A Network-Agnostic Approach to Enforcing Collision-Free Time-Triggered Communication

Abstract—**Collision-free time-triggered communication in distributed safety- and real-time-critical systems relies on approximately synchronized clocks, a-priori-defined communication schedules, and network guardians, synchronized in the same manner, which inhibit a node's network access outside scheduled times. However, the ever-increasing complexity and interconnectivity of such systems, potentially deploying multiple types of real- and non-real-time networks, render using contemporary network-aware guardians unsuitable: firstly, significant cost, complexity and certification efforts are incurred in developing new network protocol and topology specific guardian solutions. Secondly, contemporary network guardians lack the means to protect against repetitive cyberattacks that exhaust system synchrony. Many safety-critical systems operate over extended mission times, in harsh environments, and in an unattended manner. Hence, their vulnerability to accidental faults and targeted value- and time-domain attacks cannot be precluded, and their safe operation necessitates preparing them to tolerate accidental as well as maliciously induced faults.**

In this paper, we investigate a novel class of time-domain attacks, aimed at exhausting nodes by tampering with the synchrony of their network-agnostic guardians. We counter the attacks by introducing *SyncGuard*, the first, network-agnostic and time-domain attack-resilient guardian. *SyncGuard*-equipped systems avoid synchrony exhaustion attacks by jointly coordinating network access and node-rejuvenation.

Index Terms—**safety-critical systems, time-triggered systems, intrusion-tolerance, resilient network access, node rejuvenation**

I. Introduction

Many real-time embedded systems perform safety-critical functionalities in application areas that, by their nature, are distributed. Examples include energy grids, water management, autonomous driving, and flight and navigation control. All these examples, have stringent timeliness requirements, but must also be able to tolerate faults, whether of an accidental or intentionally malicious nature. In fact, all of the above areas have already been targeted by cyberattacks [1]–[9] and we must build systems for them that are resilient to future attacks.

A renowned paradigm for simultaneously meeting strict timeliness guarantees and fault tolerance is the time-triggered paradigm [10], [11]. Time-triggered protocols (e.g., TTP [12], FlexRay [13], TTEthernet [14]) establish timely communication by implementing time-division multiple access (TDMA) as medium access strategy. Nodes collaborate based on a common notion of global time, leveraging their knowledge about the real-time communication schedule, to access the medium only during their assigned TDMA slots. Nodes maintain a consistent notion of global time by approximately synchronizing their local clocks and a handful of subsystems, implementing one or more of the above-mentioned time-triggered protocols. These protocols further equip the system with network components, called guardians [15]–[18], that prevent nodes from accessing the network outside their assigned TDMA slots. Failure to prevent out-of-slot communication can have devastating effects, such as fail-safe stopping systems for extended periods of time to prevent more catastrophic consequences of missing communication deadlines.

Unfortunately, the above mechanisms no longer suffice for ensuring timeliness and fault and intrusion tolerance in today's systems. Safety-critical systems have been emerging from closed to open systems. They run increasingly interconnected applications and implement increasingly complex functionality, including a combination of the above-mentioned real-time communication protocols alongside other protocols for non-real-time communication. Moreover, they are increasingly deployed in harsh environments where they are exposed to increasing fault rates and targeted attacks. At the same time, we expect them to operate for prolonged mission times while being rarely attended to, if at all.

Network guardians have proven valuable in the design of safety-critical systems as they convert the behavior of timing faulty nodes into fail silence from the perspective of other nodes in the system. In particular, they protect other (healthy) nodes against the consequences of such faults by blocking any transmission outside a node's allocated TDMA slots. However, contemporary guardian designs are network-aware, i.e., they are built for a specific network interface, communication protocol and topology. They participate in sending and receiving messages (in particular to re-synchronize their local clocks to the maintained global one). They, therefore, include the full functionality of a networked component, including protocol stacks, drivers, etc. Aside from the complexity that this necessarily trusted functionality adds, network-aware guardians also come with the costs of having to implement and certify their functionality each time the network and/or topology changes. In particular, for systems that combine more than one type of network, these costs may be significant. This is why in this work we leverage *network-agnostic* guardians as an essential building block for a time-domain attack-resilient safety-critical system.

Rather than replicating all communication infrastructure that is also present in the node, network-agnostic guardians use this infrastructure to synchronize with the global time base and possibly communicate with other guardians and their nodes. With perpetually correctly synchronized clocks, a network-agnostic guardian would receive messages through the controlled node and possibly send messages back in the node's scheduled communication slots at which time the guardian would grant the node access to the network. Unfortunately, such permanent synchrony cannot be maintained over extended periods of time, in particular, when nodes are susceptible to accidental faults or when they become the victim of partially successful cyberattacks.

In this paper, we investigate a novel class of time-domain attacks, aimed at exhausting nodes by tampering with the synchrony of their network-agnostic guardians. We counteract the attacks by introducing *SyncGuard*, the first, network-agnostic and time-domain attack-resilient guardian. We do so by extending the responsibility of guardians to not only prevent nodes from accessing the network outside scheduled slots, but also to initiate and coordinate the rejuvenation of nodes. Rejuvenation aims at returning nodes to a state that is at least as secure as initially. This is achieved by resetting nodes, thereby removing active adversaries, and by returning them to a known good state. Coordinating when nodes rejuvenate and repeatedly initiating this process, guardians are able to guarantee the absence of intentionally malicious actors, at least for a short while, but long enough to re-synchronize with high probability. They are also able to abort re-synchronization before this guaranteed safe period expires or to fail-stop nodes that are permanently damaged due to accidental causes. Therefore, by combining rejuvenation with attempts to re-synchronize and repeating both as long as accidental fault syndromes allow, guardians obtain the chance to re-synchronize and prevent malicious exhaustion of resources, leaving accidental faults the only reason why nodes may fail by stopping to send.

This work presents the following contributions:

1) An investigation of time domain attacks targeting the synchrony of network-agnostic guardians;
2) The design of *SyncGuard*, the first network-agnostic guardian for enforcing collision-free time-triggered communication and resource exhaustion safe operation.

*SyncGuard*s not only provide compatibility with different time-triggered network topologies and protocols but also provide real support for the runtime reconfiguration of network access schedules based on pre-runtime configured operational modes. This comes at the cost of an acceptable reduction of available network data bandwidth.

The remainder of the paper is organized as follows. We uncover the underlying concepts for time-triggered communication and its enforcement in Section II. In Section III, we state our assumptions on the system structure and then discuss, in Section IV, time domain attacks against network-agnostic guardians. We describe our fault model in Section V and then present, in Section VI, the design of *SyncGuard*s and how they can correctly deliver their intended functionality by recurrently rejuvenating their nodes and by exchanging messages among each other. In Section VII, we describe how nodes characterized by short rejuvenation times can rejuvenate more frequently and replace feedback exchange with static modifications to TDMA slots. We finally conclude in Section VIII.

II. Background and Related Work

This section introduces the relevant background and discusses related work. We focus on clock synchronization as a prerequisite of time-triggered communication, slot enforcement mechanisms, rejuvenation, and how the latter achieves exhaustion safety.

A. Clock Synchronization and Time-Triggered Communication

In a distributed system, the local clocks of nodes tend to drift apart and require recurrent synchronization to maintain a consistent view of global time. Deterministic clock synchronization algorithms [19]–[21] leverage the fact that real-time communication provides bounded network latency to keep clocks internally synchronized within a bounded interval, called *precision* or *synchronization error* (π). In other words, the maximum difference between any two correct clocks is no larger than the interval π. Although clock drift and network latency are bounded from both ends, their exact value is uncertain. Nodes rely on estimates of other nodes' clocks, which contain a reading error due to these uncertainties. Therefore, perfect clock synchronization is not attainable [22].

