Consensual Resilient Control: Stateless Recovery of Stateful Controllers

Aleksandar Matovic
Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg, Luxembourg

Rafal Graczyk
Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg, Luxembourg

Federico Lucchetti
Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg, Luxembourg

Marcus Völp
Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg, Luxembourg

Abstract

Safety-critical systems have to absorb accidental and malicious faults to obtain high mean-times-to-failures (MTTFs). Traditionally, this is achieved through re-execution or replication. However, both techniques come with significant overheads, in particular when cold-start effects are considered. Such effects occur after replicas resume from checkpoints or from their initial state. This work aims at improving on the performance of control-task replication by leveraging an inherent stability of many plants to tolerate occasional control-task deadline misses and suggests masking faults just with a detection quorum. To make this possible, we have to eliminate cold-start effects to allow replicas to rejuvenate during each control cycle. We do so, by systematically turning stateful controllers into instants that can be recovered in a stateless manner. We highlight the mechanisms behind this transformation, how it achieves consensual resilient control, and demonstrate on the example of an inverted pendulum how accidental and maliciously-induced faults can be absorbed, even if control tasks run in less predictable environments.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Computer systems organization → Embedded and cyber-physical systems; Computer systems organization → Dependable and fault-tolerant systems and networks

Keywords and phrases resilience, control, replication

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2023.14

Funding This work is supported by the European Commission through H2020 grant 871259 – ADMORPH.

Acknowledgements Thanks to the anonymous reviewers and shepherd for their fruitful comments and suggestions how to improve this paper. A special thanks goes to Martina Maggio and to Filip Markovic for their helpful feedback and advice.

1 Introduction

Safety-critical systems used to be closed systems built from highly predictable components and with accidental fault tolerance obtained through triple-modular redundancy (TMR) [37] (e.g., in the time-triggered architecture (TTA) [31]). Although some systems continue to be built along these lines, real-time systems, in general, became more open, networked, functionally richer and dynamic and, as such, also more susceptible to accidental faults and targeted attacks. Cyberattacks are a reality for real-time as well as safety-critical systems [34, 11, 58, 69, 36, 70, 28, 56, 49, 53, 41].

Fault and intrusion detection, paired with a mechanism to recover and re-execute faulty tasks (see, e.g., Zou et al. [73]), as well as fault-masking through voting has been proposed as application-agnostic techniques to mitigate accidental and intentionally-induced malicious faults. However, these techniques come at high costs, in particular, due to cold-start effects when running recovered tasks from their initial state or from a checkpoint. As control systems become more complex, we observe recovery effects become more prominent. For example, stopping and restarting (from its initial state) the perception module of an autonomous driving stack may well lead to cold-start effects that require the vehicle to stop for several seconds before environmental perception gets restored[1].

In this paper, we address the performance problems of recovering tasks from a cold state to allow them to rejuvenate each time the control task is invoked. We utilize this possibility to rejuvenate to operate control with a quorum that is just large enough to detect faults. We then leverage recent results from Maggio et al. [39] and Vreman et al. [65], which state conditions under which a controlled system can tolerate missing up to m subsequent actuations, to reach consensus over time. More precisely, in case the detection quorum is not able to reach consensus immediately (which is the case if a fault manifests in a disagreement of votes), we rejuvenate and re-execute control task replicas in the subsequent control periods – which we call *epochs*. Rejuvenated tasks re-execute the original problem (i.e., sensor inputs and state) to collect over up to k epochs the matching proposals we need to reach consensus. We do so while ensuring k is bounded from above by the missable deadlines (i.e., $k \leq m$).

More precisely, Maggio et al. [39] identified an inherent stability of many plants that allows them to tolerate several deadline misses in a row without losing said stability, provided no wrong actuator signal reaches the plant. Vreman et al. [65], further found that an even larger number of deadline misses can be tolerated, provided the controller enters a subsequent no-miss phase in which deadlines can be guaranteed to be met. Whereas the first result allows operating the controller just with a detection quorum, reaching an agreement over time, the latter gives rise to adjust the system's resilience by switching from a detection to a masking quorum, by adjusting its resilience to adapt to more critical failures [57] or by engaging in more elaborate recovery actions.

The prerequisite for applying any of these techniques is the system's ability to recover faulty replicas extremely fast to allow rejuvenating them after each invocation. Naive recovery would require creating a new instance of the control task, bringing it up to speed with the state of its peers (e.g., by resuming it from a checkpoint and by replaying previous requests), and configuring its privileges to participate in the consensus decisions instead of the faulty task it replaces. The costs of these operations are high and challenging to bind from above. In other words, such a recovery method is not suitable to be applied in between any two invocations of the control task. Imagine instead the task would be stateless in the sense of observing all required information by reading out the plant's sensors. It would need to maintain no other state from one invocation to another. Then, rejuvenation would amount to a trivial reset of the task, its control flow and stack to the beginning of its control loop. Unfortunately, most control tasks are not stateless and even seemingly stateless control algorithms, such as the Linear Quadratic Regulator (LQR), may become stateful in case, not all values can be directly observed from the plant.

This paper demonstrates how stateful tasks can be systematically transformed into instances that can be recovered like stateless ones. This way, recovery becomes fast enough to be executed before every invocation. Moreover, we show how consensual memory [19] helps protect any state that needs to be maintained across control-task invocations.

[1] Observation from injecting crash faults into Apollo's perception system [14].

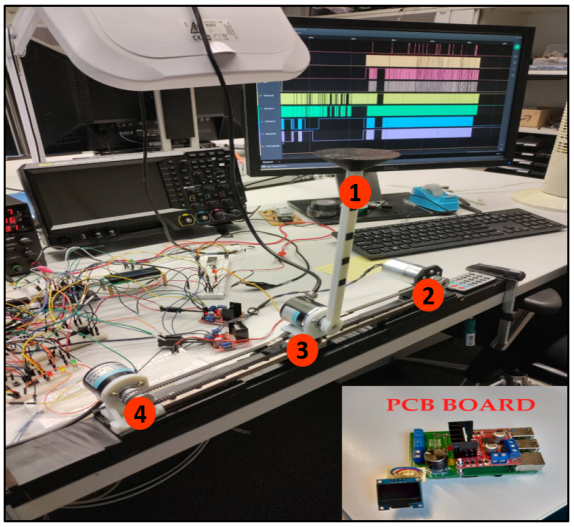
We make the following contributions:

- We introduce in Section 5 our Consensual Resilient Control (CRC) approach, capable of masking up to f accidental faults every control period T (which we call *epoch*), including some maliciously-induced faults as detailed in our system and fault model in Section 4.
- To achieve this, we systematically convert stateful control tasks into statelessly recoverable instances (see Section 5.1), which we execute in a replicated manner, but just with a detection quorum of $n = f + 1$ replicas.
- We equip replicas with a means to revert to the current and previous plant state (see Section 5.2) and utilize the latter to reach consensus, in case this was not possible in a single epoch.
- We provide them with a trusted voter, which is simple enough to allow hardware or FPGA implementations, but more importantly, which can be brought to a zero defect target. The voter is used to actuate the plant after agreement has been reached, but also to store data in consensual memory if this information needs to be carried across control-task invocations (see Section 5.3). An FPGA implementation is not part of this work.
- And, we evaluate the performance of our approach against a non-resilient (singleton) version of the control task, as well as against a classical Triple Modular Redundancy (TMR) setup. To prevent adversaries from exhausting the healthy majority that TMR needs to mask faults, we consider also the proactive rejuvenation of replicas that participate in TMR (e.g., by operating with $n = 4$ replicas to tolerate one fault, even when one of the four replicas is rejuvenated). We restart this replica from its initial state.

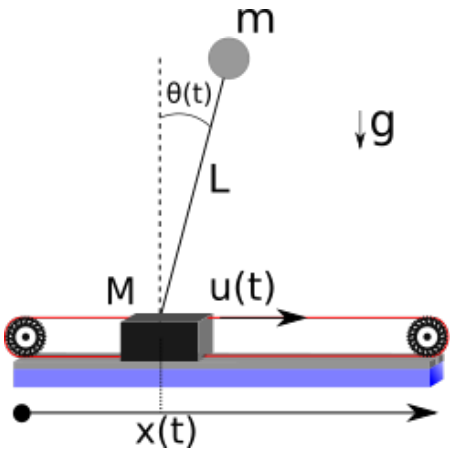
This paper is organized as follows: in Section 2, we present our evaluation vehicle and running example (an inverted pendulum). We continue by relating our work to the works of others (in Section 3) and by formally introducing our system and fault model (in Section 4). In this work, we focus exclusively on faults pertaining to the execution of control tasks. We assume (possibly fused) sensor data to be correct. In the future, we will then investigate the interplay of resilience mechanisms for internal faults with measures, such as Choi et al.[12], to cope with sensor failures.

2 Running Example: Inverted Pendulum

The inverted pendulum serves today as the text-book example in control-theory [3]. It lies at the heart of many control theory problems ranging from self-balancing hover-boards to stabilizing rocket propulsion systems. Indeed the inverted pendulum is investigated throughout the scientific literature as a minimum-viable benchmark to study a myriad of control problems present in neural-network based controllers [67], complex-simplex control systems [42], and works in the real-time systems domain [54]. By virtue of its simplicity, we chose the one-dimensional inverted pendulum model as the running example throughout this paper. We evaluated our Consensual Resilient Control approach by using a custom-made design (shown in Figure 1(a)). The pendulum consists of a moving mass (M) articulating another mass (m) through a free falling rigid rod (of length L) along a one-dimensional axis. In this work, we shall use the pendulum as an example to illustrate how our novel CRC approach can mask faults in stateful controllers with just $n = f + 1$ replicas, which normally allow only detecting the presence of a wrong actuation. For that, we restrict ourselves to the linear regime where the inverted pendulum (in the literature referred to as *the plant*) state $h(t)$ is proximal at any moment t to its stable point i.e. $h(t) = [x(t), \dot{x}(t), \theta(t), \dot{\theta}(t)] \rightarrow h_s = [x_0, 0, 0, 0]$. Here x is the position of the pendulum along the one-dimensional axis, $\dot{x}$ its linear velocity, θ



(a) Custom-made inverted pendulum breadboard-based setup and the PCB board that replaces it in the most recent version. The pendulum uses a 12V DC motor coupled with rotary encoders to measure position and angle.



(b) Pendulum schematics and parameters that govern the pendulum's equations of motion.

Figure 1 Experimental setup of our inverted pendulum. A motor (2) provides a force u via a belt on a rail-sliding mass M whose position along the x axis is determined by a rotatory encoder (4). A secondary rotatory encoder (3) measures the angle θ of the rod (1) which has a length L.

its time-varying angle with-respect to the vertical axis and $\dot{\theta}$ the associated angular speed. In this stability region and in the presence of a feedback control $u(t)$ the fully non-linear equation of motion approximately linearize and are given by $\ddot{\theta} = \frac{1}{L}\big(g\cos\theta(t) - u(t)\sin\theta(t)\big)$. The task of every feedback control system is to keep the state of the plant close to the stable point by first sensing its current state and subsequently imparting a counter balancing force u, in our case to the sliding mass M. Various control algorithms exist, the most common ones used in order to stabilize the inverted pendulum are the Linear-Quadratic-Regulator (LQR) and the Proportional–Integral–Derivative controller (PID).

