Detecting and Mitigating Network Packet Overloads on Real-Time Devices in IoT Systems

Robert Danicki*
r.danicki@tu-berlin.de
Technische Universität Berlin
Berlin, Germany

Martin Haug*
m.haug@tu-berlin.de
Technische Universität Berlin
Berlin, Germany

Ilja Behnke
i.behnke@tu-berlin.de
Technische Universität Berlin
Berlin, Germany

Laurenz Mädje
maedje@campus.tu-berlin.de
Technische Universität Berlin
Berlin, Germany

Lauritz Thamsen
lauritz.thamsen@tu-berlin.de
Technische Universität Berlin
Berlin, Germany

ABSTRACT

Manufacturing, automotive, and aerospace environments use embedded systems for control and automation and need to fulfill strict real-time guarantees. To facilitate more efficient business processes and remote control, such devices are being connected to IP networks. Due to the difficulty in predicting network packets and the interrelated workloads of interrupt handlers and drivers, devices controlling time critical processes stand under the risk of missing process deadlines when under high network loads. Additionally, devices at the edge of large networks and the internet are subject to a high risk of load spikes and network packet overloads.

In this paper, we investigate strategies to detect network packet overloads in real-time and present four approaches to adaptively mitigate local deadline misses. In addition to two strategies mitigating network bursts with and without hysteresis, we present and discuss two novel mitigation algorithms, called Budget and Queue Mitigation. In an experimental evaluation, all algorithms showed mitigating effects, with the Queue Mitigation strategy enabling most packet processing while preventing lateness of critical tasks.

ACM Reference Format:
Robert Danicki, Martin Haug, Ilja Behnke, Laurenz Mädje, and Lauritz Thamsen. 2021. Detecting and Mitigating Network Packet Overloads on Real-Time Devices in IoT Systems. In *4th International Workshop on Edge Systems, Analytics and Networking (EdgeSys'21), April 26, 2021, Online, United Kingdom.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3434770.3459733

1 INTRODUCTION

Safety-critical application areas like plant floors, automotive and aerospace use embedded real-time systems for control, monitoring and automation [5, 12]. With the advent of the industrial Internet of Things (IoT), industrial control systems are connected to large IP networks [9]. Receiving network traffic is accomplished by triggering interrupts for incoming network packets which preempts the process currently running on the assigned core independent of its priority. Since, however, the timing of network packets is difficult to predict, connecting the used microcontrollers with low processing power to IP networks might expose critical infrastructure to the traffic patterns of a larger network or the internet.

When a device is subject to high packet loads, local processes can be fully preempted by interrupt service routines (ISRs) and driver processes, amounting to a denial of service (DoS) whether maliciously or by network fault. In real-time scenarios this becomes especially relevant as unpredictable processing delays can have deadline breaking effects [1]. Mission critical and hard real-time devices must therefore detect high network loads and mitigate their consequences with limited processing resources.

Mitigating the effect of network-generated interrupts from software in a real-time operating system is challenging as the timing impact of mitigation techniques themselves has to be kept minimal. They are furthermore restricted to react after interrupts have already occurred as this is controlled by hardware.

Addressing this, our paper presents:

- Three metrics for detecting amounts of network traffic that may jeopardize the local real-time guarantees: The network interrupt count, receive queue fill state, as well as the lateness of critical processes.
- Four techniques to mitigate the impact of high packet loads while maintaining the network services on a best-effort basis. *Burst Mitigation* caps packets received per time slice, *Hysteresis Mitigation* puts lower and upper boundaries on earliness, the novel *Budget Mitigation* calculates how much time is left for handling interrupts, and finally, the novel *Queue Mitigation* uses the queue fill-state as an indicator for interrupt activity.
- Experiments evaluating the four mitigation techniques on the real-time operating system FreeRTOS.

Outline. Section 2 presents three detection metrics and four mitigation techniques. We then evaluate the approaches in Section 3. The results are discussed in Section 4. We give an overview of related work in Section 5, while Section 6 concludes this paper.

[*] Both authors contributed equally to this research.

EdgeSys'21, April 26, 2021, Online, United Kingdom

2 APPROACH

In the following section, we present detection techniques and mitigation algorithms to handle network-generated interrupt floods in real-time systems.