Following the sparse time concept [23] and TDMA-based medium access strategy, time-triggered communication shares the network channel bandwidth among the different system nodes by dividing global time into disjoint intervals called TDMA slots. Each node is assigned a unique TDMA slot to accommodate the dissemination of its message on the shared communication channel. The pattern of TDMA slots, which contains a unique TDMA slot for every message on the shared communication channel, represents the time-triggered schedule for that channel. During runtime, the consistent perception of global time and knowledge of the time-triggered schedule steer the operation of the system, where all nodes are able to infer message transmission or reception time.

The process of accepting a received time-triggered message and using its arrival time for clock correction is shown in Figure 1. Here, node n_x is the sender of message msg_x and node n_w is one of the receiving nodes. The planned message transmission or reception time at a given TDMA slot is called the action time (t_{at}), which is an agreed-upon global point in time stored in the communication schedule. However, since clocks of nodes are approximately synchronized within precision π, n_x calculates a delayed action point ($t^x_{dat} = t^x_{at} + \pi$) for when it shall start transmitting msg_x such that all correct receivers are guaranteed to perceive the start of its TDMA slot. Similarly, n_w calculates a delayed action point ($t^w_{dat} = t^w_{at} + \pi + T^{est}_{COMM}$) for when it expects msg_x to arrive based on the clock precision and channel delay estimate (T^{est}_{COMM}). Node n_w additionally uses the calculated t^w_{dat} to define an acceptance window (T_{AW}) during which reception of msg_x should be considered valid from n_w's point of view. Since clocks drift in either direction, the minimum size for the acceptance window should be 2π. If the message is received within the acceptance window, the estimated time difference (Δ^{est}_{DIFF}) between the expected t^w_{dat} and actual reception time (t^w_{rcv}) of msg_x is used by n_w to calculate a correction term for its clock. Notice that the estimated time difference Δ^{est}_{DIFF} contains a reading error (compared to Δ_{DIFF}) due to the uncertainty in network communication delays $\upsilon_{comm} = T^{max}_{COMM} - T^{min}_{COMM}$ and clock drift, which prevents clocks from converging closer than a certain bound. Node n_w then proceeds to calculate the average of time differences derived from message reception time on all communication channels and possibly from multiple senders (e.g., from $3F + 1$ nodes to tolerate readings from up to F arbitrary faulty nodes) and corrects its clock based on the convergence function of the clock synchronization algorithm.

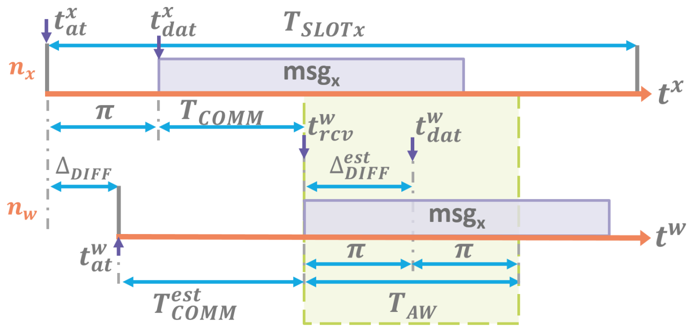


Fig. 1: Capturing clock difference and message acceptance window in time-triggered communication

B. Network and Protocol-Aware Guardians

Existing approaches for enforcing conformance to time-triggered communication schedules encompass encapsulating the network access control functionality in independent and protocol-specialized hardware components called guardians. Guardians are organized as on-chip or off-chip components at each node and/or communication channel and control access to the latter. The rationale behind including a guardian component is to convert a node's faulty behavior in the time domain to fail-silence (as perceived by other system nodes). This is made possible by equipping the guardian with the knowledge and details of the communication protocol (e.g., communication schedule, clock synchronization, channel coding and protocol frames). Therefore, existing guardian designs mainly differ by the targeted network topology and amount of shared resources with the guarded node/s. Designs that encompass the guardian functionality in a fully independent networked component [13], [16], [18] are typically complex and expensive but provide more guardian fault-independence compared to cheaper designs. Less complex guardian designs, on the other hand, sacrifice fault tolerance features for cost reduction. The local bus guardian of the time-triggered protocol [15] is one such design. It depends on the node communication controller to periodically synchronize its clock. Hence, if guarded nodes become malicious, their bus guardians would not be able to rule out collisions. The TEA protocol [24] suggests placing two different nodes on the same chip and use their communication controllers as guardians to each other. This introduced dependency puts both nodes in the same fault containment region (i.e., it cannot be ruled out that the failure of one node causes a failure in the other). The BRAIN protocol [25] relies on its ring topology and a node's direct neighbors to efficiently achieve the guardian functionality by having the latter refrain from relaying out-of-slot messages on the ring.

In this work, we present a distributed solution of network-agnostic guardians (*SyncGuards*) that are placed locally to each node and communicate through it. Unlike previous works we place no restrictions on network topology but require at least $F + 1$ disjoint paths to tolerate F faulty channels. *SyncGuards* additionally incorporate the functionality to trigger and coordinate the rejuvenation of their nodes, which, as shall become clear in the following sections, enable them to safely re-synchronize even when nodes may fail arbitrarily.

C. Exhaustion Safety and Rejuvenation

As nodes fail over time (e.g., due to adversarial action), the system's ability to mask additional faults behind a healthy majority is increasingly threatened up to the point where fault assumptions get violated. Exhaustion safety [26] aims to characterize the absence of such a loss of healthy resources over time and rejuvenation is a technique to achieve this. Proactively rejuvenating all replicas of a system in a periodic manner that is faster than an adversary can compromise more than the tolerated fault threshold (e.g., by recovering their code and state to a known-good instance) ensures exhaustion safety. Reactively rejuvenating faulty replicas once detected [27], [28] removes possible adversarial influence on them and restores the system's ability to tolerate faults faster.

A replica group (e.g., comprised of $2F + 1$ nodes in the case of triple modular redundancy (TMR)), which implements a complex functionality, typically has long rejuvenation times due to cold-start effects when starting up and re-initializing nodes from their initial state. For example, the re-initialization time for Apollo's autonomous driving perception stack is in the order of several seconds [29]. Therefore, the availability of the service delivered by the replica group and its ability to mask faults requires careful coordination when its nodes may rejuvenate. Another consequence of periodic and lengthy rejuvenation is the necessity to increase the number of replicas in the group by an additional K replicas. During runtime, all $N = 2F + K + 1$ replicas periodically execute an offline generated proactive rejuvenation schedule, which comprises $G = \left\lceil \frac{N}{K} \right\rceil$ rejuvenation slots. During each such slot, up to K replicas of the group may rejuvenate in parallel. Figure 2 shows an example of a proactive rejuvenation scheme for a group with $N = 5$ and $K = 2$. A replica node n_x of the group is assigned to proactive rejuvenation slot PRS_w if $(x \mod G == w)$, where $x = \{0, \ldots, N - 1\}$ and $w = \{0, \ldots, G - 1\}$ indicate the unique index of the replica and rejuvenation slot respectively. The size of the rejuvenation slot T_{RS} needs to be set long enough to accommodate the worst-case node rejuvenation time $T_{WCRejuv}$, which includes the time to re-initialize the node and re-integrate it with other nodes in system. As a result, a lower bound on the rejuvenation period is calculated as: $T_{PRejuv} \geq G \times T_{RS}$.

Furthermore, exhaustion safety for a replica group can be guaranteed, according to Sousa [26], if the interval between two rejuvenations of any of its nodes does not exceed the minimum time to compromise $F+1$ nodes ($T^{min}_{Exhaust}$), which bounds T_{PRejuv} from above, i.e, $T_{PRejuv} \leq T^{min}_{Exhaust} - T_{RS}$.