LQR implements the control task by feeding back a force which is proportional to the error of the current state with respect to the stable point such that $u_k = -\mathbf{K}\cdot\delta h_k$, where $\delta h = h_k - h_s$ and $\mathbf{K}$ is a matrix of constant weights that are fine-tuned as a function of the plant's dynamical properties.

PID takes a similar approach by adding two additional terms to the proportional term of LQR $u_k = \mathbf{K}_p\cdot\delta h_k + \mathbf{K}_i\cdot\sum_{m=k-l}^{k}\delta h_m + \mathbf{K}_d\cdot\frac{d\delta h_k}{dk}$ where the integral term accumulates the $k-m$ historic errors and the derivative term determines how fast the stable point is reached[2]. One striking difference between PID and LQR is that the former needs to keep track of the historic states in order to compute the integral term and is therefore referred to as a stateful controller as opposed to LQR being a stateless controller. However, given the technical specifications of our measurement devices[3], instantaneous measures such as

[2] We refer the reader to the corresponding control literature for further information on how to tune the PID gains and K matrix for LQR.

[3] Position and angle are sensed through two rotary encoders, which detect changes of the encoders' rotation angle as quadratically shifted square waves in two channels. That is, angular changes are reported as raising and falling edges of the square waves, whereby the shift between channels indicates the direction of rotation.

linear and angular speed are not immediately available for a given epoch but have to be indirectly inferred through first recording past positions and angles and then computing the temporal variation of the latter. Consequently, LQR becomes effectively stateful. Formally for a given epoch k the forward function that models the control feedback loop can be written as $f(h_{k-l}, ..., h_k) = u_k$ where for LQR $l = 1$ and PID $l \geq 1$. The necessary state that the controller needs to keep track of (h) would allow us to turn an effectively stateful controller into a stateless recoverable instance (see Section 5.1).

For PID and LQR of an inverted pendulum, this state is trivially small for modern computational devices as just a couple of variables are saved across invocations. However, for this work, the actual algorithms are not relevant. We must observe that over the years, several increasingly sophisticated control algorithms have been proposed to cope with increasing plant complexities, including Model Predictive Controllers (MPC). One glaring example is the electric microgrid, as exemplified by Huo et al. [27] where MPC optimizes the energy generation and storage decisions based on a state as large as 420KB (at each step).

On the other hand, there is a theoretical possibility to optimize an MPC algorithm implementation on a 43KB-limited microcontroller [40], using techniques that enable satisfactory control performance while respecting memory constraints.

3 Related Work

To the best knowledge of the authors, this is the first work to leverage application-specific knowledge to optimize the more general resilience problem of tolerating up to f simultaneous faults over extended periods of time (from at least $n = 2f + 1$ replicas – plus potentially additional replicas to compensate for unavailable ones during rejuvenation – to $n = f + 1$ replicas). Several works contribute as individual building blocks for this work, which we review in the following.

Hard and weakly hard real-time systems. Traditionally hard-real time systems consider deadline misses fatal as they put safety at risk. However, this is not generally true. Weakly-hard real-time systems [7] characterize systems by the number of deadlines that can be missed during any given time window. The $m - K$ model[4] [2, 18, 23] allows up to m deadlines to be missed among any K consecutive jobs of the task. In control, the parameters m and K can be derived when analyzing the inherent stability of plants [39, 65]. In this work, we leverage these results to operate under a detection quorum, respectively, in reduced tolerance settings until the system can be adapted to mask faults immediately.

Fault tolerance and recovery. The Time-Triggered Architecture (TTA) [31] is among the most advanced and elaborate bodies of work developed to tolerate faults in safety-critical systems. TTA ensures message exchange in non-overlapping message slots and provides membership, fault tolerance, and actuation voting by leveraging apriori knowledge about the messages that replicas should send in the individual slots. TTA and its time partitioning are required in many standards, including ISO 26262 (automotive), IEC 61508 (industrial control), and DO-178C (avionics), and adopted by prominent industrial players, such as Audi, Volkswagen, and Honeywell [51]. The fault tolerance layer [5] is based on cold restart from the ground state (g-state) or history states (h-state). TTA is often used in conjunction with other methods for fault tolerance, such as redundancy, re-execution, and self-healing, to

[4] Not to confuse our K gain matrix of the LQR controller with a $m - K$ model from the control literature.

build robust and reliable real-time systems. Our CRC differs in that it ensures that both recovery states, critical for preserving data integrity and consistency, are always accessible to applications, even in case of a system failure or unexpected interruption.

One of the most popular methods used in fault tolerance is TMR [37], an instance of active replication. Its purpose is to improve the reliability of a system by using three (or, in general, n) functionally equivalent components to perform the same function, with the system's output being the majority vote of the three (n). TMR is commonly used in safety-critical systems, such as aerospace [68], nuclear power plants [66], and medical devices [17], where a single failure could result in severe harm or damage. The idea behind TMR is that the probability of all three components failing simultaneously is extremely low, so the system is highly reliable. While effective in improving system reliability, there are some downsides such as increased cost, power consumption, and complexity, which may make it less practical for some applications. Our CRC is an instance of active replication. It reduces the costs of the system by reducing the number of nodes from $n = 2f + 1$ to $n = f + 1$. One of the prerequisites for this reduction is the possibility of utilizing shared memory.

In addition to active replication, spare replicas can be kept hot, warm or cold, which translates to different response times for taking over after a fault is detected and the spare activated. Our approach applies active replication. Both active and passive replication will, over time, exhaust the majority of healthy replicas [59], in particular when cyberattacks persist. Our approach specifically address this concern by providing an extremely fast rejuvenation scheme for control-task replicas. Rejuvenating active replicas normally requires a cold restart of the faulty replica and additional replicas to compensate this downtime. Our approach avoids both.

Re-execution [72, 25] is a fault recovery technique used to improve the reliability of tasks by executing them multiple times and by selecting the correct output from multiple executions. It uses slack time on the processor to detect faults locally at the end of task execution and re-execute the task when a fault is detected. A faulty task can either be re-executed from the start or restored from the most recent checkpoint before the fault occurred [48]. Our CRC approach extends this technique by transforming control tasks to always maintain a known healthy saved state from which we restart tasks. This is achieved by not only voting on the result, but also on the state that must be maintained.

Other recent works in the intersection of fault tolerance and real-time systems include[32, 46, 10, 13, 1, 21].

Shared State. Replicated systems are typically constructed to avoid shared state due to the vulnerabilities entailed with this state failing. However, since recent microcontroller product families for safety-critical systems [61] offer ECC and RAID-protected shared memory [24, 64], we will leverage such memory to allow control replicas to maintain state across epochs. In particular, we will turn this memory into consensual memory, as exemplified by Gouveia et al. [19]. Read-only shared memory is commonly used in hypervisor-based systems to isolate VMs [43] (e.g., when deduplicating pages in their memory image). Our solution works in a hypervisor or RTOS setup, but equally well also on a bare-metal configuration where replicas receive read-only access to their shared memory. Read-only access suffices because, as we shall see, updates are performed consensually through a voter.

ECC embeds the possibility to tolerate faults without exposing them to the application and its state. It encodes values (e.g., into a hamming code) to tolerate a certain number of bit flips by correcting them when reconstructing the original value. The same coding can further be used to detect additional bit flips. As bit flips accumulate over time when

unhandled, ECC should be frequently be overwritten with a technique called scrubbing to restore its tolerance capabilities. Scrubbing overwrites the memory with the same value to restore non-stuck bits to their correct encoding of the value, which allows tolerating the original number of bit flips minus those that got stuck. We shall use ECC memory to protect state in consensual memory.

Checkpoint Recovery. Macroscopically, our approach could be characterized as a checkpoint recovery scheme albeit with an ultra-high checkpoint frequency and unconditional recovery at the end of each epoch. However, when we compare to other checkpointing approaches, such as [47, 71, 4], in more detail, there are substantial differences, which we characterize in the following:

- Checkpoints typically capture the entire state of a task in a consistent cut across all replicas (updating the previous checkpoint with what has been modified since then), whereas in our approach and using consensual memory, replicas propose only very selectively what portion of that state they will need for future epochs. The remaining (writable) state is simply discarded.
- Computation of and agreement on a new checkpoint are typically separate operations. Our approach combines both by only changing the shared state after $f+1$ healthy replicas agree on the update.
- Checkpoints are typically computed asynchronously to the execution of checkpointed tasks (e.g., by marking modified pages as copy on write to create a consistent cut). This is not necessary, since replicas end their activity in an epoch by proposing both what should be updated in the shared state and how to actuate the plant. This further ensures that the agreed-upon state corresponds to the latest plant actuation, since agreement is reached on both simultaneously and the voter follows suite in applying the updates.
- Recovery from a checkpoint is typically performed only after replicas have failed. This is not sufficient as compromised replicas might remain stealthy and go undetected. For this reason, we recover replicas after every epoch, by resetting them to the beginning of their control loop.
- Last but not least, checkpointing diversified replicas requires determining whether the individual checkpoints denote the same progress, whereas agreeing just on the values to be carried across control epochs, avoids such complications, because (i) the agreed upon state needs not to be diversified (it can only be written consensually), and (ii) healthy replicas agree on the same updates, despite computing them in a sufficiently different manner.

CPS Attacks. Various studies have investigated attacks on CPSs, including sensor [60], GPS spoofing [63, 26], and AI-related attacks [20]. In this work, we focus exclusively on tolerating cyberattacks that may be successful in compromising up to f control replicas while assuming adversaries cannot exceed that bound faster than T_a. Rejuvenating all replicas faster than T_a allows us to tolerate such adversarial attacks over extended periods of time [59].

4 System and Fault Model

System Model. This work concerns the fault tolerant control of a plant by means of replicating its control task across n nodes (see Figure 2). We assume nodes fail independently, but are sufficiently closely coupled to access a voter through the IO channels it offers and to access shared ECC and possibly RAID protected memory. These can be cores of a multi- or

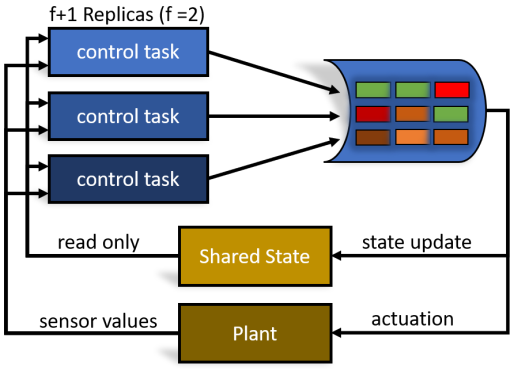


Figure 2 Replicated control architecture. Control task replicas sense the plant and have read-only access to shared state. They propose an actuation signal and state update, which the voter applies after reaching consensus.

many-core system (e.g., controlling a drone), multi-chip modules or more loosely coupled, but close compute nodes. If cached, the minimum requirement for the shared ECC memory is to invalidate cachelines upon writes, which as we shall see are updated exclusively by the voter.