2.1 Detection

To prevent a critical task from being drowned by network interrupts, we first need to detect such a situation. There are several different direct or indirect metrics we can use to do that.

Early- or Lateness. The most direct metric is the critical task's early- or lateness. In many real-time systems, a process periodically performs a critical computation, targeting to finish it within a fixed duration. When the critical task completes this computation in time (i.e. in less time than the target duration), we have positive earliness, defined as the target duration minus the actual time spent. When, however, the critical task exceeds its target duration, it incurs lateness, defined as how much longer it took than targeted. We will express lateness as a ratio of the critical task's target duration, e. g. 100% lateness means it took twice as long as intended.

This metric very directly corresponds to what we are trying to detect but also has a few drawbacks: For once, it introduces latency as processes can only report earliness/lateness once per task cycle. Thus, mitigation techniques might need to be overly cautious (disabling networking while there is still some earliness) because otherwise, it will react too late, when lateness already crept in. Additionally, there is the more practical concern that the metric may be hard to come by in real systems since somehow, the metric must be reported. We may thus call mitigation techniques relying on this *cooperative.*

Network Interrupt Count. A less direct metric is the number of incoming interrupts, discretized by dividing time into fixed time slices and counting network interrupts in them.

The number of interrupts can be easily counted since Interrupt Service Routines (ISRs) often run custom code anyway. Since many network interrupts occur per time slice, the metric's resolution is equally high. Timing precision is not crucial because misattributing the first few packets to the passed time slice does not introduce significant errors.

The drawback of this metric is that it only correlates with the situations we are trying to prevent if certain preconditions are true: The network interrupt count shows approximately how much strain the interrupts are putting on a CPU. We can thus estimate overall system resource usage if the resource requirements of the critical task stay approximately the same for each of its cycles. The first precondition, therefore, is that the critical task has to require the same amount of resources at all times — mitigation techniques relying on this can not react elastically to load change. Furthermore, this metric assumes that the processing of each packet takes about the same time. Mitigation techniques using this metric can only be effective if the ensemble of incoming packets is homogeneous enough such that the assumption that each packet takes approximately the same time to process is either true or practical because of the regular distribution of packet response time.

Network Receive Queue Fill State. A third possible metric is detecting if the queue of received packets to be processed by the network driver is full or not. The queue fill state shows whether the network driver can keep pace with the incoming packets. If more packets arrive than the network driver can handle, the queue will fill up until it reaches maximum capacity; it would empty in the opposite case.

Of course, the queue's capacity impacts the quality of this metric. With a queue capacity that is too low, the metric will show saturation even for short bursts of packets that are not representative of the overall traffic pace. Low queue sizes will also lead to saturation if the scheduler did not wake up the network driver for some time. If, conversely, the queue is too large it acts as a cushion and will delay alerting by filling up, allowing the network routines to stay activated for longer than is prudent.

The metric scales well in terms of packet response time deviation and elastic critical loads because it directly mirrors the ability of the network driver to process incoming traffic. It has to be coupled with a mechanism like scheduler priority that moderates network driver execution such that an appropriate resource ceiling is found for the tasks.

2.2 Mitigation Techniques

In the following, we present four different mitigation techniques. With the *Burst Mitigation*, we suggest a technique that prevents unresponsiveness in case of short packet bursts. The *Hysteresis Mitigation* controls the networking tasks based on the early- or lateness of the critical task. This idea taken further, the novel *Budget Mitigation* attempts to balance out the networking part and the critical task by assigning a networking budget to the driver that is derived from the critical task's earliness. Finally, the novel *Queue Mitigation* takes a step back and explores a different way to detect high load, enabling a simple yet effective mitigation technique.

Burst Mitigation. Burst Mitigation is a simple approach to deal with very high packet loads in short periods. Conceptually, time is split into fixed time slices, each having a fixed maximum packet capacity. The network ISR tracks these time slices by querying the operating system for the tick count each time it is invoked, starting a new slice if enough time has passed since the start of the last one. In each slice, the number of packets is counted. If this packet count surpasses the fixed capacity, network interrupts are disabled until the start of the next time slice. We visualize this in Figure 1: The length of each time slice is $20ms$ and the curve plots the number of received packets in each slice. The red zone on top is off-limits — when 600 packets are reached, interrupts are disabled.