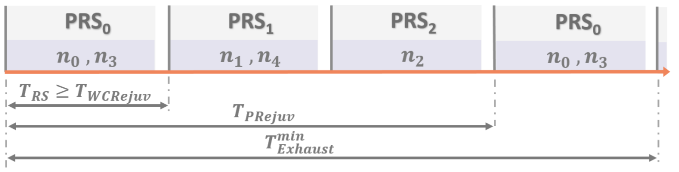
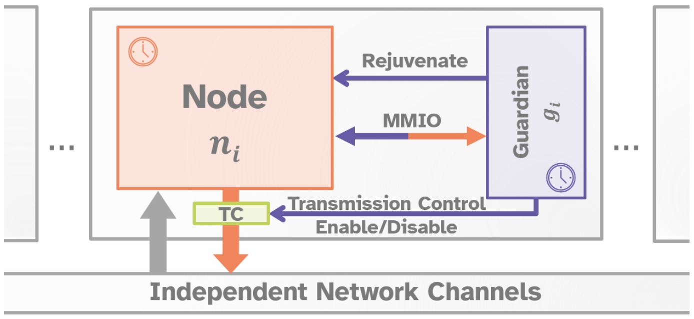


Fig. 2: Proactive rejuvenation of a group comprising $N = 5$ replicas, where up to $K = 2$ may rejuvenate in parallel.

III. System Model

We consider real-time distributed systems (as shown in Figure 3), which are comprised of N nodes connected via a time-triggered network. The network consists of C independent communication channels that implement TDMA as a medium access strategy. The role of a node $n_i | i \in \{0, \ldots, N-1\}$ is to participate in the execution of the distributed real-time application. Access to the network and rejuvenation of node n_i is coordinated through a novel guardian component g_i, which we call *SyncGuard*. The guardian *SyncGuard* is network-agnostic, i.e., in contrast to a network-aware guardian, *SyncGuard* has no understanding of the utilized network protocol and topology. Therefore, guardian g_i interfaces with n_i through a memory-mapped I/O interface and must involve this node for the purpose of exchanging synchronization messages.

Additionally, g_i maintains its own copy of n_i's time-triggered communication and proactive rejuvenation schedules, which g_i use to track the system state in relation to global time. Based on its view of global time, g_i enables or disables n_i's send interface to a given communication channel through the transmission controller (TC) to enforce that n_i accesses the channel only during its allotted TDMA slots. Similarly, g_i uses its view of global time and system state to trigger the rejuvenation of n_i.



Node
n_i
Rejuvenate
MMIO
Guardian g_i
TC
Transmission Control Enable/Disable
...
...
Independent Network Channels

Fig. 3: Distributed system of N nodes communicating over a time-triggered network, whose access and rejuvenation are coordinated through a per-node *network-agnostic* guardian

IV. Potential Timing Attacks

In this section, we discuss potential attacks that hamper the synchrony of network agnostic guardians. We consider the system model presented in Section III and adopt a similar approach of formalizing the notion of time and clocks as in [19]. We distinguish the following two time notions: *real-time* $\mathbb{R}$ as an abstract global time quantity, and *local-time* $\mathbb{C}$ as node or guardian local time. A clock is formally a function $C() : \mathbb{R} \rightarrow \mathbb{C}$, such that $C_{g_x}(t)$ returns the local clock value of g_x at real-time instant t. Following clock synchronization principles (see Section II-A), two guardians g_x and g_w are considered synchronized if Equation 1 is satisfied, i.e., their clocks at any real-time instant t are no more than the precision π apart.

$$\forall t \in \mathbb{R}, \; |C_{g_x}(t) - C_{g_w}(t)| \leq \pi \tag{1}$$

We assume that a node n_x and its *network-agnostic* guardian g_x are initially synchronized with other nodes and guardians in the system and are in agreement on global time. However, a clever adversary successfully compromised n_x, and now attempts to prevent g_x from re-aligning its clock with global time by mounting timing attacks on it. We consider the re-synchronization problem under the assumption that re-synchronization must not interfere with any message healthy replicas send over the network. Such interference may be fatal in highly-critical scenarios where messages must be transmitted before their deadline to avoid system failure, in particular if resend-after-collision is not possible.

A. Omission Attacks

As discussed earlier, since clocks drift away from each other, re-synchronizing to a remote guardian g_w requires g_x to capture information about the state of g_w's clock by receiving a synchronization message from it. However, g_x is network-agnostic, i.e., it must rely on n_x to relay g_w's message to it. The malicious n_x can, nevertheless, omit to relay any such messages to g_x, which can lead to the violation of the relation in Equation 1. Hence, without realigning g_x's clock to global time, g_x cannot rule out collisions if it were to allow n_x to transmit. Under such scenario, g_x can reactively rejuvenate n_x to regain an opportunity to resynchronize.

Note that g_x can not discern whether the sender n_w did not send synchronization messages, whether n_w crashed or whether n_x omitted relaying the message to g_x. In the presence of omission faults, the situation where n_w crashed or did not send the message can be compensated by re-synchronizing to any one of $F+1$ synchronization messages, where F is a parameter and an upper bound on the number of such omission faults that the system can tolerate.

B. Replay Attacks

Replay may cause the repeated or deferred delivery of the most recent synchronization message, the synchronization messages from previous synchronization rounds or any combination of the two. Thus, malicious n_x can present g_x with an outdated but otherwise correct view of global time by replaying synchronization messages. If g_x adopts this view, it may allow n_x to send outside the slots allocated to it. To mount a replay attack, n_x injects an arbitrary delay $T_{INJECTED}$

alongside the relay delay T_{RELAY} before delivering g_w's synchronization message to g_x as follows:

$$C_{g_x}(t^w_{rcv}) = C_{n_x}(t^w_{rcv}) + T_{RELAY} + T_{INJECTED}$$

In other words, n_x can influence the value of $C_{g_x}(t^w_{rcv})$, which denotes the reception time of g_w's message according to g_x's clock and which g_x uses to correct its clock (see Section II-A).

Since the initially synchronized g_x can detect the omission of synchronization messages as well as the replay of an old sequence of authentic messages (by leveraging cryptographic means and knowledge of communication schedule), it can indefinitely block n_x from transmitting or trigger its rejuvenation. Therefore, instead of exhausting nodes by causing them to fail silently, we consider a replay attack with the main goal of allowing n_x to transmit partially outside its allocated slot, which leads to collisions and/or asymmetric reception.

To undetectably veer g_x's local view away from global time, node n_x systematically delays the delivery of synchronization messages to g_x by exploiting the variance between best- and worst-case communication latencies and while still meeting the acceptance window that g_x has for these messages. We refer to this timing attack as *acceptance window attack*. Suppose that g_x is scheduled to receive a synchronization message from remote guardian g_w at global time t^w_{at} and to use its reception time to bring its clock closer to g_w's. As per the time-triggered approach (see Section II-A), g_x would normally accept g_w's message only if its reception time $C_{g_x}(t^w_{rcv})$ meets the following acceptance window check:

$$C_{g_x}(t^w_{dat}) - \pi \leq C_{g_x}(t^w_{rcv}) \leq C_{g_x}(t^w_{dat}) + \pi \qquad (2)$$

Now, consider the following two scenarios:
Scenario1: n_x delivers all synchronization messages to its guardian g_x in a timely manner such that they satisfy Equation 2 and hence, g_x's clock is corrected to tick at a rate that keeps it in agreement with the global notion of time.