A minimal control task senses the state of the plant, executes a control algorithm and proposes a signal for actuation. However, control tasks may also be more complex (e.g., structured as a directed acyclic graph of runnables) and involve filters, sensor aggregation and models of the plan to compute hidden state. Our goal is to support plants that are unaware of their controller's replicated nature. As such, we introduce a voter, which is trusted to consolidate control-task proposals into a singleton actuation signal. We shall also use the voter to update shared memory after reaching a consensus on how this state should be updated.

For simplicity, we assume a single control task having a single period is responsible for actuating the plant. Replicas of that task are invoked periodically every T time units and receive a consistent view of the plant state as far as this is observable through the plant's sensors. Supporting multi-periodic tasks (see e.g., Pagetti et al. [45]) is trivial as long as actuations are independent one from another. By replicating the multi-periodic control tasks individually and by introducing additional voters for each such group of replicas, one can achieve the desired actuation rates in the absence of errors. However, under faults, plants will have to tolerate missing some actuations for a number of epochs while others keep arriving. A discussion of multi-periodic control tasks with dependent actuations, where related actuations must not be passed to the plant if any of them cannot be performed at a given time, is out of the scope of this paper. We return in Section 5.2 to demonstrate how periodic activation and consistent sensing can be achieved using a trusted real-time operating system (RTOS) but also on bare metal. We call the T-distant invocations *control epochs*. Being invoked synchronously every T with a consistent view implies we operate under the assumptions of a synchronous system model.

Our approach is parametric in the number of faults f it can tolerate in an epoch (see our Fault model below for details) and in the number of epochs k by which we guarantee it to reach consensus. The parameters f and k determine how many replicas are required. Maggio et al. [39] found that many plants tolerate missing deadlines. The parameter k is bounded from above by this number m of deadlines that can be missed. Some plants, such as electric

steering can miss up to $m = 17$ deadlines. Our goal is to leverage this possibility to miss deadlines to optimize the system by reducing the number of replicas n required. Immediate masking (i.e., $k = 1$) within a single epoch (e.g., TMR) requires $n \geq 2f + 1$ replicas. Our goal is to reduce this number to as low as just $n = f + 1$ replicas, which can only detect faulty invocations that manifest in deviating proposals passed to the voter. With $n = f + 1$ replicas, at least one replica is guaranteed to make a healthy proposal. More generally, we are guaranteed to receive $n - f$ such healthy proposals during each epoch. To reach consensus, we have to collect $f + 1$ matching proposals including from at least one healthy replica. Our fault model rules out reaching such a match without healthy replica. But with only $n - f$ healthy proposals in each epoch, we are sure to collect the $f + 1$ matching proposals only after $\left\lceil \frac{f+1}{n-f} \right\rceil$ epochs, which bounds k from below. TMR typically operates under the assumption of $f = 1$, but there are systems deployed (e.g., for energy-grid safety), which have to tolerate $f = 5$ or even more faults simultaneously.

Fault Model. As mentioned above, we aim to protect against both accidental faults and intentionally induced, malicious faults (e.g., from cyberattacks). We shall therefore discuss several classes of faults and how we represent them in our abstract fault model, which is a slightly extended variant of the standard fault model for persistent and repeatedly partially-successful cyberattacks originally introduced by Sousa et al. [59].

Our fault model and hence the system we propose is parametric in the number of simultaneously occuring faults f that it can tolerate as well as in a few parameters ($T^f_{fault-type}$) which depend on the type of fault and which characterize when faults of that type may reoccur. The combination of number of faults and time of re-occurrence is quite standard, even in real-time systems. For example, the fault-tolerant time-triggered protocol by Kopetz and Grundsteidl [30] already assumed such a model. TTP can tolerate one fault among four synchronization replicas, provided the fault will not re-appear in the immediately following synchronization interval.

These parameters f, k and $T^f_{fault-type}$ are related to the configuration of the system in terms of the number of control-task replicas n that need to be deployed to tolerate this number of faults, the time $T_{rejuvenate}$ to rejuvenate replicas and in the time T_{agree} until agreement must be reached. The latter is further constrained by the number of subsequent deadline misses m that the plant can tolerate.

Deploying a system in an environment where these constraints cannot be guaranteed (e.g., because more than f faults of a class occur faster than $T^f_{fault-type}$) constitutes a failure in using the system. Fault tolerance and in consequence safety are no longer guaranteed once the thresholds are exceeded.

In terms of accidental faults, we consider transient and correctable faults that manifest in all parts of the system state, including in memory and in the internal and architecturally visible registers of the CPU. Such faults include bit flips due to radiation, charge deposited in flash memory cells, etc. We assume the RTOS frequently corrects such faults (e.g., by overwriting registers with the correct value or by scrubbing ECC memory). In particular we shall establish consensually-updated memory in ECC and possibly even RAID-protected memory. This memory will be shared between replicas and must return the latest written values, even in the presence of faults. If, on the other hand, bit flips occur only in a replica's memory and in not more than f over a period $T^f_{accidental}$, leaving our consensual memory unaffected, our system can handle these faults by collecting proposals from other replicas and by rejuvenating the faulty replica.

Accidental faults typically follow well understood characteristics from which a mean-time-to-failure (MTTF) can be derived and hence a high probability that faults will not reoccur within a certain time period, which for accidental faults we call $T^f_{accidental}$.

Faults of this nature often arise from external factors like radiation or fluctuations in temperature and are generally considered to be random events. To model these faults, stochastic processes, such as Poisson processes, are commonly employed. These models can help estimate the likelihood of faults happening within a designated time frame [38].

In the case of alpha particles hitting memory, this phenomenon is known as soft errors or single-event upsets (SEUs). Soft errors are transient faults that do not cause permanent damage to the system. They occur when high-energy particles, such as alpha particles or cosmic rays, strike the sensitive regions of semiconductor devices, causing bit flips in memory cells [6].

Malicious faults result from an attacker redirecting the control flow and altering the task's state to serve its purposes. Notice that techniques, such as return-oriented programming [9, 50], allow deviating from the task's intended control flow without modifying its code. This can happen, for example, by exploiting vulnerabilities such as buffer overflows to push return addresses that redirect the control flow to snippets of the task's code that, when combined, implement the adversary's desire. We assume a strong adversary capable of identifying, reaching, and exploiting such a vulnerability in control replicas.

Obviously, with identical replicas, no bound on the simultaneously affected replicas can be guaranteed. Instead, replicas need to be sufficiently diverse such that an attack applied to one replica cannot just be applied to another replica. Over time, adversaries may find vulnerabilities in more than f replicas by analyzing their current state (e.g., how its address space is randomized) and adjusting their exploit to the replicas state. This leads to two durations, which characterize the adversary. The time T^f_{deploy} required to deploy an attack and compromise a replica in the desired way, and the time T^f_{exceed} by which the adversary has analyzed more than f replicas. In this work, we allow T^f_{deploy} to become small (see below). However, we shall assume, as recommended by Sousa [59], that the RTOS diversifies all n replicas as part of the rejuvenation process faster than T^f_{exceed} (e.g., by re-randomizing their address space layout [8, 15] or by applying other diversification techniques [16, 35, 52]). Notice also that fault statistics do not apply to malicious faults.

We shall not further discuss diversification in this paper, as this needs to be applied at a different time scale[5], but they can easily be merged with the rejuvenation process we will introduce in Section 5.1 by not returning to the original binary's control loop and instead first activating a transition control loop after adjusting the address space of the task and then to the diversified version's control loop. See Section 5.4 for further details.

Rejuvenating replicas before each epoch to address faults, our approach can tolerate accidental and malicious faults, provided (i) the overall number of accidental and malicious faults will not exceed f and provided (ii) no more than f faults happen within any sliding window of length kT. The second condition holds if $T^f_{accidental} > kT$ and if $T^f_{deploy} > kT$. Of course, mean-time-to-failure is a value derived from fault statistics, which means that with a certain probability accidental faults can occur more frequently.

We shall not make such assumptions about accidental faults, but support more frequent occurrence of accidental faults, by leveraging another characteristics of such faults, namely that it is highly unlikely that two faults in two replicas will result in identical proposals. We

[5] T^f_{exceed} well exceeds the m epochs by which agreement needs to be reached.

will further reduce the likelihood of this happening by requiring replicas to solve a challenge for their proposal to be considered. Notice that this also addresses persistent faults, since a replica experiencing such a fault is unlikely to solve the challenge. Of course, persistent faults must be properly attributed in the overall number of faults and must be addressed by replacing the affected replica, which is out of the scope of this work. We shall return to this in Section 5.4.

It is also highly unlikely that adversaries will be able to predict accidental-fault intrinsics of a replica that still is able to solve the challenge, such that it can predict what that replica will propose. For application scenarios which can tolerate a low residual likelihood that the system becomes unsafe in case such a combination of events happen, we can therefore also deploy our solution in environments where up to f faults happen in each of the k epochs and where from the fk total faults, at most f are maliciously induced.

While replicas may fail as described above, we shall assume that the voter, the RTOS and the system clock will not fail in a similar manner. We assume they remain correct even in the presence of accidental faults and cyberattacks. For the voter, this assumption is justified by its simplicity, which allows implementing it entirely as custom logic in silicon or on an FPGA. Implementing the RTOS itself in a fault tolerant manner [55, 19] allows lifting the second assumption. Such a fault tolerant RTOS may then consensually update the system clock to remain in synchrony with other nodes in the system. Our solution is not resilient to physical attacks.

5 Consensual Resilient Control

In this section, we present our approach to consensual resilient control to tolerate up to f faults with just a detection quorum of $n \geq f + 1$ replicas. For now, let $n = k = f + 1$ and $f = 1$. That is, n replicas are periodically invoked with a consistent view of the plant state and are expected to produce an actuation signal, which they pass to the voter, which actuates the plant only after $f + 1$ replicas agree to the actuation value. In addition, to allow extremely fast recovery from faults and to make it possible to rejuvenate replicas in between any two subsequent invocations, they also vote on the state they would like to preserve across epochs.