Technically, the maximum number of packets that are accepted in a very short amount of time may be up to twice as large as the capacity limit. To see this, consider the following case: When *capacity*-many packets arrive directly at the end of one slice, the counter is reset immediately after that burst, allowing another burst of up to *capacity*-many packets. However, since the limit must be tuned manually anyway, this is not important conceptually.

Hysteresis Mitigation. When the network load exceeds the trigger capacity for the Burst Mitigation by a small amount, network interrupt activation will oscillate quite a bit. We used the number of

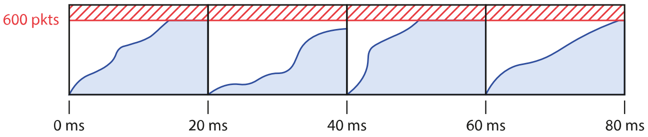


Figure 1: Burst Mitigation: Visualization of time slices.

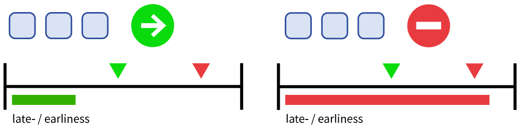


Figure 2: Hysteresis Mitigation: Late- / earliness on a scale with the block and unblock thresholds.

packets per time frame to detect whether we had to act to maintain the local real-time guarantees. This, however, is just an imperfect proxy for the metric we are actually interested in: The critical task's performance, i. e. the lateness of the critical task. Therefore, we based this mitigation on it.

Hysteresis is a popular technique for controlling processes [7]. It defines two thresholds: A maximum allowable threshold for a metric above which whatever action contributing to the rise of that metric will be ceased and a minimum threshold below which the action will be re-started. We use the lateness/earliness metric reported by the critical task, i.e. how much time is left in the time slice when the task has been accomplished as the hysteresis control metric. Once the earliness falls below the minimum allowable value, we stop processing new packets in the network driver, deactivate interrupts from there and wait (in a loop that sleeps for some time in every iteration) for the critical task to report an earliness higher than the threshold.

Budget Mitigation. This novel mitigation technique ties the earliness metric more closely to the amount of work that is permissible within the network subsystems. For example, there could be a local critical task load that uses up 90% of computation time per cycle. Even a minuscule network load could then lead to the Hysteresis Mitigation permanently disabling the network. Furthermore, we wanted to use the late-/earliness metric to react more responsively to elastic loads.

Budget Mitigation is another cooperative approach. The critical task reports its earliness to the network driver after each completed cycle of computation. This earliness is interpreted as the time budget of the network driver. The driver will measure the time of its operations and subtract that from the latest budget. Once the budget is depleted, the network subsystems will be suspended (including the interrupts) until a new earliness notification is issued. For this mitigation to work, we set both the critical and the network driver task to equal priorities such that the network driver has a chance to deplete its budget after which it will actively yield to the critical task. This is an important tweak since the Budget Mitigation acts as a specialized scheduler by deciding when the network subsystem has to cease to operate. It would otherwise be defeated by the scheduler's time slice logic.

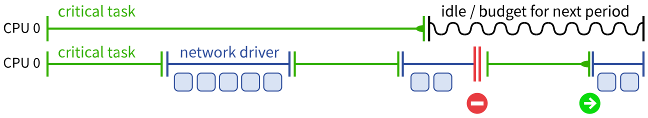


Figure 3: Budget Mitigation: Timelines of tasks on CPU. The network driver can preempt the critical task until its budget is depleted.

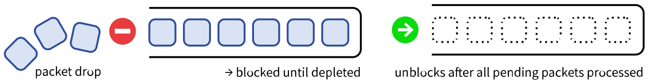


Figure 4: Queue Mitigation: After the packet queue fills up, all incoming packets are dropped until it is empty again.

