Fault Tolerance and the Five-Second Rule

Ang Chen Hanjun Xiao Andreas Haeberlen Linh Thi Xuan Phan
University of Pennsylvania

Abstract

We propose a new approach to fault tolerance that we call *bounded-time recovery (BTR)*. BTR is intended for systems that need strong timeliness guarantees during normal operation but can tolerate short outages in an emergency, e.g., when they are under attack. We argue that BTR could be a good fit for many cyber-physical systems. We also sketch a technical approach to providing BTR, and we discuss some challenges that still remain.

1 Introduction

In everyday life, we are often willing to overlook small mistakes, as long as they are fixed quickly. For instance, when a customer is shortchanged at the register but the clerk immediately notices the mistake and corrects it, the customer will typically not complain. This makes sense because a) ensuring perfection – perhaps by hiring additional clerks that double-check the change before handing it over to the customer – would be cumbersome and expensive, and because b) the mistakes cause no actual damage, *as long as they are noticed and fixed quickly*. The second point is crucial: if the customer notices the mistake after leaving the store, it will be difficult or impossible to get the money back.

In this paper, we argue that allowing small mistakes could also be a useful approach to fault tolerance in distributed systems. For instance, systems could provide the equivalent of the infamous "five-second rule":[1] when a fault occurs, the system is allowed to produce incorrect outputs for at most R seconds, but must recover and resume correct operation afterwards. We refer to this property as *bounded-time recovery (BTR)*.

At first glance, bounded-time recovery seems very weak, particularly if we consider adversarial faults, since it allows the system to output anything (or nothing) during the R-second recovery period. However, the potential damage depends on how these outputs are used. For instance, many distributed systems interact with some kind of physical system (say, an airplane), and this system often has some inertia or is able to tolerate some faults for other reasons. For instance, the flight control system in an airplane can typically operate within a relatively large flight envelope [61] and is already equipped to handle small disturbances, e.g., because of air turbulence. Because of inertia, a *short* malfunction will not be enough to push the airplane out of this envelope and can thus be tolerated, *as long as the system returns to correct operation quickly enough*. A similar argument holds true for many other cyber-physical systems (CPS) [64, 72].

BTR has at least three potential advantages to offer over existing alternatives, two direct ones and one indirect one. The direct ones are efficiency and support for timeliness. BTR can be more efficient than, say, BFT because it provides weaker guarantees; for instance, detection requires fewer replicas than masking [36], and BTR can use the output of some replicas without waiting for the others to complete. BTR can also guarantee that outputs are timely when an attack is absent, and that an attack can only affect the outputs for a bounded amount of time. This is in direct contrast to most existing fault-tolerance solutions, which tend to sacrifice liveness to preserve safety, and it is an important property for the applications we envision (e.g., CPS), where it is not enough if the system recovers "eventually": real, physical damage can occur if correct outputs are missing too long.

BTR's indirect advantage is an ability to offer more fine-grained responses to faults and attacks. Since BTR guarantees timeliness, a BTR-enabled system must necessarily monitor the workload and the available resources very carefully; thus, when a fault does occur, it can make very detailed tradeoffs to utilize the remaining resources in the best possible way. For instance, many CPS run a mixed workload with tasks of different criticality levels [67] – the CPS on an airplane might run flight control and the in-flight entertainment system. Thus, when a fault occurs, the system can disable some of the less critical tasks and allocate their resources to the more critical ones. This is in contrast to many existing fault-tolerance approaches that treat the workload as a "black box" (and thus protect either all of it or none of it), and that abstract away most of the details of the resources, e.g., the amount of available bandwidth or the network topology.

Building a practical BTR technique involves several interesting challenges – e.g., with respect to scheduling, fault detection, and reconfiguration – and we discuss these in the rest of this paper, along with a sketch of a possible BTR technique. However, our main goal is to make the case for BTR as an interesting addition to the fault-tolerance toolbox that should have useful, practical applications in systems such as CPS.

[1] The five-second rule states that a food item that has been dropped on the floor may still be eaten if it is picked up within five seconds.