Figure 3 shows for $f = 1$ and $n = k = 2$ how such a majority for the state update and actuation signal can be formed. In the first epoch, no faults happen, and the two replicas propose the same state update and actuation signal, which the voter applies since the $f + 1$ agreement has been reached. Even though replicas were correct, they are proactively rejuvenated to also return compromised but stealthy replicas to a known good state. In epoch 2, replica R_1 becomes faulty (either due to an attack or accidentally) and proposes an actuation value a_2' instead. Without further knowledge about the plant, the voter cannot discern which of the two proposed actuation signals is correct and will therefore not actuate (while possibly holding the previous actuation value a_1 if the plant requires that). It will also not update the state, even though the replicas agree on this part of the proposal. This is to avoid inconsistencies between the plant and the state maintained by the replicas. As for now, the plant is not actuated, and we experience a deadline miss, which, since we so far missed less than k deadlines, we assume the plant tolerates. After rejuvenating the replicas, the replicas start. However, this time, no agreement could be reached in the previous epoch, with the sensor information captured at the beginning of epoch 2. This time replica R_2 fails in epoch 3. If the previous fault of R_1 was due to a cyberattack, R_2 could fail only due to accidental causes because we assume adversaries cannot compromise more than f

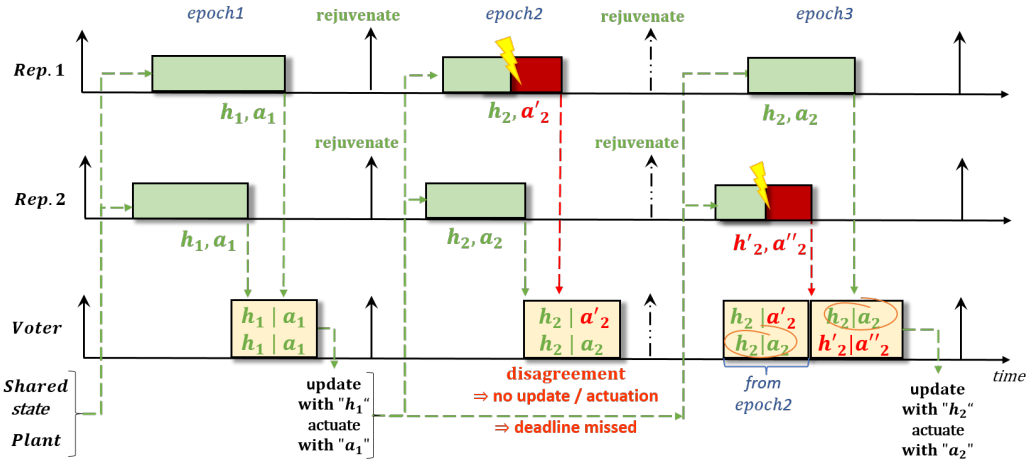


Figure 3 Example illustrating how $f + 1$ agreement can be achieved despite replicas failing. Shown is a scenario with $f = 1$ and $n = k = f + 1 = 2$ over three epochs. In the first, correct replicas agree. In the second epoch, no agreement can be reached due to replica R_1 failing. In epoch 3, the voter is able to collect $f + 1$ matching proposals after R_1 rejuvenates, even if this time R_2 fails.

replicas faster than the duration of k epochs. Also, by our fault model, the proposal of such an accidentally failing replica will not match the proposal R_1 made during epoch 2. If R_1 fails accidentally, adversaries are unlikely to predict how the failure will manifest. In both cases, the proposal from R_1 in epoch 2 and R_2 in epoch 3 will not already form a majority. However, after $k = 2$ epochs, the voter collected two votes from correct replicas (from R_2 in epoch 2 and R_1 in epoch 3). Finally, the voter is able to actuate again (with a_2) and update the state (with h_2).

Moreover, operating the system with more than $n = f + 1$ replicas is possible. In this case, $n - f$ replicas are correct by our fault model, and the voter can collect $n - f$ correct proposals in each epoch. Therefore, the number of epochs k needed before $f + 1$ agreement can be reached is $k = \left\lceil \frac{f+1}{n-f} \right\rceil$. As long as a plant can tolerate at least k deadline misses, n and k will be a correct configuration to tolerate up to f faults for that plant. An important prerequisite for this approach to work is that replicas can be recovered fast enough from faults and rejuvenated between any two invocations.

In the following, we shall therefore discuss how to systematically turn stateful control tasks into statelessly-recoverable instants (Section 5.1), how to invoke replicas with the same plant state (Section 5.2), and how to design a voter that is capable of supporting this construction and that is sufficiently simple to be implemented at the hardware level as a trusted-trustworthy component (Section 5.3). Then in Section 5.4, we bring everything together and discuss in Section 5.5 why it is safe to deploy our solution in an environment that meets the conditions laid out in the fault model in Section 4.

5.1 Converting stateful replicas into statelessly-recoverable instants

Control tasks, like other applications, modify their internal state. For example, both single and multi-threaded control tasks typically implement function invocation and local variables using a stack, they use global variables and, at least during startup, they may allocate objects on the heap. The easiest way of converting this state into easily recoverable information is to make all state read only and to store it in ECC-protected memory. This way, accidental

faults cannot manifest in the state and the error correcting code (ECC) captures accidental faults that occur in the memory block. Moreover, by making state read only, adversaries have no chance of modifying it without bypassing the processor's protection mechanism[6].

Indeed, it will not be possible to make the entire state of a control task read-only. At least the stack must remain writeable to support function calls and local variables. Fortunately, resetting the stack pointer to the location before a function call discards all values a previous function call has pushed. Therefore, to trivially recover control tasks, we turn the control loop of these tasks into a call to a function, which, as we shall see, will never need to return. Instead, it checks the voter to see whether the previous epoch was successful, which determines whether the current plant state should be considered or whether the replica needs to execute the control problem of a previous epoch, by reaching to that epoch's captured state and sensor values, to reach consensus about this epoch's control problem. It then proposes the actuation value and whatever part of this dynamic state should be preserved for the next epoch. Then, because we rejuvenate control tasks irrespective of their fault status, the only remaining part is to return to the control loop function while resetting the stack pointer. In other words, we turn the control loop function into a continuation and invoke it after every rejuvenation of the control task.

As we have seen, the state that control algorithms need to carry across epochs may range from a few values (such as the error and accumulator for PID) to several kilobytes of data (as in the electric microgrid controller from Huo et al. [27]). Our strategy for protecting this state is to store it in consensually-updated memory [19]. Consensually-updated memory is a memory shared among several replicas. To read, replicas can directly access the memory as it can be mapped read-only into the replica's address space. However, writing requires agreeing on which part of the memory should be updated and how. We leverage the voter to perform also these updates. In particular, we propose simultaneously all updates and the actuation signal to avoid inconsistencies due to partial updates. Also, since we collect proposals over k epochs, we cannot go back in time to receive additional parts of a proposal from a replica.

Control tasks not specifically built for our system will include code to read and write parts of this state. The transformation required to turn these control tasks into consensual-resilient-control (CRC) aware tasks is as follows. We analyze the program and allocate space on the stack in the context of the control-loop continuation. Upon the first write to a variable in consensual memory, we create a copy in that space and modify this copy instead of the original location[7]. Subsequent reads and writes then refer to this copy instead of the original location, and finally, the value of this copy is proposed as part of the update that the voter should apply. Transformations like the above are readily available in modern compilers (e.g., when constructing single-assignment form).

Figure 4 illustrates the above on an example address space layout of control task applications. Replicas receive a read-only copy of the plant state (see next section) and share as read-only mapping the code, read-only data, and shared memory. Stack and the MMIO interface to invoke the voter remain mapped in a writable manner. As part of the first write, a copy of the consensually updated state is created in the space allocated in the control-loop continuation's context on the stack, and all subsequent reads and writes are directed to this copy. The last operation of the control-loop continuation (fn_ctl) is to propose the copy as part of the state update and with the actuation signal, which the voter applies once $f + 1$

[6] We hope future safety-critical systems will be constructed from hardware components that are resistant to protection-bypassing attacks, such as Rowhammer [29].

[7] Being at the top of the stack, functions called from the continuation can reach this space.

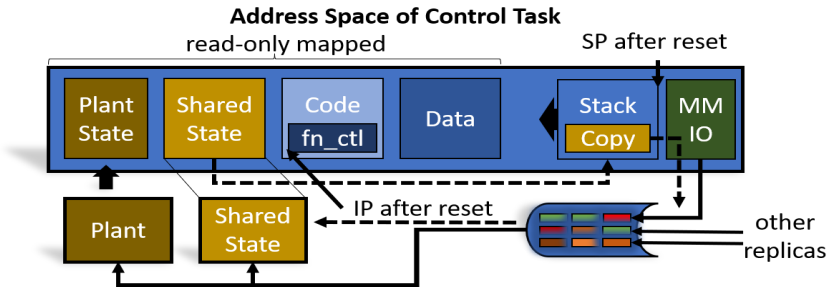


Figure 4 Address space layout of a control task replica. Shared state, code, and data are mapped read-only into the address space. Dashed lines indicate the data flow for variables in the consensually updated shared memory. Upon the first write, a copy is created on the stack and finally proposed to update the state after reaching a consensus. After reset, the instruction pointer (IP) is reset to the control loop function (fn_ctl) and the stack pointer (SP) to the beginning of the stack.

agreement with other replicas is reached. Irrespective whether or not fn_ctl terminates, the control task will be resumed in the next epoch with that function, after resetting its stack to remove any modifications an attack could have performed. This is important since compromised replicas may fail in an arbitrary manner, including by not proposing or by not terminating.

5.2 Sensing and control-task invocation

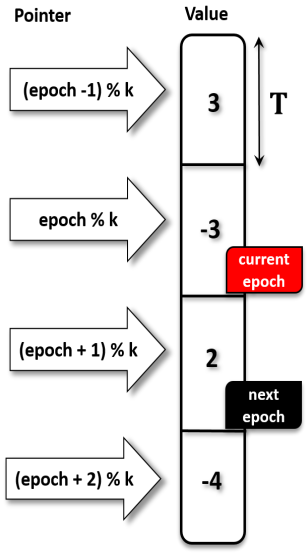
This section explains how we ensure replicas are invoked with the same view of the plant, both in a hosted environment and on bare metal. We start by looking at the implications of not invoking replicas with the same view of the plant. In this case, each replica would need to sense the plant state individually and would produce slightly different actuation signals and values to carry to the next epoch, even if we consider only correct replicas. Consequently, the voter would now need to identify when values are sufficiently close, which adds extra complexity to support this form of approximate agreement.

Instead of adding this complexity to the voter, it can also be added to the replicas by either reaching agreement on the sensed values or by agreeing on the actuation value and state update before presenting this to the voter. In either case, first reaching agreement requires collecting the opinions of $f + 1$ healthy replicas, which can only be guaranteed to happen after k epochs. Therefore, any additional agreement would require the plant to tolerate an extra k epochs of deadline misses, which would severely limit the applicability of our approach. In the following, we therefore present solutions that do not require additional agreement rounds other than for actuating the plant and updating consensual memory.