Scenario2: n_x starts a virtual time base V initialized with its own clock and delays the delivery of g_w's message (and other messages) to g_x by $T_{Injected} = C_{n_x}(t^w_{rcv}) - C_{n_x}(t^w_{dat})$. Since the clocks of n_x and g_x should satisfy Equation 1 at this point, all the delayed synchronization messages are guaranteed to satisfy Equation 2. At the same time, n_x calculates the real correction terms from the received synchronization messages to know the direction global time is heading (i.e. whether it ticks at a slower or faster rate). Now, for every subsequent re-synchronization round, n_x corrects its virtual clock and that of g_x with the maximum allowed clock correction term κ and with opposite polarity to where global time is heading. That is, node n_x replays synchronization messages to guardian g_x with an added delay of $C_V(t_{rcv}) - C_V(t_{dat}) \pm \kappa$, where the first two terms are represented using the value of the virtual clock V. The injected delay accumulates throughout multiple re-synchronization rounds (as the virtual time base diverges from the global one) such that Equation 1 no longer holds.
Since g_x cannot discern between the above scenarios, the safe re-synchronization of g_x cannot be guaranteed without causing interference. The situation worsens if g_x loses synchrony or tries to join a network cluster, since g_x can neither use an acceptance window check nor can it deter n_x from injecting an arbitrary delay before the defined sequence of messages in the schedule. Therefore, enforcing collision-free network access requires guardians to proactively rejuvenate their nodes to gain an initial interval where malicious activity on them is absent and be able to safely re-synchronize. Additionally, the inability of network-agnostic guardians to proactively deter acceptance window attacks, require them to exchange messages among each other for the purpose of assessing the timeliness with regard to meeting acceptance windows, to then reactively rejuvenate detected faulty nodes. Moreover, from the discussed attacks, it is clear that a node completely controls the delivery time of synchronization messages from other nodes to its network-agnostic guardian. Therefore, it is more efficient to have the node synchronize its guardian clock to global time, thereby eliminating the overhead of storing for every synchronization message of every remote node on every channel in the re-synchronization round all parameters related to calculating its acceptance window at each guardian.

C. *Asymmetric Reception Attacks*

A new attack emerges when network-agnostic guardians exchange messages with required local input from their nodes. Since meeting an acceptance window is a local input of a node, correct nodes and guardians have no way to discern between faulty transmitter that intentionally cause an asymmetric reception (by first de-synchronizing its node from a subset of nodes via acceptance window attacks) or faulty receiver that lies about whether the transmitted message meets its acceptance window. Consequently, while network-agnostic guardians may still be able to enforce no interference between TDMA slots, they cannot ensure that messages sent are received within the acceptance window of all correct receivers. Note, however, that symmetric reception of messages can still be attained if nodes implement any known multi-round reliable broadcast or multicast protocols [30], [31].

V. Fault model

We adopt the fault model introduced by Sousa et al. [26] in assuming the presence of strong adversaries (and accidental faults). More precisely, we assume nodes can become repeatedly faulty due to accidental or intentionally malicious reasons, but during any sliding window of length T_{PRejuv} at most F new faults occur. Moreover, T_{PRejuv} defines the periodicity of proactively rejuvenating replica nodes in the system. As described in Section II-C, the choice of T_{PRejuv} is inferred from the maximum time needed to rejuvenate replica groups as well as the minimum time to failure and compromise of nodes. We further assume that the minimal time adversaries need to compromise an individual replica T^{Ind}_{Compr} (e.g., the time to again mount an attack) is not arbitrarily small. This can be achieved by using techniques such as diversification [32] and obfuscation [33]–[35], the former strengthens independence between replicas and the latter helps with rendering any gained knowledge from successful previous infiltrations futile. We further assume that the deployed obfuscation techniques suffice for protecting replicas diversity throughout the mission time.

We assume critical functionality to be replicated and conclude, following the argument in [26], that replica groups, continue to be able to mask faults, provided all replicas of such groups are recurrently rejuvenated faster than the time to compromise more than F replicas, and that rejuvenated nodes can safely re-join the system, which requires re-synchronizing their guardians. Ensuring the latter despite the previously discussed timing attacks is the main contribution of this paper.

Whereas nodes can fail in an arbitrary and possibly intentionally malicious and Byzantine manner, we assume *SyncGuards* and TCs to only fail by silencing the node in the process. The simplicity of *SyncGuard* defines its trustworthiness and as such the coverage of the assumption that it will not fail arbitrarily, which turns *SyncGuards* into trusted-trustworthy components [36]. We consider the failure of *SyncGuards* and TCs as well as permanent node damage to be rare events. We make no further timeliness or availability claims once the system's (spare) resources are exhausted due to such failures.

To the extent that cryptography is involved in the protocols we consider, we make the usual assumptions about the strengths of applied algorithms and the protection of involved credentials. That is, adversaries cannot get hold of keys actively used by trusted components (including the guardians) and they cannot break the ciphers without such keys.

We shall also assume that the network is sufficiently resilient against accidental and intentionally malicious faults such that adversaries cannot delay or otherwise influence the delivery of messages between nodes. Attackers may influence message forwarding to guardians only through compromising nodes.

VI. *SyncGuard* Design

In this section, we describe the design of our network-agnostic guardian (*SyncGuard*) and how it can be safely re-synchronized and integrated with nodes in the system.

A *SyncGuard* contains, as shown in Figure 4, the following components: 1) a state machine, which implements the network access and node rejuvenation functionality; 2) a memory element and interfaces (NGI and CSI), which are used to store *SyncGuard* configurations and exchange messages between nodes and *SyncGuards*; 3) an oscillator and a timer, which give *SyncGuard* its own notion of time progression and drive the state machine; 4) cryptographic and arithmetic unit,

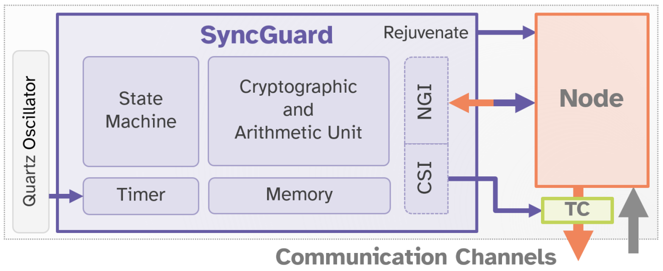


Fig. 4: *SyncGuard* structure

which analyzes and ensures the authenticity and integrity of exchanged *SyncGuard* messages.

A. *SyncGuard Cycle and State Machine*

As discussed in Section II-A, clock synchronization is essential to maintain a consistent view about global time. Therefore, nodes must periodically synchronize the clocks of their *SyncGuards* for the latter to remain within the assumed clock precision π. Additionally, *SyncGuards* have to remain in synchrony w.r.t. system state development in terms of network communication and node rejuvenation progress, which necessitates the exchange of messages.

As illustrated in Figure 5, *SyncGuards* operate, due to the above-mentioned reasons, in a cycle of four-phases:

1) The *await* phase is entered by a *SyncGuard* g_x upon reset, node rejuvenation or after completing a previous *SyncGuard* cycle. Upon rejuvenation or reset, g_x is put into the *await-first re-synchronization (AFR)* state, where it sets a timeout that equals the worst-case node rejuvenation time $T_{WCRejuv}$, and waits for the first synchronization message from n_x. Failing to present g_x with the synchronization message before the timer expiry is considered a re-synchronization error and results in g_x reactively rejuvenating n_x by transitioning its state machine into the *node rejuvenation trigger* (NRT) state. An endless succession of such failures is perceived by other nodes as a permanent node failure. On the other hand, when g_x enters the *await* phase after the completion of the previous cycle, its state machine transitions into the *await subsequent re-synchronization (ASR)*, where it waits for a synchronization message from n_x for a smaller duration of 2π (because n_x is already synchronized and integrated with the other nodes). Again, if n_x fails to present g_x with the synchronization message before timer expiry, it gets reactively rejuvenated.
2) The *entry-selection* phase is entered from the *AFR* or *ASR* states upon the valid reception of n_x's synchronization message, triggering the start of a new *SyncGuard* cycle and transitioning g_x's state machine into the *schedule entry selection (SES)* state, During this phase, g_x uses the relayed synchronization message from its node to select a valid entry in its communication and rejuvenation schedule (i.e., updates its view on system state). If the conveyed message content matches g_x's knowledge of the current system state (e.g., schedule entry is immediate successor of previous), g_x creates a message holding its system view and authenticates it using the cryptographic unit, which n_x then relays to other nodes. Otherwise, g_x reactively rejuvenates n_x by transitioning to state *NRT*.
3) The *entry-execution* phase is entered after the expiry of the *entry-selection* phase duration, where g_x transitions into the *network access (NCA)* state. During this phase, the guardian g_x grants its node n_x access to the network channels, by enabling transmission control for the duration of n_x's allotted TDMA slots on the corresponding