Queue Mitigation. We looked at three mitigation techniques so far — a very simple approach that requires a manual definition of a capacity limit and two cooperative ones that require communication with the critical task. With Queue Mitigation, we propose a new, more universal, yet effective non-cooperative mitigation.

This mitigation technique is based on the simple observation with regards to the network queue we made in Section 2.1: The network driver can keep up with the traffic when the packet queue is not full. Thus, with queue mitigation, we simply disable network interrupts if they fail to put a packet into the queue (because it was already full) and only re-enable the interrupts from the driver once it has processed all queued packets.

When using this mitigation technique, we assign a higher priority to the critical task. As a result, the network driver is the first process running out of time when CPU resources get scarce or the interrupt frequency rises. The direct effect is that the queue fills up. With a reasonable queue size, the networking is then disabled before the critical task is too stressed, thus protecting the critical task from interrupt overload.

3 EVALUATION

To investigate the effectiveness of the presented metrics and techniques, we test on the ESP32 microcontroller. The controller is a common ARM-based development board with two CPU cores and the lightweight FreeRTOS operating system. FreeRTOS provides basic task scheduler and interrupt management as well as some data structures for our application. The multi-core platform allows us to separate the traffic generator from the traffic consumer by placing each on one of the two cores. A local task simulating time-critical computation is additionally placed on the consuming core such that both compete for CPU time.

In our test setup, the interrupt handler fills up a FreeRTOS queue while the network driver empties it, thus processing the incoming data. This allows us to analyze the effect of both, the interrupts and the driver on the critical computation load.

The traffic generator running on the second core generates network loads with a pyramid-shaped load pattern. It sets a GPIO pin to high for each received packet, triggering a GPIO interrupt on the other core over a GPIO bridge. There, the network simulator synthesizes a TCP SYN packet and places it into a queue for consumption by the network driver which acknowledges the packet's arrival. Meanwhile, the observed critical task (also running on the first core) calculates an ascending series of binomial coefficients to generate an equal work load for each task cycle. Its goal is to reach a target (n, k) in a small time frame of $10ms$. If the target is reached before the permissible time expired, the critical task will sleep the remaining time and start over in the next period. If, however, the critical task fails to reach its target in time, it will accumulate lateness, i.e. continue until it reached the target values and then start its new period immediately. Figure 5 provides a graphical overview of this setup.

The amount of

- interrupts triggered on the second core
- interrupts executed on the first core
- packets processed in the network driver
- critical task cycles
- accumulated critical task lateness

are saved into atomic variables of a monitoring routine running on core 1. Every second, the controller reports these variables via the serial interface and then clears the counters.

The design imposes a few limitations: First, it does not account for interrupt sources other than network interrupts. However, in high network load situations, network interrupts should significantly outnumber other interrupts such that this is not too much of a problem (for reference, in a typical FreeRTOS system there are 1̃000 background interrupts per second to control the scheduler, in contrast to 1̃00,000 network interrupts in high-load scenarios [4]).

Secondly, the critical task may only use one core since we need one of the ESP32's cores for traffic generation and analysis. We believe that this is not a significant drawback since most current microcontrollers remain single-cored [3].

Scheduler. Before diving into custom mitigation techniques, we take a look at the built-in FreeRTOS-Scheduler as a baseline. The option to assign (different) priorities to the critical task and network driver is a first, simple way to balance them out.

As a baseline, we tested how the system performs when both tasks are assigned equal priority. As expected, the critical task started to incur significant lateness (up to 500%), while the network driver was able to process almost all packets. Next, we tested a configuration where the critical task has higher priority than the network driver. Interestingly, this alone brought the lateness down to almost zero.