5.2.1 Hosted environments

Assuming a trusted-trustworthy operating system (OS), OS-level drivers could read sensors on the replicas' behalf and provide them with the values they read. Since replicas may need to revert to the past k elements, a $k + 1$ element ring buffer suggests itself as data structure, which the RTOS can map to the replicas' address spaces in a read-only manner.

Figure 5 shows the ring-buffer data structure used to grant the control task access to past sensor values and the pseudocode for a very simple controller leveraging this data structure. Retrieving from the voter the last epoch where votes were successful, replicas either contribute to the current control problem at hand (if this was the previous epoch)




```
1  // control loop
2  fn_ctl() {
3   epoch = voter.last_successful_epoch + 1;
4
5   // read sensor values
6   input = ring_buffer.get_sensor((epoch + 1) % k);
7
8   // compute controller response
9   <output,new_state> = control(input, state);
10
11  // make proposal
12  propose(voter.get_nonce(),epoch,output,new_state);
13
14  // wait for next epoch
15  // (resumes with continuation fn_ctl)
16  sleep();
17 }
```

(a) Ringbuffer's W/R mechanism. **(b)** Simple controller pseudocode.

Figure 5 Controller function fn_ctl and the ring buffer data structure used to refer back to previous plant states in case the previous epoch was not successful.

or they contribute to forming a majority for a past control problem that has failed so far to reach an agreement. As the code shows, with the ringbuffer in place, both cases can be treated in the same way, by obtaining the sensor value from the buffer (Line 6), computing the control output and state to carry over (Line 9) and by proposing both (Line 12) before yielding or sleeping until the next invocation. Replicas will be woken up as part of the rejuvenation process and resume at the beginning of the control loop captured in fn_ctl (at Line 3).

```
1 on rotary_interrupt:
2   epoch = (now() - start_time) / T;
3   angle[(epoch + 1) mod m] += direction()
4   return from interrupt
```

Figure 6 Interrupt handler for decoding rotary controller interrupts from the rotary encoder sensors of our pendulum into angular values (See also the pendulum in Section 2).

Let us illustrate the use of this data structure on an example with less cooperative sensors. Rotary encoders do not reveal the angle directly, but instead signal a change of their rotation angle by triggering interrupts. In our running example, we use the data structure shown in Figure 5 to sample the angles of the rotary controllers from the interrupts they generate at the rising and falling edge. To obtain the desired angle, rotary controllers require the operating system to accumulate angular changes, which they notify through interrupts. The interrupt handler code in Figure 6 shows this decoding of interrupts to angular values, where angle is the ringbuffer shown in Figure 5(a) and direction returns +4 or -4, depending on whether the encoder was turned right or left (i.e., depending on which of the two channels preceded the other (see Section 2)).

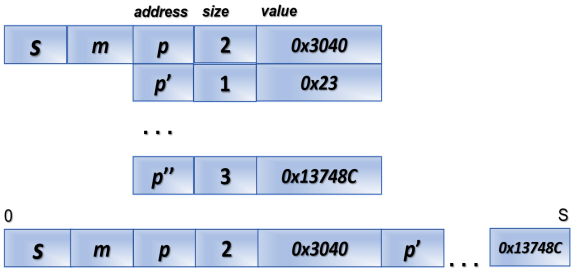


Figure 7 Layout of one of the voter buffers. The size s of the proposal and its inner structure in the form of m address, size, value triples are stored consecutively for easier comparisson.

5.2.2 Bare metal

Not all control tasks run in a hosted environment. In the following, we therefore sketch how replica invocation and sensing can be handled in a simple microcontroller for applications running on bare metal. We assume that in such an architecture, we still have the possibility to statically configure privileges (before the system starts critical operation) and to program a non-maskable timer to enter a read-only interrupt service routine that executes the code to activate the fn_ctl continuation at the beginning of the epoch and to reset the stack accordingly (Line 2 in Figure 5(b)).

Without additional hardware support, replicas, in a bare-metal configuration, have to sense themselves, which requires agreement and plants that tolerate at least $2k$ deadline misses before actuation can be guaranteed. To avoid the overhead entailed with this agreement, we suggest deploying capture hardware units [62] to periodically sample sensors in a reliable way and store the sampled results in memory that gets mapped to the replicas' address spaces in a read-only manner. Deploying $2f_{ccu}+1$ such units, where f_{ccu} is the tolerated fault threshold for these units allows replicas to immediately mask wrong sensor values and proceed with their control tasks. In particular for interrupt-driven sensors, such as the rotary controllers of our pendulum, the capture units should perform the accumulation task to avoid replicas having to accumulate captured interrupts themselves.

5.3 Voting on state updates and actuation

Consolidating the replicas' proposals into a single actuation output turns the voter into a necessarily trusted component, which, to be trustworthy, should remain as simple as possible. However, unlike voting in traditional TMR systems, not all proposals are available simultaneously, which requires the voter to buffer requests before $f+1$ matching votes can be extracted. In particular, we need nk buffers for $n \geq f+1$ replicas and for $k = \left\lceil \frac{f+1}{n-f} \right\rceil$ epochs.

For our pendulum and most systems we have investigated, actuation amounts to writing several memory-mapped registers, where the final write typically triggers the actuation. Likewise, consensual memory updates of state that should be carried to the next epoch also amount to writes to ECC-protected memory, respectively, to multiple locations in case of RAID. These write locations are typically not consecutive and, as we have seen before, proposals must be submitted in their entirety. Therefore an interface suggests itself where replicas specify m writes as address-size-value triples, stored as s consecutive bytes, as shown in Figure 7. This way, the $f+1$ matching proposals can be identified by matching s and the strings of size s in the respective buffers. Moreover, the voter can apply a successful vote (i.e.,

one reaching $f + 1$ agreement) by performing the m writes to the specified locations. Aside from maintaining the order of writes, we did not see any need for sophisticated consistency models other than register consistency, since voter-initiated writes will typically happen after replicas end their activity in an epoch and before the next epoch starts. In addition, healthy replicas may identify state updates in progress, by means of simple sequence locks in consensual memory.

To interface with the voter, we implement channels and map each channel to one replica. Moreover, we make the voter aware of epochs by exposing two read-only registers to each replica. The first contains the current epoch and is advanced every T. The second contains the epoch number when the last successful vote has happened (see Line 3 in Figure 5(b)). Making the voter aware of epochs avoids costly operations when resetting replicas, which otherwise would require changing the permissions of a replica to use a different channel. Upon receiving a message through the channel, the voter copies the proposal to the respective buffer for this replica and the epoch it is executing in, rotating through buffers as epochs advance.

As indicated in our fault model, we further complicate the case of faulty replicas reaching $f + 1$ agreement by introducing a challenge response mechanism to the voter interface. At the start of each epoch, after rejuvenating all replicas, the voter presents each replica a different random value – called *nonce*, which they are asked to reflect to the voter by xor-ing their proposal with this value. Then, rather than comparing the strings bytewise, the voter first xors the proposed string with the replica's current nonce, which returns the original string and then tries to find $f + 1$ matching proposals. This way, accidentally faulty replicas, in addition to adversaries needing to guess their proposal, must still be sufficiently correct to encode both the state update and the actuation signal using the provided nonce and to propose both to the voter before they are rejuvenated at the end of the epoch, which is highly unlikely. Notice that because the nonce is random and different every epoch, replicas which do not propose in an epoch are automatically considered as faulty replicas.

In preparation for changing the active replica set, we equip voters with more than n channels and with buffers for more than k epochs. This way, the active set can be supported with a subset of the resources available in the voter. A trusted replica manager can change this subset and the parameters n, k, and f over time, should that be necessary. The active subset is encoded in a bit vector with one bit per channel (considering those channels as active whose bit is set). Replicas with access to an inactive channel may already propose, but are ignored until their channel becomes active. This way, additional replicas can already be started and allowed to participate while the previous set of replicas is still in control of the plant. Then, once the new set of replicas are prepared, the trusted replica manager atomically transitions to this set by means of writing a single register. The change will become effective at the beginning of the next epoch. Figure 8 shows the channels, buffers, epoch registers as well as the reconfiguration registers just described.

From the description above, it should be clear that such a voter can be implemented as a service at the application level (waiting for signals from the replicas) as an operating-system service (invoked by system calls) or as a fixed-function custom logic mapped to an FPGA or implemented in silicon. In the latter two cases, replicas interface with the voter through memory-mapped IO registers that are mapped into the replicas' address spaces.

Notice also that while replicas must produce identical actuation signals and state updates to reach agreement (given the same consensually updated state and sensor values), they may (and in fact should to ensure fault independence) compute these proposals in a sufficiently different manner, such that an attack of one replicas does not automatically apply to others.

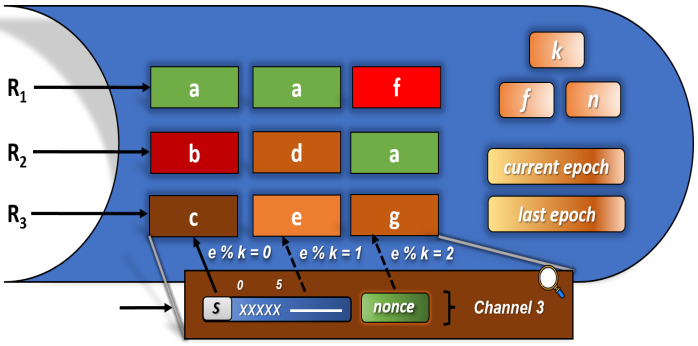


Figure 8 Voter internal structure. The voter provides one buffer per replica and epoch, which the replica can access through a channel. The proposal communicated through the channel is copies into the corresponding buffer of this replica for the current epoch. The voter reveals as well the current epoch and the last epoch where $f + 1$ agreement could be reached and allows k, f and n to be reconfigured by a trusted replica manager (if necessary).

5.4 Bringing it all together

With the above building blocks, we can now bring everything together. The system starts by initializing the plant, the trusted replica-management service (if necessary), and the replicas, which enter the control loop (fn_ctl) as a continuation. When the continuation starts, the RTOS or capture units have already sensed the observable part of the plant and captured that information in a ring-buffer. Therefore if the previous epoch was successful, the replica can proceed with the current sensor values and the current state in consensual memory to produce an actuation signal and update for the state that needs to be carried to the next epoch. Both are proposed to the voter to reach an agreement.

If the previous epoch was unsuccessful, the replica performs the same steps but with the sensor values for the epoch that precedes the last successful one. Notice that in this case, the consensual memory has not been updated and contains the control parameters (e.g., accelerator and previous value for PID control) required for that epoch. Once the voter receives $f + 1$ matching proposals, it marks the current epoch as the last successful and executes the agreed-upon sequence of writes, updating the consensual memory and actuating the plant.

Healthy replicas end their activity in an epoch by sleeping. However, regardless of whether a replica sleeps, the RTOS/non-maskable timer will signal a protected handler in the replicas, which resets the replica by returning to the start of the control-loop continuation (fn_ctl) and by resetting the replica's stack.