2 Case study: CPS

For the purposes of this paper, we use cyber-physical systems (CPS) as our running example. A cyber-physical system contains both digital control and physical sensors and actuators; examples include factory or power plant control systems [54], avionic systems [24], building control systems [22], robots [66], and even self-driving vehicles [7]. Unlike classical embedded systems, which typically have a central controller, CPS are truly distributed systems, with a heterogeneous mix of different nodes that are connected by various networks: even a simple CPS such as a modern (non-self-driving) car contains about a hundred microprocessors [20].

Requirements: CPS require very high reliability: if they fail, or even take too much time to respond to a trigger from their environment, the result can be physical damage or even loss of life. As discussed earlier, *timing is critical*: for instance, when a sensor indicates a pressure increase in some part of the system, the system may need to respond within seconds – e.g., by opening a safety valve – to prevent an explosion. However, somewhat counterintuitively, perfect accuracy is not as critical: the physical part of the system has properties like inertia or thermal capacity, and thus *can tolerate small mistakes or omissions, as long as they are fixed within a bounded amount of time*. Control algorithms can be designed to preserve stability under these conditions [56, 64, 72]. In other words, this domain requires stronger liveness, and potentially weaker safety, than classical fault-tolerance solutions tend to offer, making it suitable for BTR.

Network: The strict timing requirements have consequences for the network: in the presence of an end-to-end deadline for multi-node data flows, queueing delays or packet drops can be devastating. Hence, it is more common to see circuit-switched networks with strict bandwidth reservations, which enable predictable timing and prevent packet drops due to queue overflows. Packets can still be dropped due to transmission errors, but forward error correction (FEC) can be used to minimize this risk where necessary. Thus, stronger synchrony assumptions about the network seem reasonable in this domain.

Processing power: CPUs in this domain are often far less powerful than CPUs in servers and workstations. This is partly due to a desire to reduce cost (or size, weight, power consumption, ...), which encourages system designers to use the least powerful CPU that will do the job, at the lowest possible clock frequency. Indeed, the impact on clock frequency is a common evaluation metric in this domain, e.g., for scheduling techniques. Because of this, developers may be reluctant to accept the cost of techniques like BFT, especially if their strong safety guarantees are not strictly needed.

2.1 System model and threat model

Existing fault-tolerance techniques commonly assume either an asynchronous (or at most weakly synchronous) system model [17, 40] or a simple synchronous one in which processes take abstract "steps" and the details of the network (such as bandwidth and topology) are abstracted away. Neither model is suitable for our purposes: to give timeliness guarantees, we need both synchrony and a more detailed view of the available resources. Therefore, we define our own tentative system model that captures some of the special features of CPS that we have outlined above.

System model: The system consists of a set of *nodes* and a set of *links*. Nodes have a finite processing speed and access to a local clock. For simplicity, we assume that the processing speeds are all the same; some nodes are *sources* or *sinks*, that is, they can receive inputs from, or accept outputs for, the physical world. Each link is connected to some subset of the nodes and has a finite bandwidth. We assume that losses are rare enough to be ignored, and that there is some solution to the babbling-idiot problem [11] – e.g., that the bandwidth of each link is statically allocated between the nodes.

Some of these assumptions are very strong (for instance, there is a rich literature on clock synchronization alone [3, 10, 25, 32, 46]), and in a longer paper we would definitely discuss them in more detail. Here, we can only briefly argue that they do seem reasonable for typical CPS hardware: for instance, buses like CAN support the use of FEC to mask packet corruption [11], and the MAC is often implemented in hardware and thus can enforce bandwidth allocations even if nodes are corrupted.

Workload: For simplicity, we assume a static, periodic workload that can be described as a dataflow graph. (This model clearly does not fit all possible CPS, but seems appropriate for many – e.g., [58].) The system has a period P and releases a set of tasks during each period. Each task requires some inputs from the sources and/or from other tasks, and it sends at least one output to a sink or another task. Each output has a criticality level and a deadline by which it must arrive at the appropriate sink.

Threat model: We assume Byzantine [47] faults – that is, there is an adversary who has compromised some subset of the nodes and has complete control over them. Perhaps surprisingly, this model is considerably stronger than the one CPS typically use: although faults can result (and have resulted [44, 48, 63, 73]) in substantial damage and many CPS must therefore undergo a strict certification process, this process is usually based on crash faults. However, we note that there is growing concern about non-crash faults and attacks, both in the community [15, 16] and from operators and the government [1].