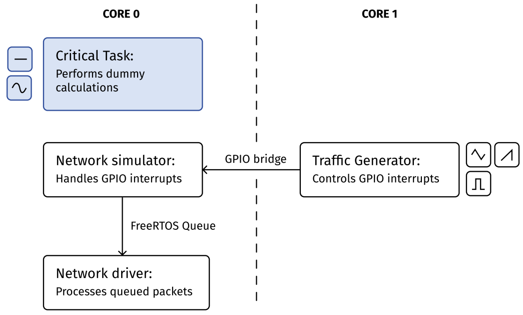
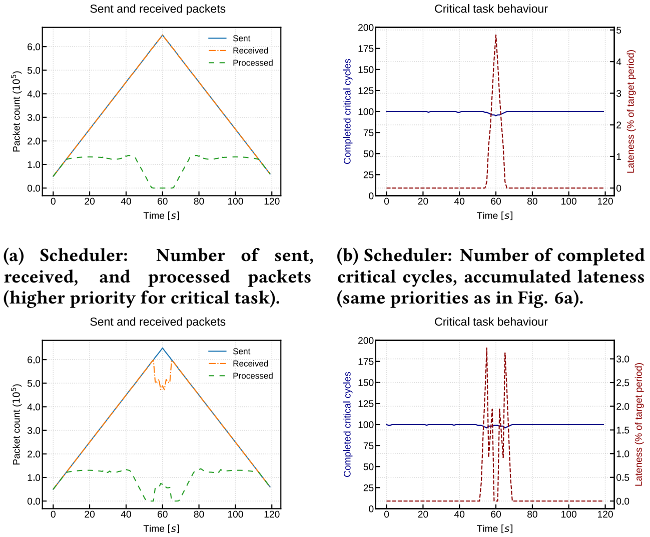


Figure 5: Setup used to simulate different interrupt load scenarios on the ESP32 SoC.



(a) Scheduler: Number of sent, received, and processed packets (higher priority for critical task).

(b) Scheduler: Number of completed critical cycles, accumulated lateness (same priorities as in Fig. 6a).

(c) Burst Mitigation: Number of sent, received, and processed packets (capacity of 600 packets per $20ms$, higher priority for critical task, queue size 100).

(d) Burst Mitigation: Number of completed critical cycles, accumulated lateness (same parameters as in Fig. 6c).

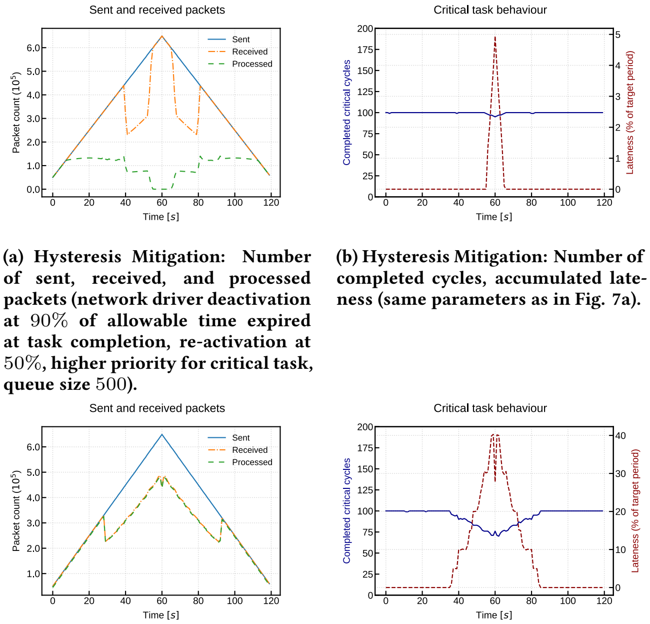
Figure 6: Performance of the scheduler and Burst Mitigation

A slight lateness of about 8% is observed when the number of packets per second exceeds $50,000$, cf. Fig. 6b). At the same time, the number of packets processed by the driver is far lower than previously, dropping even more as the interrupt count rises (cf. Fig. 6a).

Burst Mitigation. We evaluated Burst Mitigation with a capacity of 600 packets per $20ms$. We assigned a higher scheduler priority to the critical task since even though we disable the network interrupts, the network driver still processes queued packets. By setting the priorities in favor of the critical task, we inhibit not only the ISR but also the driver from taking too much time off the critical task.

Figure 6c shows the number of sent, received, and processed packets we measured with the stated parameters. The number of received packets per second does not exceed $30,000$, as is expected from 600 packets per $20ms$. In this experiment, the critical task did not introduce lateness, demonstrating the effectiveness of the mitigation when the parameter value is well chosen. In further tests, we measured that raising the packet capacity significantly leads to lateness, showing that $30,000$ is the optimal condition for our setup.