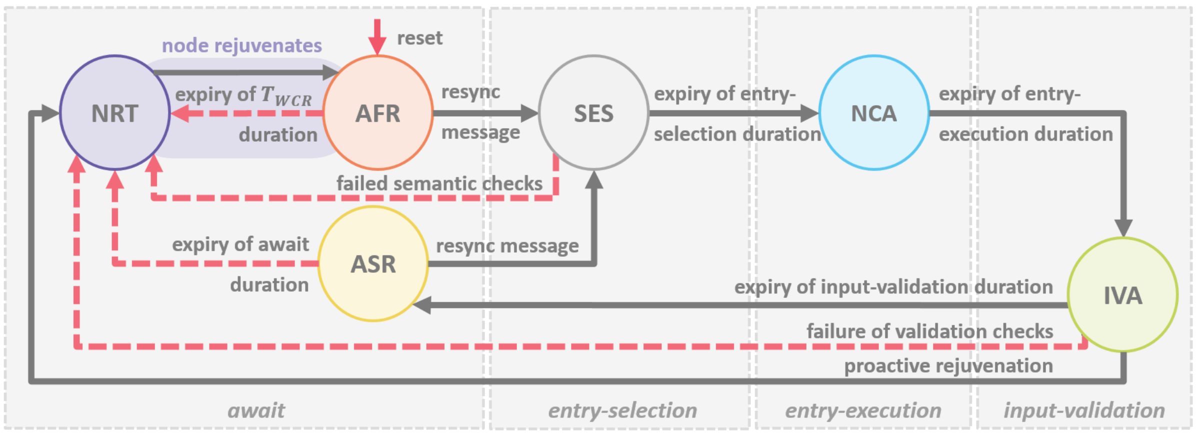


Fig. 5: *SyncGuard* abstract network access and rejuvenation state machine.
Solid black lines indicate normal progression through *SyncGuard* cycle phases.
Dashed red lines denote breaking out of the *SyncGuard* cycle.

communication channel. This is the only phase where g_x grants its node write access to the network channels.

4) The *input-validation* phase is entered upon the timeout of n_x's TDMA slot end, moving g_x's state machine into the *input validation (IVA)* state. Under our fault model (Section V), nodes may behave maliciously only after T^{Ind}_{Compr} relative to their rejuvenation time, which means that before T^{Ind}_{Compr} is reached, a *SyncGuard* can trust the relayed temporal information from its node, given they pass the rudimentary semantic checks during the *entry-selection* phase. Note that T^{Ind}_{Compr} cannot be set to zero, since g_x would not be able to prevent malicious n_x from causing collisions (see Section IV-B). Practically, a re-initialized *SyncGuard* needs to trust its node for at least one *SyncGuard* cycle to join a synchronous cluster of nodes. Hence, during this short interval of trust, the behavior of g_x in the *IVA* state is restricted to passively receiving system view messages from remote *SyncGuard*s, for the duration of the *input-validation* phase. When T^{Ind}_{Compr} elapses, g_x also executes a deterministic input validation algorithm (see Section VI-C) in the *IVA* state. The algorithm counteracts timing attacks by using the received and authenticated system view messages from other *SyncGuard*s to evaluate the correctness of the relayed information from the guarded node. It causes the *SyncGuard* to reactively rejuvenate its node if any of its checks fails. At the end of the *input-validation* phase, g_x moves to the *NRT* state if the proactive rejuvenation of n_x is scheduled (beginning n_x proactive rejuvenation slot). Otherwise, it transitions into the *ASR* state. Note that the length of this phase is dependent on the maximum number of messages, which must be received in any *SyncGuard* cycle, as well as the capabilities of the cryptographic unit.

The length of the *SyncGuard* cycle $T^{SyncGuard}_{Cycle}$ and thereby the frequency of exchanging *SyncGuard* messages can be determined based on the gap intervals between TDMA slots. When slots in the TDMA schedule are separated by a minimum gap of 2π, following the guardian design rules for node transmission windows [37], the local clocks of *SyncGuard*s and nodes need to be synchronized at the same rate to remain within the same precision. It is also possible to stretch the *SyncGuard* cycle, thus worsening clock precision for *SyncGuard*s compared to the clock precision between nodes to reduce the frequency of re-synchronizing *SyncGuard*s and exchanging their messages. However, the system designer must choose gaps between TDMA slots based on the *SyncGuard* clock precision instead of that between nodes to rule out collisions. Note that reducing the frequency of exchanging *SyncGuard* messages, also affects the freshness of *SyncGuard*s knowledge about the operational state of their peers. Therefore, it is up to the system designer to weigh up these trade-offs for the system he/she constructs, while ensuring that $T^{SyncGuard}_{Cycle}$ suffices for reception and authentication of all scheduled messages in the cycle. Note that apart from the bandwidth required to transmit and relay feedback messages as well as the inserted gaps between TDMA slots, the *SyncGuard* cycle has no additional influence on the effective network bandwidth since the cycle phases do not overlap at different *SyncGuard*s.

B. *SyncGuard Messages and Interfaces*

*SyncGuard*s implement two types of messages:

a) feedback messages, which represent the sending *SyncGuard*'s view on the system state via the following fields:

ID	***position***			**membership**	***HMAC***
	current mode	*requested mode*	*cycle ID*		

ID: uniquely identifies the sending *SyncGuard* among the communicating *SyncGuard* peers; ***position:*** indicates the position within the TDMA schedule as *current mode* — index of the current TDMA schedule — *requested mode* — index of the TDMA schedule to switch to after the current one completes — *cycle ID* — a counter, large enough to avoid replay attacks, that denotes the sending *SyncGuard*'s cycle (as counter modulo TDMA schedule length) as well as current proactive rejuvenation slot (counter modulo length of rejuvenation period); ***membership:*** is a bit vector denoting for each node its operational status; ***HMAC:*** contains the keyed hash of the message, created with the symmetric key shared by all *SyncGuard*s; Only *SyncGuard*s have access to this key.

b) synchronization messages, relayed from a node to its *SyncGuard*. They only hold the position field of feedback messages. Additionally, their arrival time at the receiving *SyncGuard* is used to correct its clock.

We allow *SyncGuard*s to change the mode of operation, including the TDMA schedule. However, we restrict the time when such a change can happen to the last schedule entry (i.e, last *SyncGuard*'s cycle) of a given operational mode to ensure that first the request is stable and cannot be changed anymore and that all *SyncGuard*s receive the same request from their nodes and agree on it. Nodes that requests to change the mode of operation earlier are considered faulty in their attempt to create a mode synchronization error and are reactively rejuvenated by their guardians.

With respect to communication interfaces, *SyncGuard* has two interfaces: a) The node guardian interface NGI is a dual port memory with bidirectional data flow for the relay of messages between *SyncGuard*s through their nodes. The NGI is structured into two sections: the first is reserved for the *SyncGuard*'s own feedback message and the second for relaying the node synchronization message as well as feedback messages from remote *SyncGuard*s. All *SyncGuard*s enforce windows for reading/writing messages from/to the NGI during each cycle. For example, the synchronization message from the node must be received within the *await* phase and the node must read *SyncGuard*'s feedback message during the *entry-execution* phase. *SyncGuard* trashes these messages by the end of the cycle. Note that *SyncGuard*s receive relayed feedback messages for a given cycle from the beginning of the *input-validation* phase of the previous cycle until the beginning of the one in the current cycle. b) The channel state interface CSI is a rudimentary interface that reflects *SyncGuard* control over the communication channels. The transmission controller TC reads changes in the CSI and enforces the node transmission path to the network channels accordingly. The CSI has for each channel an associated signal, which, when set, causes TC to grant the node write access to that channel.