3 Bounded-time recovery

We now offer a first intuitive (but still imperfect) definition of BTR. Assume that, for each node in the system, there is some notion of its expected behavior – for instance, a function that maps a sequence of messages m_i^{in} that the node receives at times t_i^{in} to a sequence of messages m_j^{out} that the node should send at times t_j^{out}. We say that the node is *correct* in an interval $[t_1, t_2]$ if its actual behavior matches the expected behavior, and we say that a fault *manifests* on the node at time t if it is correct in $[0,t)$ but not in $[0,t')$ for any $t' > t$. Similarly, we say that the outputs of the system as a whole (e.g., its commands to the actuators) are correct in an interval $[t_1, t_2]$ if they are consistent with the outputs of a system in which all nodes are correct. Then we can define BTR as follows:

Definition 3.1 (Bounded-time recovery)**.** A system offers recovery with a time bound R if its outputs are correct in any interval $[t_1, t_2]$ such that no fault has manifested in $[t_1 - R, t_2)$.

In other words, the system is allowed to produce incorrect outputs during an interval of length R whenever a fault manifests on a new node.

In practice, BTR would obviously need a bound f on the number of nodes that can become faulty. Note that, if an adversary controls $k \leq f$ nodes, he can trigger a new fault every R seconds and thus potentially force the system to produce bad outputs for kR seconds; thus, if the system has an overall deadline D after which damage can occur in the absence of correct outputs, it seems prudent to set $R := D/f$ rather than $R := D$. Also, our intuitive definition does not yet account for mixed-criticality, but it should not be difficult to extend it, e.g., by allowing a certain set of outputs to fail permanently if the number of faults rises above a certain level.

3.1 Strawman solutions

There are two special cases where BTR closely resembles existing fault-tolerance approaches: for $R = 0$, BTR is analogous to classical fault tolerance – as in BFT [17, 47] – where all faults must be masked; without a hard upper bound on R, BTR closely resembles self-stabilization [28], where the system is simply required to return to correct operation eventually. However, even with $R = 0$, BTR is not quite the same as BFT because a) it has a different system model, with finite resources and a partially connected network, and because b) it gives strong timing guarantees, even in the presence of compromised nodes. Thus, an implementation of BTR always requires a set of detailed schedules for different scenarios to ensure that the timing guarantees can be met.

The difficulty of BTR also depends on the magnitude of R and the amount of resources the system has available. If R is large, the system can simply drop all of its current tasks as soon as a fault is detected, and then perform diagnosis and rescheduling at its leisure; if there are plenty of resources, the system can afford enough replicas for fault tolerance, which of course simplifies recovery (though not necessarily planning). However, recall that CPS are often resource-constrained and tend to have strong timeliness requirements, so we expect the "easy" cases to be less common in practice.

4 Solution sketch

Our approach to BTR is centered around the concept of a *plan*, which is basically a distributed schedule: it maps the tasks from the workload (and some additional tasks, such as replicas) to specific nodes, and it prescribes a schedule for each of the nodes. At runtime, the system is either operating in one of several *modes*, which correspond to a particular plan, or it is executing a *mode transition*, in which the system starts, reconfigures, or shuts down tasks on some of the nodes in order to switch to a new plan. (Mode transitions are triggered by detected faults and are intended to isolate the faulty nodes from the rest of the system.) Together, the plans, and the conditions for switching between them, form the system's *strategy* for responding to faults.

We use four components to achieve BTR: an offline *planner*, which computes a feasible strategy, based on the requested workload, the fault assumptions (such as an upper bound on the number of faulty nodes), and the desired recovery bound R; an online *fault detector* that looks for manifested faults and generates some kind of evidence; an *evidence distributor* that quickly and reliably gets the evidence to the nodes that need to know about it; and a *mode switcher* that executes the appropriate mode transition based on valid evidence of faults. Next, we discuss each of the components in more detail.

4.1 The Planner

Before the system can run a given workload, it must first find a strategy that can ensure BTR. Some representation of the strategy is then installed in each node, so that correct nodes will have a consistent view of it at runtime. Choosing the strategy offline seems safer than dynamic rescheduling at runtime because a) a centralized scheduler would be an obvious target for the adversary, and because b) to guarantee BTR, we would need a time bound *on rescheduling*, which seems difficult to obtain.