5.5 Safe Deployment

In our fault model in Section 4, we have defined constraints for the safe deployment of systems that implement our solution. If these constraints are met, control tasks will reach an agreement, and the agreement is on a correct proposal only. System safety then depends on correct control tasks proposing correct actuations in the given situation, which to ensure is out of the scope of this work.

The constraints highlighted in Section 4 are that (i) no more than f total faults occur and that (ii) the system addresses faults faster than kT with no further faults occurring before that time. In particular, we have discussed that malicious faults must be constrained through diversification such that not all replicas become faulty simultaneously, with the additional

constraint that the time to re-deploy an attack remains above kT. We have also discussed that persistent faults, which cannot be handled at that timescale need to be accounted for. That is, if there are $f_{persistent} < f$ faults, present, only up to $f - f_{persistent}$ faults of another kind may occur within the sliding windows of length at least kT.

With up to f faults over a time kT, at most f proposals may originate from faulty of otherwise compromised replicas. Therefore, when $f + 1$ matching proposals are collected, they are collected from at least one healthy replica. In particular, by the measures we discuss in Section 5.2 and Section 5.3, we ensure that all replicas operate from the same state and are invoked in a reliable manner after rejuvenation. This means (due to our assumption that fused sensor values are correct) that any healthy replica will propose a correct proposal and that agreement can only be reached on such a proposal.

It is always possible to reach agreement on such a proposal because over up to k epochs, $n - fk$ replicas are correct (possibly after rejuvenating them), which because $k = \left\lceil \frac{f+1}{n-f} \right\rceil$, is larger or equal to $f + 1$. Hence the system is life.

To see why our system is also correct in case up to f faults occur during each epoch, but with the additional constraints that (iii) among the fk faults over any sliding window of length kT only up to f are malicious faults, (iv) that malicious faults do not propose the same value as accidental faults and (v) no two accidental faults agree in their value, we have to see that agreement cannot already be reached among faulty replicas. Condition (iii) rules out agreement just by including malicious replicas and (iv) that malicious replicas collude with an accidentally faulty replica, even if up to f malicious replicas agree in their proposal. (v) avoids agreement among accidentally faulty replicas. Notice though that these are probabilistic arguments and that, as mentioned in Section 4, systems cannot safely be deployed if the residual likelihood of agreement among accidentally faulty replicas or if in the targeted environment, the residual likelihood of malicious replicas guessing the fault characteristics of an accidentally faulty replica, cannot be tolerated by the system. Our challenge to require replicas to send their proposals by xor-ing a voter provided nonce, further reduces these likelihoods, as accidentally faults replicas must remain able to do so, despite the fault manifesting.

The above condition also ensures liveness in this setting, since $n - f$ replicas remain correct in each epoch, which, when collecting their proposals over k epochs sums up to at least $f + 1$ proposals from correct replicas.

5.6 Distributed Control

Until now and for our evaluation, we have assumed systems that are sufficiently tightly coupled so that communication through shared, ECC- and RAID-protected memory remains possible. In such a setting, a voter can collect proposals, update consensual memory and actuate the plant. The same applies to closely coupled nodes for the control task, but a remote plant as long as communication between the voter and the plant is reliable.

To support distributed nodes for the control tasks and a remote plant, both the sensor signals and the actuation signals must be communicated reliably to all notes in the system (e.g., by using communication media that already have such a reliability built in [22] or by running a suitable reliable transmission protocol [33, 44]. In such a setting, shared memory will likely not be available and should therefore be replaced by consensually updating locally accessible, read-only mapped memories and by reconstructing the state of such a memory from its peers in case one of the nodes' memory fails. Investigating the tradeoffs of such a solution is out of the scope of this paper.

6 Evaluation

To demonstrate that our approach to consensual resilient control is in fact robust, even in the presence of errors, we have implemented the voter and two control algorithms (LQR and PID) by leveraging Linux's user-level driver infrastructure to sense and actuate our inverted pendulum. In particular, we were interested in whether the ability to tolerate accidental faults in the time domain allows controlling such an application from the less predictable environment that standard Linux offers. For small epochs ($T < 10ms$), we had to use Linux' "silent core" feature to limit OS activity on the cores to which we pinned our control tasks.

All measurements were conducted on a 4-core Raspberry 3 Model B+ with 1GB RAM, running at 1.4 GHz, using the pendulum shown in Section 2. In addition, to evaluate the scalability of our approach, we used a 4x6 core Intel Xeon Gold 6334 CPU, running at 3.4 GHz, and a software emulator of the pendulum (implementing its equation of motion and random turbulence).

We have implemented the voter in software as a user-level task and have pinned replicas and the voter each to a separate core. Replicas communicate with the voter through a dedicated shared memory region, as depicted in Figure 4, which implements the voter's channel interface. We inject faults into randomly selected replicas and consider only faults that manifest in proposing values that are different from those of healthy replicas. For accidental faults, a random value is proposed. For maliciously-induced faults, we as well select a random value but will use the same value for all compromised replicas. In addition, for demonstration purposes, pressing a button on the Raspberry PI will as well cause a random replica to fail.

6.1 Overhead

Since our approach is to re-execute replicas after rejuvenation for up to k epochs, the runtime overhead in terms of time to agreement under faults is dominated by the number of epochs required to collect $f + 1$ faults. In no faults occur, replicas actuate within a single epoch and the worst-case time to agreement is the WCET of the control task plus the overhead to propose and update the state that needs to be preserved across epochs.

We measured this overhead for PID and LQR on the Raspberry PI and with our inverted pendulum ($f = 1, n = 2$) to be 0.39 μs for the time that the replica needs to preserve the state for the next epoch, by proposing the error and accumulator (PID) and the measured angles to calculate angular velocities (LQR). The voter required 0.034 μs to update consensual memory and actuate the plant.

In addition, we performed a series of microbenchmarks on our x86-based simulator of the pendulum to understand these overheads for different controllers that require preserving increasing amounts of state across epochs. Figure 9 shows these results for the same scenario ($f = 1$ and $n = 2$). Shown are the maximum observed (bars), average (green dot) and P95 (top) and P05 (bottom) percentiles of these execution-time overheads.

As can be seen, the overhead of turning control tasks into statelessly-recoverable instants, by pushing all state that needs to be maintained across epochs to consensual memory, is negligible for controllers with small state and well below 2ms for controllers that operate on a significant amount of dynamic state (such as the one from Huo et al. [27]). It should be noted that typical book-keeping tasks can also be performed on consensual memory, with little additional overhead for logging system states in consensual memory.

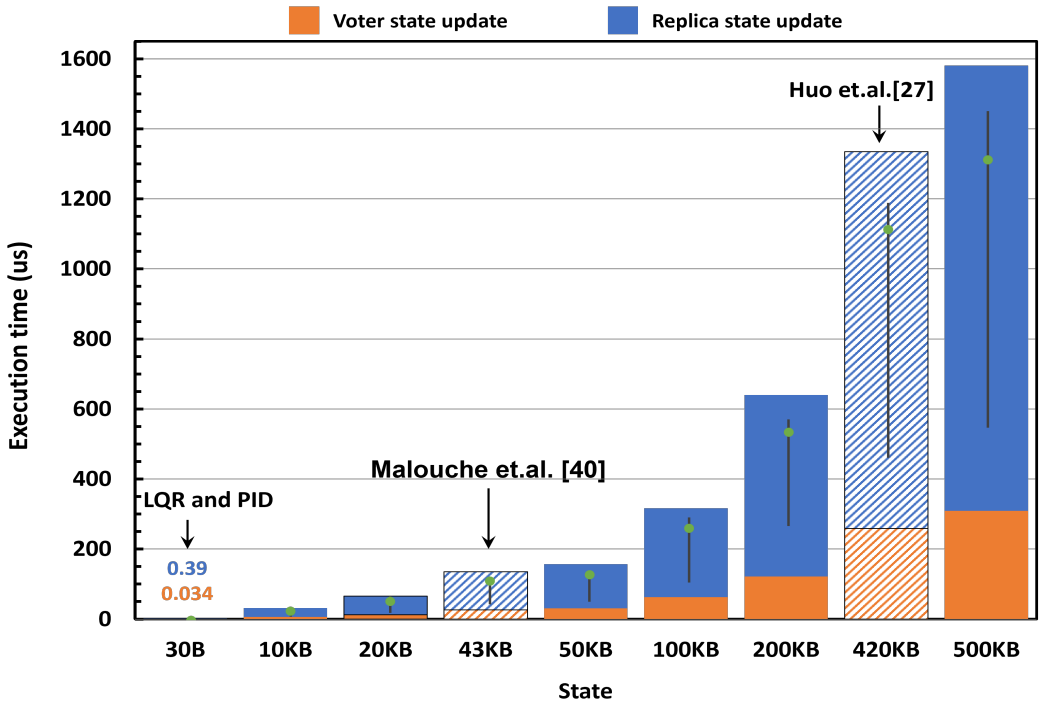


Figure 9 Overhead of consensual resilient control (in μs) broken down into the overhead on the replica side to propose the actuation value and update of the state that should be preserved for the next epoch and into the voter overhead of applying this update and the actuation signal. Shown is the scenario for $f = 1$, $n = 2$.

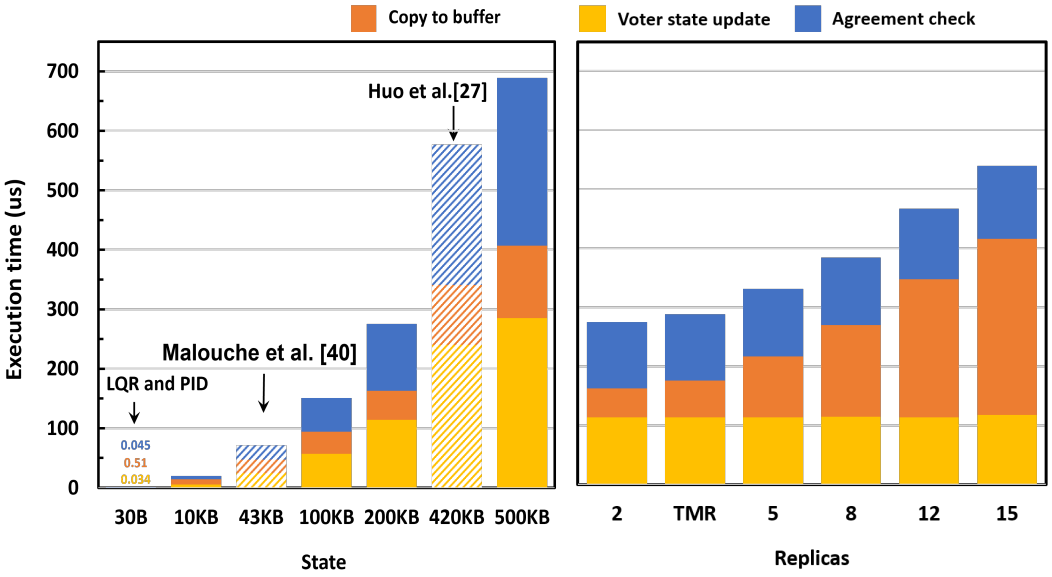
6.2 Breakdown of Voting Overheads

Figure 10 further breaks down the overhead for voting into the different operations that a software-level voter needs to perform. Hardware implementations can avoid buffering costs by directing inputs directly to the current epoch's buffer and they may parallelize agreement checks. Figure 10(a) investigates for $n = 2$ replicas how the voting overhead scales with the size of the state that needs to be preserved across epochs. Figure 10(b) shows these results for increasing n and therefore also for increasing f and a fixed state size of 200KB.