In addition to the needed memory for storing the dynamic data structures represented by the above-mentioned messages, each *SyncGuard* must be equipped with enough memory to store *SyncGuard* configurations, which are needed for its operation. These include the symmetric cryptography key and the network access and rejuvenation schedules.

C. The Input Validation Algorithm

The purpose of the input validation algorithm is to reach a consensual view about the system state across the communicating *SyncGuard*s despite the presence of participating faulty nodes, which is necessary to ensure the correct delivery of their provided services. Algorithm 1 illustrates the pseudocode for the input validation algorithm (implemented by procedure *ValidateIn()*). It also lists procedure $selectEntry()$ (lines 1-5) to indicate the phase, where *SyncGuard*s update their view and create their feedback messages. Note that *SyncGuard*s store feedback messages according to their IDs, so duplicates or messages with invalid IDs are discarded. Furthermore, *SyncGuard*s use their cryptographic unit to verify the authenticity and integrity of the received feedback messages based on the appended HMACs, which works by recalculating the HMAC using the message content and the secret *SyncGuard* key.

Each *SyncGuard* keeps a counter called $nack$, which counts the number of *SyncGuard*s that are in disagreement with its system view. It also maintains the *expected* set, which is a configuration parameter holding, for each *SyncGuard* cycle, the IDs of *SyncGuard*s, from which it shall receive feedback messages.

Property 1. *The input validation algorithm causes a SyncGuard g_x to reactively rejuvenate its node n_x by the end of the validation phase if $O + R + A > F$ holds, where: O, R, A is the number of omitted, replayed feedback messages and respectively the number of out-of-slot receivers during n_x's last transmission.*

By the end of the *input-validation* phase, the $check4Omissions()$ procedure at g_x must have updated the value of the $nack$ counter with the difference between received and expected messages in the $expected$ set, which equals O (line 18). Further, expected messages that are manipulated in the value domain fail authentication and get dropped by g_x. Therefore, they are considered omitted and count towards O. Afterwards, procedure $check4Replays()$ compares the position field in the received authentic feedback messages with that in n_x's synchronization message (line 25). Therefore, g_x must have incremented the $nack$ counter R times for R replayed authentic feedback messages by n_x.

Finally, If n_x managed to desynchronize g_x from A other guardians, thereby transmitting such that it causes an out-of-slot reception at A nodes in the preceding *SyncGuard* cycle, or indiscernibly if A other remote nodes accuse n_x's transmission of missing their acceptance window (refer to Section IV-C). g_x reacts by executing the $check4TimelyAccess()$ procedure, which increments the $nack$ counter and resets the bit position in its membership field for every one of the A remote *SyncGuard* that disacknowledges n_x's transmission by equally resetting g_x's bit in its expected feedback message (lines 34-35). Note that this action is not performed if the remote *SyncGuard* sends a feedback message with an empty membership field (line 33), which happens during the first network access of the remote *SyncGuard*'s node after rejuvenation.

The algorithm then executes the $rejuvenateIfFaulty()$ procedure, which causes the *SyncGuard* to reactively rejuvenate its node if the value of the $nack$ counter is larger than the tolerable threshold F (line 39). In other words, n_x may only proceed unrejuvenated if by the end of the validation phase, the check $nack = O + R + A > F$ fails.

Property 2. *The input validation algorithm causes all communicating SyncGuards to reactively rejuvenate their nodes by*

Algorithm 1 Input validation algorithm

```
    // Parameters and variables for a SyncGuard $g_x$    ▷ set upon integration
    $expected$        ▷ a set holding SyncGuard IDs, from which $g_x$ expects feedback
    $feedback$        ▷ a set of messages, carrying the local system views of SyncGuards
    $myFeedback$      ▷ $g_x$'s feedback message to other SyncGuards
    $nack$            ▷ variable holding the number of disagreeing feedback messages
    $syncMsg$         ▷ synchronization message from $n_x$ in the current SyncGuard cycle

 1: procedure $selectEntry()$    ▷ triggered at the beginning of entry-selection phase
 2:     $checkSemantics(syncMsg)$    ▷ e.g., whether schedule entry exists and succeeds the previous
 3:     $myFeedback.position = syncMsg$
 4:     $createFeedbackMsg(myFeedback)$    ▷ forms and places $g_x$'s authenticated feedback message to be relayed by $n_x$
 5: end procedure

 6: procedure $validateIn()$    ▷ triggered at the beginning of input-validation phase
 7:     if($n_x$ is potentially compromised, i.e.,
            $globalTime \geq lastRejuvTime + T_{Compr}^{Ind}$)
        then
 8:         $check4Omissions()$
 9:         $check4Replays()$
10:         $check4TimelyAccess()$
11:         $recoverIfFaulty()$
12:     end if
13:     $feedback.clear()$    ▷ removes all entries from the feedback set
14:     $update(parameters)$    ▷ e.g., expected feedback messages and position from next entry in $g_x$'s schedule
15:     $checkProactiveRejuvenate()$    ▷ $g_x$ proactively rejuvenates $n_x$ according to Algorithm 2
16: end procedure

17: procedure $check4Omissions()$    ▷ updates nack with the number of omitted feedback messages
18:     if $feedback.size() < expected.size()$ then
19:         $nack = expected.size() - feedback().size()$
20:     else
21:         $nack = 0$
22:     end if
23: end procedure

24: procedure $check4Replays()$    ▷ updates nack with the number of replayed feedback messages
25:     for ($j = 0; j = feedback.size() - 1; j++$) do
26:         if ($feedback[j].position \neq myFeedback.position$) then
27:             $nack+ = 1$
28:         end if
29:     end for
30: end procedure

31: procedure $check4TimelyAccess()$    ▷ updates nack with the number of out-of-slot receivers
32:     for ($j = 0; j = feedback.size() - 1; j++$) do
33:         if ($!feedback[j].membership.empty()$ $and$
                $feedback[j].membership[i] == 0$ $and$
                $feedback[j].position == myFeedback.position$)
            then
34:             $nack+ = 1$
35:             $myFeedback.membership[j] = 0$
36:         end if
37:     end for
38: end procedure

39: procedure $recoverIfFaulty()$
40:     if $nack > F$ then
41:         $reactiveRejuvenate()$
42:     end if
43: end procedure
```

the end of the validation phase, if more than F faults occur in any SyncGuard cycle.

A situation where more than F faults occur means that the expression $nack > F$ (line 39) evaluates to true at all *SyncGuard*s, thus causing them, following Proposition 1, to reactively rejuvenate together by the end of their *input-validation* phase.

Note that the behavior of the algorithm is acceptable and complies with our fault assumptions in Section V. Furthermore, when these assumptions are violated, it restores the system to a safe and stable state as a never-give-up strategy.

D. SyncGuard Proactive Rejuvenation Scheme

Since the strategy of coordinating rejuvenation is not the main goal of this paper, we show a simple proactive rejuvenation scheme in Algorithm 2, which has a similar structure to the example in Figure 2, and emphasize the possibility to select different rejuvenation coordination solutions.