The planner first augments the dataflow graph with additional tasks. It adds 1) replicas; 2) checking tasks, which compare the outputs of the replicas to detect faults and generate evidence; and 3) verification tasks, which distribute and verify incoming evidence from other nodes. These tasks all consume resources at runtime and must therefore be scheduled together with the workload tasks – there are no "extra resources" for BTR.

Next, the planner computes a plan for each mode. Each task is mapped to a node; this involves some "hard" constraints – for instance, no two replicas of the same task can run on the same node – but also some heuristics: for instance, putting replicas close to each other may save bandwidth, and putting checking tasks close to replicas can make it easier to detect omission faults. The planner then tries to derive a schedule for each node and a resource allocation for each link. If the system is not schedulable (e.g., because the current mode contains many faulty nodes, and thus few resources), the planner removes some of the less critical tasks and retries.

Challenges: So far, there has not been much work on real-time mixed-criticality scheduling for distributed systems; most of the work we are aware of is for single or multicore CPUs [12]. Moreover, BTR's scheduling problem involves dependencies, which are challenging to support – especially in mixed-criticality systems. Dependencies can arise between tasks (e.g., checking tasks must run after the corresponding replicas) but also between plans. For instance, if plan B would be activated if a fault is detected on node X while the system is running some other plan A, then B must obviously reassign the tasks that were running on X, but it should otherwise change as little as possible. Any extra reassignments will consume resources (e.g., bandwidth for transferring state) and can thus prolong recovery.

Finally, planning has an interesting strategic component. Suppose, for instance, that the planner has already chosen a plan $\Pi_{\{X\}}$ for the case where node X has failed, and is now looking for a plan $\Pi_{\{X,Y\}}$ that can handle an extra fault on node Y. If the planner was not careful when choosing $\Pi_{\{X\}}$, it may be impossible to find a $\Pi_{\{X,Y\}}$ that can be activated quickly enough – for instance, a task with a lot of state may have been moved to a node whose only high-bandwidth connection to the rest of the system is via Y. Thus, computing a strategy is a bit like building a game tree for a game like chess. Techniques like empirical game-theoretic analysis [68, 69] may be useful for finding good strategies efficiently.

4.2 The Fault Detector

Fault detection and diagnosis in distributed systems are interesting problems in their own right [49], particularly when non-crash faults are considered [36, 41]. However, BTR adds an interesting twist: since there are no trusted nodes, the compromised nodes can try to confuse the detector, e.g., by reporting nonexistent faults or by making false statements about the actions of other nodes. Therefore, it is necessary to generate *evidence* of detected faults that other nodes can verify independently.

Challenges: Systems like PeerReview [37] can already generate evidence for asynchronous systems, but synchronous systems – and BTR particularly – present at least three additional challenges. First, BTR additionally requires the detection of timing-related faults (such as doing the right thing at the wrong time). Second, BTR requires a time bound *on detection*; this is difficult because an adversary can break the BTR guarantee simply by delaying the detector, e.g., by running a DoS attack against some of the nodes. The strong assumptions in our system model should be enough to build a countermeasure, but even so, designing and proving the correctness of a concrete protocol that does this seems challenging.

The third challenge is handling omission faults. In contrast to commission faults, there is no direct way to prove that a faulty node *failed* to send – or correctly sign – a required unicast message. Thus, a faulty node may be able to drain substantial resources from the system by constantly failing to send messages and then claiming that the problem is with the recipient. One way to avoid this would be to allow both the sender and the recipient to declare (without further evidence) a problem with the *path* between them; the system could then a) switch to a mode that does not use this particular path, and b) keep track of which paths have been declared problematic. If a node is on a large number of problematic paths, it may be possible to attribute the problem to that node.

4.3 Evidence Distribution

Once a node has detected a fault, the resulting evidence must quickly be distributed to any other nodes that need to be aware of it. (If the system's strategy is compositional, not all nodes will need to know about all faults, at least not immediately.) The distribution process must a) compete for resources with the foreground tasks, b) be completed within bounded time, and c) prevent the adversary from causing delays via DoS, e.g., by flooding the system with bogus evidence.