Hysteresis Mitigation. The network driver's throughput does not stabilize as it does with the Burst Mitigation but instead plummets as the increased interrupt count puts sustained load on the CPU core Figure (cf. 7a). This does not meet expectations as interrupts should be turned off together with the processing in the network driver by the mitigation algorithm. It turns out that the interrupts, once reactivated, will drown out the network driver



(a) Hysteresis Mitigation: Number of sent, received, and processed packets (network driver deactivation at 90% of allowable time expired at task completion, re-activation at 50%, higher priority for critical task, queue size 500).

(b) Hysteresis Mitigation: Number of completed cycles, accumulated lateness (same parameters as in Fig. 7a).

(c) Budget Mitigation: Number of sent, received, and processed packets (both tasks have equal priority, queue size 100).

(d) Budget Mitigation: Number of completed critical cycles, accumulated lateness (same parameters as in Fig. 7c).

Figure 7: Performance of Hysteresis and Budget Mitigation

where the thresholds are checked. Once the network driver is permanently preempted by ISR executions, the mitigation algorithm can no longer take into effect, explaining why the interrupt count curve starts to fit the sent packet curve again. Nevertheless, Figure 7b shows that the Hysteresis Mitigation is quite effective in preventing lateness except when the load of interrupts itself throttles network driver execution.

Budget Mitigation. Figure 7c shows that once the budget is depleted for the first time, there is a drop in received and processed packets, beyond which both curves continue to track the trend of sent packets. The slope of the received/processed curves is less steep than that of the sent packets, suggesting that the budget has a moderating impact but cannot lower the network subsystem activity enough for the real-time guarantees to be maintained. For each additional packet received, 0.76 additional packets are processed.

Figure 7d shows that the overestimation of the budget per additional incoming packet leads to eventually breaking the real-time guarantees for high loads.

Queue Mitigation. We evaluated the Queue Mitigation with different queue sizes. Figure 8 depicts the packet numbers with a queue size of 500. While the network stack is not able to cope with all incoming packets once they rise above $30,000$ per second, the processed packets nicely trace the received packets.

At the same time, the critical task incurred no lateness in this setup. Queue Mitigation was thus able to process far more packets than Burst and Hysteresis Mitigation while protecting the critical task more effectively.

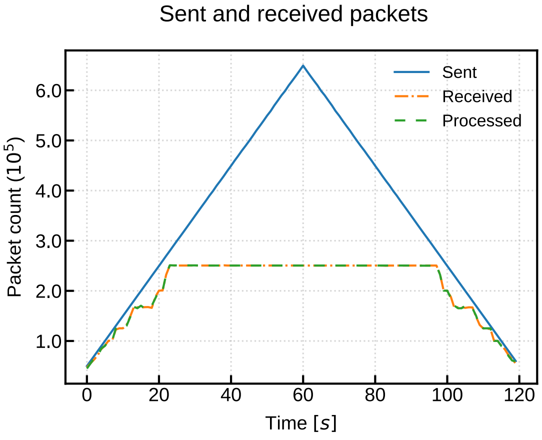


Figure 8: Queue Mitigation: Number of sent, received, and processed packets (higher priority for critical task, queue size 500). The lateness remained at zero.

We also tested smaller and larger queue sizes. Reducing the queue size to 100 diminished packet throughput, but added no lateness. This is because a smaller queue makes the mitigation more cautious as the queue fills up faster. We observed the opposite effect with a queue size of 750. Lateness started to emerge because the mitigation reacted belatedly.

4 DISCUSSION

Using only the scheduler, no priority configuration led to particularly good results—with either lateness or very little packet throughput. Additionally, its priorities cannot mitigate the problem of *interrupts* drowning the process, since ISRs always run above the process priority space. However, the scheduler priorities can still aid actual mitigation techniques in balancing out the critical task and network driver, as we will show later. Burst Mitigation allows to avoid drowning all other computation in interrupts, but depending on how the threshold was chosen, much time is spent in the ISR receiving packets that cannot be processed in time.