Algorithm 2 simple proactive rejuvenation algorithm

```
    // Parameters and variables for a SyncGuard $g_x$    ▷ set upon initialization and integration
    $PRS_i$         ▷ proactive rejuvenation slot to which $g_x$ is assigned
    $T_{RS}$        ▷ length of rejvuenation slot in terms of SyncGuard cycles
    $G$             ▷ number of proactive rejuvenation slots in a rejuvenation period
    $cycle\_ID$     ▷ counter variable holding current SyncGuard cycle number

1: procedure $checkProactiveRejuvenate()$
2:     if ($cycle\_ID \mod T_{RS} == 0$) then    ▷ beggining of a $PRS$
3:         if (($\frac{cycle\_ID}{T_{RS}}$) $\mod G == PRS_i$) then    ▷ $PRS_{current}$=$PRS_i$
4:             $rejuvenateNode()$    ▷ proactively trigger rejuvenation of node $n_x$
5:         end if
6:     end if
7: end procedure
```

All *SyncGuard*s maintain a copy of the parameters needed for proactive rejuvenation such as assigned proactive rejuvenation slot PRS and its size T_{RS} and number of proactive rejuvenation slots G within the rejuvenation period T_{PRejuv}. Rejuvenation slots are measured as multiple of the *SyncGuard* cycle. *SyncGuard*s use the variable $cycle\_ID$ to infer their proactive rejuvenation turn (lines 2-3). As mentioned in Section II-C, system availability and fault masking are guaranteed as long as no more than F faults occur every T_{PRejuv}.

E. SyncGuard Overhead

The simplicity of the input validation algorithm in Algorithm 1 confirms the possibility of turning it into a hardware-near implementation (on a simple microcontroller, in an FPGA or as custom logic). This leaves message authentication and in particular HMAC computation as the dominant factor of our solution. Note that when network agnostic guardians must communicate, such an overhead becomes inevitable because faulty nodes can intercept and manipulate any unauthenticated message, and may even impersonate other *SyncGuard*s. On the other hand, safety-critical designs with a security focus should already have message authentication in place.

Assuming message frames with 240 bytes reserved for application data (e.g., as in TTP and FlexRay), *SyncGuard* feedback messages consume around 13% of the available bandwidth for application data when communicating the allocation ID (5 bits), position (42 bits), membership information (32 bits) and an HMAC (with recommended size of 160 bits [38]) in every TDMA slot. In comparison, time-triggered protocols for Ethernet-based networks, e.g., TTEthernet and TSN, have an abundance of data bandwidth (up to 1500 bytes per frame), which reduces *SyncGuard*s message overhead to a mere 2% when using the same allocation in TT traffic.

Software-based HMAC generation and validation runtime costs can be significant and may affect synchronization accuracy, in particular if the runtime to generate HMACs vary significantly. We measured HMAC generation and validation on a 1.4 GHz Raspberry Pi 3B+ to take 6310 and 6477 cycles respectively (i.e., 4.5 us and 4.6 us). Values in that order of magnitude have also been confirmed for software implementations on deeply embedded systems [39]. Fortunately, optimized hardware implementations [40] achieve a throughput rate of 655.66 Mbps for masked and 875.22 Mbps for unmasked HMAC (i.e., 4 respectively 3 cycles for 240 byte messages). The processing of such hardware-accelerated solutions is negligible but requires FPGAs or custom hardware.

VII. *SyncGuardLite*

The systems we considered previously have, due to their complexity, node rejuvenation times that are very long when compared to the length of the *SyncGuard* cycle $T_{Cycle}^{SyncGuard}$. With the goal of improving the applicability of *SyncGuard* to less complex and more resource constrained systems, we present *SyncGuardLite*, a guardian solution for systems with nodes characterized by relatively short rejuvenation times.

A. Communication-Free Re-synchronization

The basic idea of *SyncGuardLite* is to drop the requirement to exchange messages and to execute the input validation algorithm, while still guaranteeing collision-free network access. Communication-free re-synchronization allows savings on two fronts: it eliminates the consumed network data bandwidth, which was previously occupied by feedback messages and further reduces the guardian implementation complexity since cryptographic units are no longer necessary.

Furthermore, since *SyncGuardLite*s do not exchange messages, they trade off coordinating reactive rejuvenation and mode change support for a simpler implementation. While the former is ameliorated with more frequent proactive rejuvenation, the latter can be handled as in previous work, that is, by making the guardian mode change insensitive [41], [42]

Ruling out collisions in the presence of timing attacks, in particular acceptance window ones, requires widening the gaps between TDMA slots such that they not only consider the clock precision π but also the number of *SyncGuardLite* cycles during which the node is potentially compromised before being proactively rejuvenated in the next proactive rejuvenation period T_{PRejuv}. As discussed in Section IV-B, a malicious node can desynchronize its network-agnostic guardian each re-synchronization cycle T_{Resync} by the maximum correction term of the clock synchronization algorithm κ. Equation 3 shows how to calculate the minimum gap between TDMA slots such that collisions due to timing attacks are ruled out. Notice that the gap can be minimized by decreasing the proactive rejuvenation period T_{PRejuv} or by increasing T_{Compr}^{Ind} through fortifying nodes against adversaries.

$$slot\ gap = \pi + \max\{\pi, \left\lceil \frac{T_{PRejuv} - T_{Compr}^{Ind}}{T_{Resync}} \right\rceil \times \kappa\} \quad (3)$$

B. Replica Slot Multiplexing

In comparison to *SyncGuard*, more bandwidth savings can be made by exploiting the static behavior accompanied by using proactive node rejuvenation only. For instance, it is possible to multiplex the TDMA slots for node replicas offline with effects reflected in the *SyncGuardLite* schedule. In other words, a node is allocated no TDMA slot on any communication channel during its scheduled proactive rejuvenation. Such optimization is not possible for *SyncGuard*s due to the dynamic behavior when coordinating both reactive and proactive rejuvenation of nodes, which would require dynamically changing the network access schedule.

VIII. Conclusion

In this work, we proposed *SyncGuard*, a network-agnostic guardian, which addresses the shortcomings of previous guardian solutions of being network-protocol and topology dependent and of lacking the necessary means to protect against the exhaustion of system nodes over its long mission times. We discussed timing attacks targeting the safe synchronization of network-agnostic guardians and showed how to counteract them by designing *SyncGuard*s to not only enforce conformance to the communication schedule but also to trigger and coordinate the rejuvenation of nodes. *SyncGuard*-equipped systems trade off costly and complex hardware implementations with a relatively small reduction of data bandwidth on modern networks, which is necessary to allow *SyncGuard*s to safely synchronize, coordinate rejuvenation and support operational mode changes.

Our results of indirect, but safe communication via already existing infrastructure can be reused to implement simple yet resilient distributed functionality such as reconfiguration and activity control using network-agnostic components, which support scalability, certifiability and reusability. Moreover, the technology and network independence of such components shall enable a variety of certifiable implementations that mainly differ in their performance and way of coordinating node rejuvenation.

We finally introduced *SyncGuardLite*, a simplified network-agnostic guardian solution for systems with less complex nodes that rejuvenate relatively fast, thereby enabling *SyncGuardLite* to replace message exchange with larger gaps between TDMA slots and fast periodic node rejuvenation. However, since *SyncGuardLite*s do not communicate, they lose support for mode change and reactive rejuvenation.

References

[1] R. M. Lee, M. J. Assante, and T. Conway, "Analysis of the cyber attack on the ukrainian power grid," March 2016. [Online]. Available: "https://ics.sans.org/media/E-ISAC_SANS_Ukraine_DUC_5.pdf"

[2] J. Slay and M. Miller, "Lessons learned from the maroochy water breach," in *Critical Infrastructure Protection*, E. Goetz and S. Shenoi, Eds. Boston, MA: Springer US, 2008, pp. 73–82.

[3] D. Shepard, J. Bhatti, and T. Humphreys, "Drone hack: Spoofing attack demonstration on a civilian unmanned aerial vehicle," *GPS World*, vol. 23, pp. 30–33, 08 2012.

[4] A. Greenberg, "Hackers remotely kill a jeep on the highway—with me in it," wired, 2015, last accessed: 12-12-2022. [Online]. Available: "https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/"

[5] R. M. Lee, M. J. Assante, and T. Conway, "German steel mill cyber attack," *Industrial Control Systems*, vol. 30, no. 62, pp. 1–15, 2014.

[6] K. Zetter, "A cyberattack has caused confirmed physical damage for the second time ever," wired, 2015, last accessed: 12-12-2022. [Online]. Available: "https://www.wired.com/2015/01/german-steel-mill-hack-destruction/"

[7] T. M. Chen and S. Abu-Nimeh, "Lessons from stuxnet," *Computer*, vol. 44, no. 4, pp. 91–93, 2011.