Challenges: As a first approximation, we can achieve the above properties by reserving some amount of computation and bandwidth for evidence distribution, and by having each node validate incoming evidence before distributing it further. If nodes are required to endorse evidence they distribute, invalid evidence can be counted as evidence against the signer. However, a compromised node can still fabricate evidence that is improperly signed, or that can only be recognized as invalid after a lot of expensive computation – thus, there must be a way to quickly recognize and reject such cases.

4.4 Mode Change

When a node receives evidence of a new fault, it consults the strategy, picks the plan for the new fault pattern, and initiates a mode change to transition to this new plan. This can involve starting new tasks or terminating existing ones, sending or receiving the state of migrating tasks, and adjusting the local schedule.

At first glance, it may seem that some form of global agreement (e.g., via BFT) is needed to reconfigure the system. Agreement would certainly suffice, but it does not seem necessary: since the new plan is a function of the set of faulty nodes, it is sufficient for the nodes to agree on the latter – but (if we ignore physical repair by the administrator) this set is append-only, and, if a node receives valid evidence of a fault on some other node X, it can safely add X to its local set. Thus, as long as all new evidence reaches each correct node, the system should converge to a single, consistent plan.

Challenges: The key challenge here is not convergence itself but rather coordination and timing: if different nodes switch modes at different times, some confusion can briefly result, e.g., when a new task on some node X waits for an input from another node Y that has not yet completed (or even started) its mode transition. Since BTR allows the system to produce incorrect outputs for a limited time, some brief confusion may even be acceptable, as long as evidence distribution and mode transitions are fast enough, but for quick recovery, a more sophisticated solution is needed.

5 Related work

So far, the literature on real-time systems and on non-crash fault tolerance has had few intersections – the former has mostly focused on crash faults, and the latter mostly on asynchronous systems. This disconnect has been pointed out before us, e.g., by Aguilera and Walfish [4], and BTR is an attempt to bridge this gap.

Recovery: The idea of recovering systems from disruptions is not new; indeed, the term "bounded-time recovery" has been used in the database literature [60] to describe a real-time database that can recover from a failure within bounded time. However, most of the recovery work we are aware of is application-specific, such as [18, 19, 35], or focuses exclusively on crash faults, such as [13, 60]. Some systems also support simple forms of recovery, such as rebooting faulty machines [17] or application components [14].

BFT: There is a rich literature on protocols for tolerating Byzantine faults [2, 17, 26, 34, 45], and some of these protocols have been applied to time-critical distributed systems (e.g., in [39, 42, 52]). Many classical BFT protocols are unsuitable for time-critical systems [62], but more recent protocols have improved in this respect [6, 23, 51], although they still do not provide "hard" timing guarantees. In contrast, BTR focuses explicitly on timeliness, which requires different assumptions and a more detailed system model; it also offers different properties, e.g., less masking at lower cost.

ZZ [71] reduces the normal-case overhead of BFT by running only $f+1$ replicas by default, and by changing to agreement only if these replicas disagree. BTR shares this reactive, detection-based approach, but ZZ does not offer timeliness or fine-grained recovery strategies.

Self-stabilization: One way to make a distributed system fault-tolerant is to ensure that it converges to a correct state even if it is started in an incorrect state. This approach was first proposed by Dijkstra [28] and has led to a rich body of work on self-stabilizing systems (see [30, 59]). Much of the early work assumed that faults are benign and cannot handle malicious nodes that might constantly steer the system away from its goal. Recent work [9, 27, 29, 31, 38, 50] has extended the approach to the Byzantine setting, but this line of work tends to use a very different system model that does not consider scheduling, deadlines, task criticality, or complex network topologies; thus, an application in the context of CPS would be difficult.

CPS security: There is some existing work on fault-tolerant real-time systems, such as Mars [43] and DeCoRAM [8], as well as on fault-tolerant and/or reconfigurable control systems [74]. However, most of this work has considered various types of "benign" faults, such as hardware defects, software bugs, or electromagnetic interference. There is also an emerging security-oriented research trend within the control systems and CPS communities (see [33] and the references therein). However, existing solutions either assume a centralized setting (e.g., [70]) or focus more on attacks on the sensors and actuators, and not on the controllers (e.g., [5, 33, 53, 65]).

