonsson, Long Zhang, Brice Morin, Benoit Baudry, Martin Monperrus, Observability and chaos engineering on system calls for containerized applications in Docker, Future Generation Computer Systems, Volume 122, 2021, Pages 117-129, ISSN 0167-739X, https://doi.org/10.1016/j.future.2021.04.001.
14. Kevin Ni, Nithya Ramanathan, Mohamed Nabil Hajj Chehade, Laura Balzano, Sheela Nair, Sadaf Zahedi, Eddie Kohler, Greg Pottie, Mark Hansen, and Mani Srivastava. 2009. Sensor network data fault types. ACM Trans. Sen. Netw. 5, 3, Article 25 (May 2009), 29 pages. https://doi.org/10.1145/1525856.1525863.
15. 2015. Predictive power consumption adaptation for future generation embedded devices powered by energy harvesting sources. Microprocess. Microsyst. 39, 4 (June 2015), 250–258. https://doi.org/10.1016/j.micpro.2015.05.001.
16. Yun Wang and M. Saksena, "Scheduling fixed-priority tasks with preemption threshold," Proceedings Sixth International Conference on Real-Time Computing Systems and Applications. RTCSA'99 (Cat. No.PR00306), Hong Kong, China, 1999, pp. 328-335, https://doi.org/10.1109/RTCSA.1999.811269.
17. M. Fogli, C. Giannelli, F. Poltronieri, C. Stefanelli and M. Tortonesi, "Chaos Engineering for Resilience Assessment of Digital Twins," in IEEE Transactions on Industrial Informatics, vol. 20, no. 2, pp. 1134-1143, Feb. 2024, https://doi.org/10.1109/TII.2023.3264101.
18. M. Verma et al., "A Chaos Recommendation Tool for Reliability Testing in Large-Scale Cloud-Native Systems," 2024 16th International Conference on COMmunication Systems & NETworkS (COMSNETS), Bengaluru, India, 2024, pp. 270-272, https://doi.org/10.1109/COMSNETS59351.2024.10427311.
19. S. Oussane, H. Benkaouha and A. Djouama, "Fault Tolerance in The IoT: A Taxonomy Based on Techniques," 2023 Third International Conference on Theoretical and Applicative Aspects of Computer Science (ICTAACS), Skikda, Algeria, 2023, pp. 1-8, https://doi.org/10.1109/CTAACS60400.2023.10449571.
20. Long Zhang, Javier Ron, Benoit Baudry, and Martin Monperrus. 2023. Chaos Engineering of Ethereum Blockchain Clients. Distrib. Ledger Technol. 2, 3, Article 22 (September 2023), 18 pages, https://doi.org/10.1145/3611649.

ICCS Camera Ready Version 2024
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-63783-4_18