Hysteresis Mitigation can be effective in environments without network interrupt loads able to drown the network driver task. In a high-load environment, it has to be coupled with an interrupt-reducing mechanism like Burst Mitigation in the ISR, which removes some of the advantages of stand-alone usage tested above.

Unlike the Burst Mitigation, the mechanism does not require any knowledge about system throughput – the required threshold constants generalize better across platforms with differences in processing power and are more closely aligned with the goal of minimizing lateness.

The Budget Mitigation, not unlike the FreeRTOS scheduler, accumulates more lateness the higher the load, the link is approximately linear. This is due to its underestimation of packet processing time mentioned in Sec. 3. The behavior can partially be explained by the fact that only the activity in the driver and not in the ISR is timed due to implementation difficulties. The ISR activity thus does not impact budget consumption.

The characteristics of earliness as a reporting mechanism are also crucial: Our critical task reports the difference of the timestamps between cycle termination and cycle target, *not* the time spent on the critical task alone. If, therefore, the network driver has been allocated a large budget in one cycle, it will compete with the critical task for processing time, leading to it closely matching its deadline, and reporting little earliness. The critical task can then terminate early the next cycle because the network task had a small budget, introducing oscillations in the budget ceiling.

Queue Mitigation always keeps the critical task on time while the packet throughput remains at a constant level. The number of packets received by the ISR and processed by the driver is almost equal. This indicates that interrupt handling and processing in the driver are well-balanced.

Alas, with Queue Mitigation there is still a hyper-parameter we need to tune, as with Burst Mitigation. However, we argue that the queue size is a parameter that must be defined in any case, with any mitigation, and as argued in Section 2.1 defining a fixed queue length still leaves the system more flexible and elastic than defining a fixed burst capacity.

5 RELATED WORK

The authors of [10] present rate-limiting schedulers (both in software and hardware) and a burst scheduler, which is very similar to our Burst Mitigation. However, in comparison to our mitigation techniques, their approach is not specifically tuned to network interrupts and as such cannot use metrics like the queue fill state.

Many approaches to solving the problem of high interrupt counts breaking real-time priorities include extending the system with additional hardware. The Peripheral Control Processor is a proposed co-processor that executes interrupts and remaps priorities to unify the priority space between tasks and interrupts [11].

FPGA hardware monitoring and controlling the interrupt's execution of a connected microcontroller [13] can be a viable strategy. Although the proposed soft- and hardware interrupt limiters perform very well, the downside is the increased system cost.

To analyse real-time behaviour under different network loads and hardware configurations [2] presents a playground for network interrupt experiments in IoT environments. The tool allows to run experiments on real-time embedded systems with different network interface controller implementations, load generators and timing utilities.

The approach of [8] simulates control and background traffic to a hypothetical critical power plant control system that is exposed to the internet and as such a target of DoS attacks. The paper presents a interrupt-overload detection. However, its mitigation techniques have the same limitations as our Burst and Hysteresis mitigation. The high processing power requirements are addressed by putting a more powerful router between the network and the embedded devices.

The router strategy is extended by [14]. The paper demonstrates how to use a software-defined network architecture for edge DDoS protection, possibly leveraging infrastructure that is already in place.

Detection can be refined by using LSTMs and CNNs for traffic classification [6]. Here, packets are passed through the closest edge server for execution of the models.

6 CONCLUSION

The trend of putting embedded devices at the edge of the network to perform critical tasks can expose the whole system to risk. The edge devices may fail to hold up to a sudden surge in traffic or be targeted by denial of service attacks.

We analyzed the network interrupt count and queue fill state as well as the lateness of a critical task running on an embedded SoC while it was flooded with network traffic. We developed four mitigation strategies to maintain smooth operation and timeliness of a critical process using signals derived from the data obtained during the analysis: Burst Mitigation, Hysteresis Mitigation, Budget Mitigation, and finally, Queue Mitigation.

In our experiments, we measured the quality of the different mitigation strategies through the lateness of a critical task that runs concurrently to a network simulation triggering a high amount of interrupts. An evaluation has shown that Queue Mitigation performed the best since it receives and processes the most network packets while also protecting the critical task from lateness.