Accountability: PeerReview [37] can detect node misbehavior in distributed systems and produce evidence of it; however, with one exception, this line of work has focused exclusively on asynchronous systems. The exception is TDR [21], which can detect time-related misbehavior – specifically, covert timing channels – but can neither offer recovery nor a time bound on detection.

Multi-mode systems: Many real-time embedded systems can operate in multiple modes that involve different sets of tasks, and transitioning between modes requires elaborate mode-change protocols (MCPs) [55, 57] to prevent deadline misses and other disruptions. Our recovery approach builds on MCPs, but, to our knowledge, all the existing work on MCPs assumes benign faults and cannot work reliably when the system is compromised.

6 Conclusion

We believe that there is room in the "fault-tolerance toolbox" for an approach that focuses more on timeliness and less on perfect safety, and we have argued that many cyber-physical systems (and perhaps other systems) could benefit from such a approach. We have made a concrete proposal (BTR), and we have sketched a technique that can achieve it. However, some interesting challenges remain, and we have started to address some of them in our ongoing work.

References

[1] NSF/Intel partnership on cyber-physical systems security and privacy (CPS-security). http://www.nsf.gov/pubs/2014/nsf14571/nsf14571.htm.

[2] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *Proc. SOSP*, 2005.

[3] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Partial synchrony based on set timeliness. In *Proc. PODC*, 2009.

[4] M. K. Aguilera and M. Walfish. No time for asynchrony. In *Proc. HotOS*, 2009.

[5] S. Amin, A. A. Cárdenas, and S. S. Sastry. Safe and secure networked control systems under denial-of-service attacks. In *Proc. HSCC*, 2009.

[6] Y. Amir, B. Coan, J. Kirsch, and J. Lane. Prime: Byzantine replication under attack. *IEEE TDSC*, 8(4):564–577, 2011.

[7] Z. Anderson and M. Giovanardi. Self-driving vehicle with integrated active suspension, Oct. 2 2014. US Patent App. 14/242,691.

[8] J. Balasubramanian, A. Gokhale, A. Dubey, F. Wolf, D. C. Schmidt, C. Lu, and C. Gill. Middleware for resource-aware deployment and configuration of fault-tolerant real-time systems. In *Proc. RTAS*, 2010.

[9] M. Ben-Or, D. Dolev, and E. N. Hoch. Fast self-stabilizing Byzantine tolerant digital clock synchronization. In *Proc. PODC*, 2008.

[10] M. Buevich, N. Rajagopal, and A. Rowe. Hardware assisted clock synchronization for real-time sensor networks. In *Proc. RTSS*, 2013.

[11] G. Buja, J. R. Pimentel, and A. Zuccollo. Overcoming babbling-idiot failures in CAN networks: A simple and effective bus guardian solution for the FlexCAN architecture. *IEEE Trans. on Industrial Informatics*, 3(3):225 – 233, Aug. 2007.

[12] A. Burns and R. I. Davis. Mixed criticality systems - a review. In *Tech. Report*, July 2014.

[13] G. Candea and A. Fox. Crash-only software. In *Proc. HotOS*, May 2003.

[14] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot – a technique for cheap recovery. In *Proc. OSDI*, 2004.

[15] A. A. Cárdenas, S. Amin, and S. Sastry. Secure control: Towards survivable cyber-physical systems. In *Proc. ICDCS Workshops*, June 2010.

[16] A. A. Cardenas, T. Roosta, and S. Sastry. Rethinking security properties, threat models, and the design space in sensor networks: A case study in SCADA systems. *Ad Hoc Networks*, 7(8):1434–1447, 2009.

[17] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, 2002.

[18] R. Chandra, T. Kim, M. Shah, N. Narula, and N. Zeldovich. Intrusion recovery for database-backed web applications. In *Proc. SOSP*, Oct. 2011.

[19] R. Chandra, T. Kim, and N. Zeldovich. Asynchronous intrusion recovery for interconnected web services. In *Proc. SOSP*, Nov. 2013.

[20] R. N. Charette. This car runs on code. IEEE Spectrum, 2009. http://spectrum.ieee.org/transportation/systems/this-car-runs-on-code.