As can be seen, updating consensual memory, copying to the buffer and checking for agreement is linear in the size of the proposal, given that the number of replicas is fixed to $n = 2$ for these measurements. Similarly, updating consensual memory and copying to the buffer are constant for a fixed-size message, irrespective of the number of replicas and the agreement check linear in the number of replicas (and hence faults tolerated) in case no faults occur (as shown in 10(b)) and quadratic $(n \cdot k)$ when $f + 1$ agreement must be collected over up to k epochs. This is because whenever a replica proposes, the voter checks this proposal to all buffers that already contain a proposal for the voted-upon epoch.

6.3 Actuation Signals

Figure 11 shows the sensor (Channel 2–4) and actuation signal (Channel 1) of the inverted pendulum, controlled over three epochs with a consensual resilient PID controller. The scenario depicted in the figure roughly resembles the situation presented in Figure 3. During the first epoch, actuating at vertical line (4), no faults happen and the DC motor gets



(a) Breakdown of the overhead of voting for $n = 2$.

(b) Scalability of voting overhead for larger f and n. Shown are the results for a 200KB cross epoch state.

Figure 10 Voting overhead for the constant invocation of $T = 25$ms.

configured to a 66% duty cycle, as seen in the wider pulse width in Channel 1. As a response to this actuation, the rate of change of the angle drops, as can be seen from the longer distances between the rising and falling edges on Channel 2 and 3. Therefore, the control algorithm selects a lower duty cycle to reduce motor velocity and slow down the cart and thereby also the pendulum motion even further, with the idea of reaching the stable point where the pendulum is pointing straight to the top. Unfortunately, a replica fails during that epoch (since we injected a fault). In consequence, after the voter receives $f + 1$ proposals at the point in time denoted by vertical line (5), no agreement can be reached and the voter will hold the previous duty cycle of 66%. In the following epoch, we again inject a fault into one of the replicas, but this time, $f + 1$ agreement can be reached by combining the proposals of the current and the previous epoch. The voter applies the proposal and adjusts the duty cycle to 8.3%, as can be seen in the change of the pulse-width encoded signal. We also see slight variations between the actuation points. This is due to the control replicas executing with slightly different actual execution times within the $7ms$ epochs.

6.4 Rejuvenation costs

A central contribution of this work is the reduction of rejuvenation costs to just resuming the control loop continuation (fn_ctl) and resetting the stack, which both have overheads in the single to double-digit cycle range. In addition, we induce a maximum observed overhead of $0.39\mu s$ (with LQR and PID) for proposing and updating the state in consensual memory that must be preserved across epochs.

To compare and contrast these costs, we have also measured the average-case overhead when rejuvenating replicas traditionally by creating a new process ($329.06\mu s$), a new thread ($13.28\mu s$) and by mapping the voter interface to this replica ($100.82\mu s$). In addition, such a replica would experience cold start effects and need to catch up to the state of other replicas.

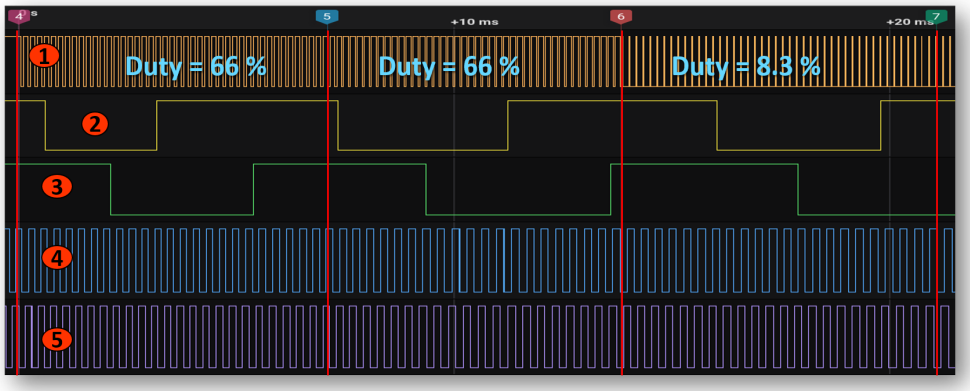


Figure 11 Sensor and actuation signals of the pendulum were evaluated using a logic analyzer. Shown are the points in time of actuation (vertical lines) for three epochs (marked on the top as 4, 5 and 6). The individual channels show DC motor actuation (1), encoded as a pulse-width modulated signal, the two channels of the rotary encoder which measures the angle of the pendulum (2) and (3), as well as the two channels measuring the position (4) and (5).

However, as can already be seen from the reported numbers, the costs of rejuvenating replicas traditionally are significant. It should also be noted that it is difficult to bind these costs from above, which is why typically, real-time systems only use these operations while they have to guarantee timeliness. Notice that rejuvenation will also be required in systems, such as TMR, that are capable of masking faults. This is because persistent attacks exhaust the healthy majority over time. Rejuvenation restores this majority.

7 Conclusion

This paper presents Consensual Resilient Control, a framework specifically designed to ensure the resilience of control tasks to accidental and malicious faults. We have shown how stateful control tasks can be systematically transformed into statelessly-recoverable instances by protecting in consensual memory any state that must be preserved across epochs (such as the PID controller's error and accumulator for derivative and integral control). This allows masking faults just with a detection quorum of $n \geq f + 1$ replicas, provided $f + 1$ agreement can be collected over $k = \left\lceil \frac{f+1}{n-f} \right\rceil$ epochs and provided the plant can tolerate up to k deadline misses.

We discuss several intricacies of applying consensual resilient control to a real-life application scenario by demonstrating its operation with our self-made inverted pendulum. In addition, we have evaluated our approach in the less predictable setting of running controllers as user-level Linux processes with user-level drivers for sensing and actuating the plant. We have also conducted several microbenchmarks to assess the behavior of consensual resilient control when increasing amounts of state have to be preserved across epochs (up to the 420KB required for the MPC electric microgrid controller by Huo et al. [27]) as well as for increasing f and n.

In the future, we plan to investigate scenarios requiring a change of the control algorithm, as well as simplex/complex controller interplays in a fault-tolerant setting.

References

1 Loveless A, Dreslinski R, Kasikci B, and Phan LT. Igor: Accelerating byzantine fault tolerance for real-time systems with eager execution. In *IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 360–373, May 2021.

2 Leonie Ahrendts, Sophie Quinton, Thomas Boroske, and Rolf Ernst. Verifying weakly-hard real-time properties of traffic streams in switched networks. In *ECRTS 2018-30th Euromicro Conference on Real-Time Systems*, pages 1–22, 2018.

3 Charles W Anderson. Learning to control an inverted pendulum using neural networks. *IEEE Control Systems Magazine*, 9(3):31–37, 1989.

4 Hakan Aydin and Qi Yang. Energy-aware partitioning for multiprocessor real-time systems. In *Proceedings International Parallel and Distributed Processing Symposium*, pages 9–pp. IEEE, 2003.

5 Günther Bauer and Hermann Kopetz. Transparent redundancy in the time-triggered architecture. In *Proceeding International Conference on Dependable Systems and Networks. DSN 2000*, pages 5–13. IEEE, 2000.

6 Robert C Baumann. Soft errors in advanced semiconductor devices-part i: the three radiation sources. *IEEE Transactions on device and materials reliability*, 1(1):17–22, 2001.

7 Guillem Bernat, Alan Burns, and Albert Liamosi. Weakly hard real-time systems. *IEEE transactions on Computers*, 50(4):308–321, 2001.

8 Hristo Bojinov, Dan Boneh, Rich Cannings, and Iliyan Malchev. Address space randomization for mobile devices. In *Proceedings of the Fourth ACM Conference on Wireless Network Security (WiSec'11)*, pages 127–138, 2011. doi:doi:10.1145/1998412.1998434.

9 Erik Buchanan, Ryan Roemer, and Stefan Savage. Return-oriented programming: Exploits without code injection. In *Black HAT USA*, August 2008. URL: https://hovav.net/ucsd/talks/blackhat08.html.

10 Gang Chen, Nan Guan, Kai Huang, and Wang Yi. Fault-tolerant real-time tasks scheduling with dynamic fault handling. *Journal of Systems Architecture*, 102:101688, 2020.

11 Thomas Chen and Saeed Abu-Nimeh. Lessons from stuxnet. *Computer*, 44(4):91–93, 2011.

12 Hongjun Choi, Sayali Kate, Yousra Aafer, Xiangyu Zhang, and Dongyan Xu. Software-based realtime recovery from sensor attacks on robotic vehicles. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, pages 349–364, 2020.

13 Roth E and Haeberlen A. Do not overpay for fault tolerance! In *IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 374–386, May 2021.

14 Haoyang Fan, Fan Zhu, Changchun Liu, Liangliang Zhang, Li Zhuang, Dong Li, Weicheng Zhu, Jiangtao Hu, Hongye Li, and Qi Kong. Baidu apollo em motion planner. *arXiv preprint arXiv:1807.08048*, 2018.

15 Joachim Fellmuth, Thomas Göthel, and Sabine Glesner. Instruction Caches in Static WCET Analysis of Artificially Diversified Software. In Sebastian Altmeyer, editor, *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*, volume 106 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:23, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ECRTS.2018.21.

16 S. Forrest, A. Somayaji, and D. H. Ackley. Building diverse computer systems. In *Hot Topics in Operating Systems*, pages 67–72, 1997. doi:doi:10.1109/HOTOS.1997.595185.

17 Markus Fras, H Kroha, O Reimann, B Weber, and R Richter. Use of triple modular redundancy (tmr) technology in fpgas for the reduction of faults due to radiation in the readout of the atlas monitored drift tube (mdt) chambers. *Journal of Instrumentation*, 5(11):C11009, 2010.

18 Goran Frehse, Arne Hamann, Sophie Quinton, and Matthias Woehrle. Formal analysis of timing effects on closed-loop properties of control software. In *2014 IEEE Real-Time Systems Symposium*, pages 53–62. IEEE, 2014.