REFERENCES

[1] Ilja Behnke, Lukas Pirl, Lauritz Thamsen, Robert Danicki, Andreas Polze, and Kao Odej. 2020. Interrupting Real-Time IoT Tasks: How Bad Can It Be to Connect Your Critical Embedded System to the Internet?. In *Proceedings of the 39th IEEE International Performance Computing and Communications Conference*. in press.

[2] Franz Bender, Jan Jonas Brune, Nick Lauritz Keutel, Ilja Behnke, and Lauritz Thamsen. 2021. PIERES: A Playground for Network Interrupt Experiments on Real-Time Embedded Systems in the IoT. In *International Conference on Performance Engineering Companion (LTB '21)*. ACM/SPEC, in press.

[3] Alessio Bucaioni, Saad Mubeen, Federico Ciccozzi, Antonio Cicchetti, and Mikael Sjödin. 2017. Technology-Preserving Transition from Single-Core to Multi-Core in Modelling Vehicular Systems. In *Modelling Foundations and Applications* (Cham) *(Lecture Notes in Computer Science)*, Anthony Anjorin and Huáscar Espinoza (Eds.). Springer, 285–299.

[4] Rich Goyette. 2007. *An Analysis and Description of the Inner Workings of the FreeRTOS Kernel.* Technical Report. Carleton University. 46 pages.

[5] K. Hanninen, J. Maki-Turja, and M. Nolin. 2006. Present and Future Requirements in Developing Industrial Embedded Real-Time Systems - Interviews with Designers in the Vehicle Domain. In *13th Annual IEEE International Symposium and Workshop on Engineering of Computer-Based Systems (ECBS'06)*. 9 pp.–150.

[6] Yizhen Jia, Fangtian Zhong, Arwa Alrawais, Bei Gong, and Xiuzhen Cheng. 2020. FlowGuard: An Intelligent Edge Defense Mechanism Against IoT DDoS Attacks. *IEEE Internet of Things Journal* 7, 10 (2020), 9552–9562. https://doi.org/10.1109/JIOT.2020.2993782

[7] Mark A. Krasnosel'skiĭ and Aleksei V. Pokrovskiĭ. 1989. Static Hysteron. In *Systems with Hysteresis*, Mark A. Krasnosel'skiĭ and Aleksei V. Pokrovskiĭ (Eds.). Springer, 1–58.

[8] M. Long, Chwan-Hwa Wu, and J. Y. Hung. 2005. Denial of service attacks on network-based control systems: impact and mitigation. *IEEE Transactions on Industrial Informatics* 1, 2 (05 2005), 85–96.

[9] Ariana Mirian, Zane Ma, David Adrian, Matthew Tischer, Thasphon Chuenchujit, Tim Yardley, Robin Berthier, Joshua Mason, Zakir Durumeric, J Alex Halderman, et al. 2016. An internet-wide view of ics devices. In *14th Annual Conference on Privacy, Security and Trust (PST)*. 96–103.

[10] John Regehr and Usit Duongsaa. 2005. Preventing Interrupt Overload. 40, 7 (15 06 2005), 50–58.

[11] Fabian Scheler, Wanja Hofer, Benjamin Oechslein, Rudi Pfister, Wolfgang Schröder-Preikschat, and Daniel Lohmann. 2009. Parallel, hardware-supported interrupt handling in an event-triggered real-time operating system. 167–174.

[12] David C. Sharp, Alex E. Bell, Jeffrey J. Gold, Ken W. Gibbar, Dennis W. Gvillo, Vann M. Knight, Kevin P. Murphy, Wendy C. Roll, Radhakrishna G. Sampigethaya, Viswa Santhanam, and Steven P. Weismuller. 2010. Challenges and Solutions for Embedded and Networked Aerospace Software Systems. 98, 4 (04 2010), 621–634.

[13] J. Strnadel. 2012. Monitoring-driven HW/SW interrupt overload prevention for embedded real-time systems. In *2012 IEEE 15th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*. 121–126.

[14] Azka Wani and S. Revathi. 2020. DDoS Detection and Alleviation in IoT Using SDN (SDIoT-DDoS-DA). 101, 2 (01 04 2020), 117–128. https://doi.org/10.1007/s40031-020-00442-z