[21] A. Chen, W. B. Moore, H. Xiao, A. Haeberlen, L. T. X. Phan, M. Sherr, and W. Zhou. Detecting covert timing channels with time-deterministic replay. In *Proc. OSDI*, Oct. 2014.

[22] J. Chen, R. Tan, G. Xing, and X. Wang. PTEC: A system for predictive thermal and energy control in data centers. In *Proc. RTSS*, pages 218–227, 2014.

[23] A. Clement, E. L. Wong, L. Alvisi, M. Dahlin, and M. Marchetti. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *Proc. NSDI*, 2009.

[24] R. P. Collinson. *Introduction to avionics systems*. Springer, 2011.

[25] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Googles globally distributed database. *ACM TOCS*, 31(3):8, 2013.

[26] J. A. Cowling, D. S. Myers, B. Liskov, R. Rodrigues, and L. Shrira. HQ replication: A hybrid quorum protocol for Byzantine fault tolerance. In *Proc. OSDI*, 2006.

[27] A. Daliot and D. Dolev. Self-stabilization of Byzantine protocols. In *Proc. SSS*, 2005.

[28] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, Nov. 1974.

[29] D. Dolev and E. N. Hoch. Byzantine self-stabilizing pulse in a bounded-delay model. In *Proc. SSS*, 2007.

[30] S. Dolev. *Self-Stabiliaztion*. MIT Press, 2000.

[31] S. Dolev and J. L. Welch. Self-stabilizing clock synchronization in the presence of Byzantine faults. *J. ACM*, 51(5):780–799, Sept. 2004.

[32] J. Elson, L. Girod, and D. Estrin. Fine-grained network time synchronization using reference broadcasts. *ACM SIGOPS OS Review*, 36(SI):147–163, 2002.

[33] H. Fawzi, P. Tabuada, and S. Diggavi. Secure estimation and control for cyber-physical systems under adversarial attacks. *Automatic Control, IEEE Transactions on*, 59(6):1454–1467, June 2014.

[34] R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić. The next 700 BFT protocols. In *Proc. EuroSys*, 2010.

[35] Z. Guo, S. McDirmid, M. Yang, L. Zhuang, P. Zhang, Y. Luo, T. Bergan, P. Bodik, M. Musuvathi, Z. Zhang, and L. Zhou. Failure recovery: When the cure is worse than the disease. In *Proc. HotOS*, May 2013.

[36] A. Haeberlen and P. Kuznetsov. The Fault Detection Problem. In *Proc. OPODIS*, Dec. 2009.

[37] A. Haeberlen, P. Kuznetsov, and P. Druschel. PeerReview: Practical accountability for distributed systems. In *Proc. SOSP*, Oct. 2007.

[38] E. N. Hoch, D. Dolev, and A. Daliot. Self-stabilizing Byzantine digital clock synchronization. In *Proc. SSS*, 2006.

[39] K. Hoyme and K. Driscoll. SAFEbus. In *Proceedings of the Digital Avionics Systems Conference (DASC)*, 1992.

[40] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schroder-Preikschat, and K. Stengel. CheapBFT: Resource-efficient byzantine fault tolerance. In *Proc. EuroSys*, 2012.

[41] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. Byzantine fault detectors for solving consensus. *The Computer Journal*, 46(1):16–35, 2003.

[42] H. Kopetz and G. Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, 2003.

[43] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger. Distributed fault-tolerant real-time systems: The Mars approach. *IEEE Micro*, 9(1):25–40, 1989.

[44] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage. Experimental security analysis of a modern automobile. In *Proc. IEEE S & P*, May 2010.

[45] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. L. Wong. Zyzzyva: Speculative Byzantine fault tolerance. *ACM TOCS*, 27(4), 2009.

[46] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[47] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

[48] R. Langner. Stuxnet: Dissecting a cyberwarfare weapon. *Security & Privacy, IEEE*, 9(3):49–51, 2011. ID: 1.

[49] J. B. Leners, H. Wu, W.-L. Hung, M. K. Aguilera, and M. Walfish. Detecting failures in distributed systems with the falcon spy network. In *Proc. SOSP*, 2011.

[50] M. R. Malekpour. A Byzantine-fault tolerant self-stabilizing protocol for distributed clock synchronization systems. In *Proc. SSS*, 2006.