19 Inês Pinto Gouveia, Marcus Völp, and Paulo Esteves-Verissimo. Behind the last line of defense: Surviving soc faults and intrusions. *Computers & Security*, 123:102920, 2022.

20 Blessing Guembe, Ambrose Azeta, Sanjay Misra, Victor Chukwudi Osamor, Luis Fernandez-Sanz, and Vera Pospelova. The emerging threat of ai-driven cyber attacks: A review. *Applied Artificial Intelligence*, 36(1):2037254, 2022.

21 Li H, Lu C, and Gill CD. Rt-zookeeper: Taming the recovery latency of a coordination service. In *ACM Transactions on Embedded Computing Systems (TECS)*, volume 20, pages 1–22, September 2021.

22 Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. Peerreview: Practical accountability for distributed systems. *ACM SIGOPS operating systems review*, 41(6):175–188, 2007.

23 Zain AH Hammadeh, Rolf Ernst, Sophie Quinton, Rafik Henia, and Laurent Rioux. Bounding deadline misses in weakly-hard real-time systems with task dependencies. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 584–589. IEEE, 2017.

24 Richard W Hamming. Error detecting and error correcting codes. *The Bell system technical journal*, 29(2):147–160, 1950.

25 Pengcheng Huang, Hoeseok Yang, and Lothar Thiele. On the scheduling of fault-tolerant mixed-criticality systems. In *Proceedings of the 51st annual design automation conference*, pages 1–6, 2014.

26 Todd E Humphreys, Brent M Ledvina, Mark L Psiaki, Brady W O'Hanlon, Paul M Kintner, et al. Assessing the spoofing threat: Development of a portable gps civilian spoofer. In *Proceedings of the 21st International Technical Meeting of the Satellite Division of The Institute of Navigation (ION GNSS 2008)*, pages 2314–2325, 2008.

27 Yuchong Huo, François Bouffard, and Géza Joós. Integrating learning and explicit model predictive control for unit commitment in microgrids. *Applied Energy*, 306:118026, 2022.

28 Greg Jaffe and Thomas Erdbrink. Iran says it downed u.s. stealth drone; pentagon acknowledges aircraft downing. The Washington Post, December 2011.

29 Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. *ACM SIGARCH Computer Architecture News*, 42(3):361–372, 2014.

30 H. Kopetz and G. Grunsteidl. Ttp - a time-triggered protocol for fault-tolerant real-time systems. In *FTCS-23 The Twenty-Third International Symposium on Fault-Tolerant Computing*, pages 524–533, 1993. doi:10.1109/FTCS.1993.627355.

31 Hermann Kopetz and Günther Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, 2003.

32 C Mani Krishna. Fault-tolerant scheduling in homogeneous real-time systems. *ACM Computing Surveys (CSUR)*, 46(4):1–34, 2014.

33 Leslie Lamport. The part-time parliament. In *Concurrency: the Works of Leslie Lamport*, pages 277–317. ACM, 2019.

34 Ralph Langner. Stuxnet: Dissecting a cyberwarfare weapon. *IEEE Security and Privacy*, 9(3):49–51, 2011.

35 P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. Sok: Automated software diversity. In *IEEE Symposium on Security and Privacy*, 2014. doi:doi:10.1109/SP.2014.25.

36 Robert M. Lee, Michael J. Assante, and Tim Conway. German steel mill cyber attack. Industrial Control Systems - avail at: https://ics.sans.org/media/ICS-CPPE-case-Study-2-German-Steelworks Facility.pdf, December 2014.

37 Robert E Lyons and Wouter Vanderkulk. The use of triple-modular redundancy to improve computer reliability. *IBM journal of research and development*, 6(2):200–209, 1962.

38 Michael R Lyu et al. *Handbook of software reliability engineering*, volume 222. IEEE computer society press Los Alamitos, 1996.

39 Martina Maggio, Arne Hamann, Eckart Mayer-John, and Dirk Ziegenbein. Control-system stability under consecutive deadline misses constraints. In *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

40 Ibtissem Malouche, A Kheriji Abbes, and Faouzi Bouani. Automatic model predictive control implementation in a high-performance microcontroller. In *2015 IEEE 12th International Multi-Conference on Systems, Signals & Devices (SSD15)*, pages 1–6. IEEE, 2015.

41 Aleksandar Matović. Case studies on modeling security implications on safety, 2019.

42 Sibin Mohan, Stanley Bak, Emiliano Betti, Heechul Yun, Lui Sha, and Marco Caccamo. S3a: Secure system simplex architecture for enhanced security and robustness of cyber-physical systems. In *Proceedings of the 2nd ACM international conference on High confidence networked systems*, pages 65–74, 2013.

43 Djob Mvondo, Alain Tchana, Renaud Lachaize, Daniel Hagimont, and Noël De Palma. Fine-grained fault tolerance for resilient pVM-based virtual machine monitors. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 197–208. IEEE, 2020.

44 Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, 2014.

45 Claire Pagetti, Julien Forget, Frédéric Boniol, Mikel Cordovilla, and David Lesens. Multi-task implementation of multi-periodic synchronous programs. *Discrete event dynamic systems*, 21:307–338, 2011.

46 Risat Mahmud Pathan. Fault-tolerant and real-time scheduling for mixed-criticality systems. *Real-Time Systems*, 50:509–547, 2014.

47 Krithi Ramamritham and John A. Stankovic. Scheduling algorithms and operating systems support for real-time systems. *Proceedings of the IEEE*, 82(1):55–67, 1994.

48 Reza Ramezani and Yasser Sedaghat. An overview of fault tolerance techniques for real-time operating systems. *ICCKE 2013*, pages 1–6, 2013.

49 Michael Riley and John Walcott. China-based hacking of 760 companies shows cyber cold war. *Bloomberg, Dec*, 14, 2011.

50 Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 15(1):1–34, 2012.

51 John Rushby. Bus architectures for safety-critical embedded systems. In *International Workshop on Embedded Software*, pages 306–323. Springer, 2001.

52 Moritz Schloegel, Tim Blazytko, Moritz Contag, Cornelius Aschermann, Julius Basler, Thorsten Holz, and Ali Abbasi. Loki: Hardening code obfuscation against automated attacks. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3055–3073, Boston, MA, August 2022. USENIX Association. URL: `https://www.usenix.org/conference/usenixsecurity22/presentation/schloegel`.

53 Sarah Scoles. The feds want these teams to hack a satellite – From home. The wired – `https://www.wired.com/story/the-feds-want-these-teams-to-hack-a-satellite-from-home/`, August 2020.

54 Danbing Seto and Lui Sha. A case study on analytical analysis of the inverted pendulum real-time control system. Technical report, Carnegie-Mellon University, 1999.

55 Yanyan Shen, Gernot Heiser, and Kevin Elphinstone. Fault tolerance through redundant execution on cots multicores: Exploring trade-offs. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 188–200, 2019. `doi:10.1109/DSN.2019.00031`.

56 D. Shepard, J. Bhatti, and T. Humphreys. Drone hack. *GPS World*, 23(8):30–33, 2012.

57 Douglas Simoes Silva, Rafal Graczyk, Jérémie Decouchant, Marcus Völp, and Paulo Esteves-Verissimo. Threat adaptive byzantine fault tolerant state-machine replication. In *40th International Symposium on Reliable Distributed Systems (SRDS)*, September 2021.

58 Jill Slay and Michael Miller. Lessons learned from the maroochy water breach. *Critical Infrastructure Protection*, pages 73–82, 2007.

59 Paulo Sousa, Nuno Ferreira Neves, and Paulo Veríssimo. Proactive resilience through architectural hybridization. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, pages 686–690, 2006.

60 Rong Su. Supervisor synthesis to thwart cyber attack with bounded sensor reading alterations. *Automatica*, 94:35–44, 2018.

61 Infineon Technologies. 32-bit aurix™ tricore™ microcontroller. URL: `https://www.infineon.com/cms/en/product/microcontroller/32-bit-tricore-microcontroller/`.

62 Infineon Technologies. ccu4 capture and compare unit 4. `https://www.infineon.com/dgdl/Infineon-IP_CCU4_XMC-TR-v01_00-EN.pdf?fileId=5546d4624ad04ef9014b0780bb082263&ack=t`.

63 Nils Ole Tippenhauer, Christina Pöpper, Kasper Bonne Rasmussen, and Srdjan Capkun. On the requirements for successful gps spoofing attacks. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 75–86, 2011.

64 Ulf Troppens, Rainer Erkens, and Wolfgang Müller. *Storage networks explained: basics and application of fibre channel SAN, NAS, iSCSI and InfiniBand*. John Wiley & Sons, 2005.

65 Nils Vreman, Anton Cervin, and Martina Maggio. Stability and Performance Analysis of Control Systems Subject to Bursts of Deadline Misses. In Björn B. Brandenburg, editor, *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)*, volume 196 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 15:1–15:23, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.ECRTS.2021.15`.

66 Xin Wang, Keith Holbert, and Lawrence T Clark. Using tmr to mitigate seus for digital instrumentation and control in nuclear power plants. In *7th International Topical Meeting on Nuclear Plant Instrumentation, Control, and Human-Machine Interface Technologies 2010, NPIC and HMIT 2010*, pages 925–934, 2010.

67 Victor Williams and Kiyotoshi Matsuoka. Learning to balance the inverted pendulum using neural networks. In *[Proceedings] 1991 IEEE International Joint Conference on Neural Networks*, pages 214–219. IEEE, 1991.

68 Aibin Yan, Zhelong Xu, Kang Yang, Jie Cui, Zhengfeng Huang, Patrick Girard, and Xiaoqing Wen. A novel low-cost tmr-without-voter based his-insensitive and mnu-tolerant latch design for aerospace applications. *IEEE Transactions on Aerospace and Electronic Systems*, 56(4):2666–2676, 2019.

69 Kim Zetter. Google hack attack was ultra sophisticated, new details show, January 2010. URL: `https://www.wired.com/2010/01/operation-aurora/`.

70 Kim Zetter. A cyberattack has caused confirmed physical damage for the second time ever. `https://www.wired.com/2015/01/german-steel-mill-hack-destruction`, 2015.

71 Ying Zhang and Krishnendu Chakrabarty. Fault recovery based on checkpointing for hard real-time embedded systems. In *Proceedings 18th IEEE Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 320–327. IEEE, 2003.

72 Junlong Zhou, Min Yin, Zhifang Li, Kun Cao, Jianming Yan, Tongquan Wei, Mingsong Chen, and Xin Fu. Fault-tolerant task scheduling for mixed-criticality real-time systems. *Journal of Circuits, Systems and Computers*, 26(01):1750016, 2017.

73 Xingliang Zou, Albert MK Cheng, and Yu Jiang. P-frp task scheduling: A survey. In *2016 1st CPSWeek Workshop on Declarative Cyber-Physical Systems (DCPS)*, pages 1–8. IEEE, 2016.