[51] Z. Milosevic, M. Biely, and A. Schiper. Bounded delay in Byzantine-tolerant state machine replication. In *Proc. SRDS*, Sept. 2013.

[52] P. Miner. Analysis of the SPIDER fault-tolerance protocols. In *Proceedings of the NASA Langley Formal Methods Workshop (LFM)*, 2000.

[53] Y. Mo and B. Sinopoli. Secure control against replay attacks. In *Proceedings of the Allerton Conference on Communication, Control, and Computing*, 2009.

[54] NIST. Guide to supervisory control and data acquisition (SCADA) and industrial control systems security, 2006. `https://www.dhs.gov/sites/default/files/publications/csd-nist-guidetosupervisoryanddataccquisition-scadaandindustrialcontrolsystemssecurity-2007.pdf`.

[55] L. T. X. Phan, I. Lee, and O. Sokolsky. A semantic framework for mode change protocols. In *Proc. RTAS*, 2011.

[56] P. Ramanathan and M. Hamdaoui. A dynamic priority assignment technique for streams with (m, k)-firm deadlines. *IEEE Transactions on Computers*, 44(12):1443–1451, 1995.

[57] J. Real and A. Crespo. Mode change protocols for real-time systems: A survey and a new proposal. *Real-Time Systems*, 26:161–197, 2004.

[58] N. R. Satish, K. Ravindran, and K. Keutzer. Scheduling task dependence graphs with variable task execution times onto heterogeneous multiprocessors. In *Proc. EMSOFT*, New York, NY, USA, 2008.

[59] M. Schneider. Self-stabilization. *ACM Computing Surveys (CSUR)*, 25(1):45–67, 1993.

[60] L. C. Shu, J. A. Stankovic, and S. H. Son. Achieving bounded and predictable recovery using real-time logging. In *Proc. RTAS*, 2002.

[61] E. J. Sinclair. The army aviator's handbook for maneuvering flight and power management. U.S. Army Aviation Branch, 2005.

[62] A. Singh, T. Das, P. Maniatis, P. Druschel, and T. Roscoe. BFT protocols under fire. In *Proc. NSDI*, 2008.

[63] J. Slay and M. Miller. Lessons learned from the maroochy water breach. In *IFIP International Federation for Information Processing*, 2008.

[64] D. Soudbakhsh, L. T. X. Phan, A. Annaswamy, O. Sokolsky, and I. Lee. Co-design of control and platform with dropped signals. In *Proc. ICCPS*, Apr. 2013.

[65] A. Teixeira, D. Pérez, H. Sandberg, and K. H. Johansson. Attack models and scenarios for networked control systems. In *Proc. HiCoNS*, 2012.

[66] D. Tesar. Robot and robot actuator module therefor, Oct. 18 1994. US Patent 5,355,743.

[67] S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Proc. RTSS*, 2007.

[68] M. P. Wellman. Methods for empirical game-theoretic analysis. In *Proc. Natl. Conference on Artificial Intelligence*, volume 21, page 1552, 2006.

[69] M. P. Wellman and A. Prakash. Empirical game-theoretic analysis of an adaptive cyber-defense scenario (preliminary report). In *Decision and Game Theory for Security*, pages 43–58. Springer, 2014.

[70] T. Wongpiromsarn, U. Topcu, and R. Murray. Receding horizon temporal logic planning for dynamical systems. In *Proc. IEEE CDC*, 2009.

[71] T. Wood, R. Singh, A. Venkataramani, P. Shenoy, and E. Cecchet. ZZ and the art of practical BFT execution. In *Proc. EuroSys*, 2011.

[72] M. Yu, L. Wang, T. Chu, and F. Hao. Stabilization of networked control systems with data packet dropout and transmission delays: Continuous-time case. *European Journal of Control*, 11(1):40–49, 2005.

[73] K. Zetter. A cyberattack has caused confirmed physical damage for the second time ever. Wired.com, January 8, 2015; `http://www.wired.com/2015/01/german-steel-mill-hack-destruction/`.

[74] Y. Zhang and J. Jiang. Bibliographical review on reconfigurable fault-tolerant control systems. *Annual reviews in control*, (32):229–252, 2008.