[8] D. Goldstein, "Mouse click could plunge city into darkness, experts say," CNN, 2018, last accessed: 12-12-2022. [Online]. Available: "https://goldsteinreport.com/mouse-click-could-plunge-city-into-darkness-experts-say/"

[9] S. Scoles, "The feds want these teams to hack a satellite—from home," wired, 2020, last accessed: 12-12-2022. [Online]. Available: "https://www.wired.com/story/the-feds-want-these-teams-to-hack-a-satellite-from-home/"

[10] H. Kopetz, "The time-triggered model of computation," in *19th IEEE Real-Time Systems Symposium*, 1998, pp. 168–177.

[11] H. Kopetz and G. Bauer, "The time-triggered architecture," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 112–126, 2003.

[12] TTTech, *TTP Communication Protocol Specification*, FEB 2011.

[13] M. Rausch, *FlexRay: Grundlagen, Funktionsweise, Anwendung*, ser. Hanser eLibrary. Hanser, 2008. [Online]. Available: https://books.google.de/books?id=OES7wAEACAAJ

[14] TTTech, *Time-Triggered Ethernet Communication Protocol Specification*, NOV 2016.

[15] C. Temple, "Avoiding the babbling-idiot failure in a time-triggered communication system," in *Digest of Papers. Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing (Cat. No.98CB36224)*, 1998, pp. 218–227.

[16] G. Bauer, H. Kopetz, and W. Steiner, "The central guardian approach to enforce fault isolation in the time-triggered architecture," in *The Sixth International Symposium on Autonomous Decentralized Systems, 2003. ISADS 2003.*, 2003, pp. 37–44.

[17] G.-N. Sung, C.-Y. Juan, and C.-C. Wang, "Bus guardian design for automobile networking ecu nodes compliant with flexray standards," 05 2008, pp. 1–4.

[18] W. Steiner, "Ttethernet: Time-triggered services for ethernet networks," in *2009 IEEE/AIAA 28th Digital Avionics Systems Conference*, 2009, pp. 1.B.4–1–1.B.4–1.

[19] L. Lamport and P. M. Melliar-Smith, "Synchronizing clocks in the presence of faults," *J. ACM*, vol. 32, no. 1, p. 52–78, jan 1985. [Online]. Available: https://doi.org/10.1145/2455.2457

[20] P. Ramanathan, K. Shin, and R. Butler, "Fault-tolerant clock synchronization in distributed systems," *Computer*, vol. 23, no. 10, pp. 33–42, 1990.

[21] C. Fetzer and F. Cristian, "An optimal internal clock synchronization algorithm," in *COMPASS '95 Proceedings of the Tenth Annual Conference on Computer Assurance Systems Integrity, Software Safety and Process Security'*, 1995, pp. 187–196.

[22] J. Lundelius and N. Lynch, "An upper and lower bound for clock synchronization," *Information and Control*, vol. 62, no. 2, pp. 190–204. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0019995884800339

[23] H. Kopetz, "Sparse time versus dense time in distributed real-time systems," in *[1992] Proceedings of the 12th International Conference on Distributed Computing Systems*, 1992, pp. 460–467.

[24] J. Lisner, "Scheduling in a time-triggered protocol with dynamic arbitration," in *Proceedings of the IEEE International Symposium on Industrial Electronics, 2005. ISIE 2005.*, vol. 4, 2005, pp. 1399–1404.

[25] K. Driscoll, B. Hall, and S. Varadarajan, "Maximizing fault tolerance in a low-s wap data network," in *2012 IEEE/AIAA 31st Digital Avionics Systems Conference (DASC)*, 2012, pp. 7A2–1–7A2–16.

[26] P. Sousa, N. F. Neves, and P. Verissimo, "Proactive resilience through architectural hybridization," in *Proceedings of the 2006 ACM symposium on Applied computing*. ACM, 2006, pp. 686–690.

[27] P. Sousa, A. N. Bessani, M. Correia, N. F. Neves, and P. Verissimo, "Highly available intrusion-tolerant services with proactive-reactive recovery," *IEEE Trans. Parallel Distrib. Syst.*, vol. 21, pp. 452–465, 2010.

[28] M. I. Alkoudsi, G. Fohler, and M. Völp, "Tolerating resource exhaustion attacks in the time-triggered architecture," in *2022 XII Brazilian Symposium on Computing Systems Engineering (SBESC)*, 2022, pp. 1–8.

[29] H. Fan, F. Zhu, C. Liu, L. Zhang, L. Zhuang, D. Li, W. Zhu, J. Hu, H. Li, and Q. Kong, "Baidu apollo em motion planner," 2018. [Online]. Available: https://arxiv.org/abs/1807.08048

[30] P. Veríssimo, L. Rodrigues, and M. Baptista, "Amp: A highly parallel atomic multicast protocol," *SIGCOMM Comput. Commun. Rev.*, vol. 19, no. 4, p. 83–93, Aug. 1989. [Online]. Available: https://doi.org/10.1145/75247.75256

[31] D. Imbs and M. Raynal, "Simple and efficient reliable broadcast in the presence of byzantine processes," *CoRR*, vol. abs/1510.06882, 2015. [Online]. Available: http://arxiv.org/abs/1510.06882

[32] M. Garcia, A. Bessani, I. Gashi, N. Neves, and R. Obelheiro, "Analysis of operating system diversity for intrusion tolerance," *Softw. Pract. Exper.*, vol. 44, no. 6, p. 735–770, jun 2014. [Online]. Available: https://doi.org/10.1002/spe.2180

[33] R. Pucella and F. B. Schneider, "Independence from obfuscation: A semantic framework for diversity." in *19th IEEE Work. on Computer Security Foundations*, 2006, pp. 230–241.

[34] T. Roeder and F. Schneider, "Proactive obfuscation," *ACM Trans. Comput. Syst.*, vol. 28, 07 2010.

[35] Jun Xu, Z. Kalbarczyk, and R. K. Iyer, "Transparent runtime randomization for security," in *22nd International Symposium on Reliable Distributed Systems, 2003. Proceedings.*, 2003, pp. 260–269.

[36] P. Verissimo, A. Casimiro, and C. Fetzer, "The timely computing base: Timely actions in the presence of uncertain timeliness," in *Proceeding International Conference on Dependable Systems and Networks. DSN 2000*, 2000, pp. 533–542.

[37] J. Rushby, "Formal verification of transmission window timing for the time-triggered architecture," 01 2001.

[38] J. R. Vacca, Ed., *Computer and Information Security Handbook*, 3rd ed. USA: Morgan Kaufmann, 2017.

[39] H. Bühler, A. Walz, and A. Sikora, "Benchmarking of symmetric cryptographic algorithms on a deeply embedded system," *IFAC-PapersOnLine*, vol. 55, no. 4, pp. 266–271, 2022, 17th IFAC Conference on Programmable Devices and Embedded Systems PDES 2022 — Sarajevo, Bosnia and Herzegovina, 17-19 May 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2405896322003597

[40] Z. He, L. Wu, and X. Zhang, "High-speed pipeline design for hmac of sha-256 with masking scheme," in *2018 12th IEEE International Conference on Anti-counterfeiting, Security, and Identification (ASID)*, 2018, pp. 174–178.

[41] H. Kopetz, R. Nossal, R. Hexel, A. Krüger, D. Millinger, R. Pallierer, C. Temple, and M. Krug, "Mode handling in the time-triggered architecture," in *IFAC Workshop on Distributed Computer Control Systems (DCCS97)*, Seoul, Korea, July 1997.

[42] F. Heilmann, A. Syed, and G. Fohler, "Mode-changes in cots time-triggered network hardware without online reconfiguration," *SIGBED Rev.*, vol. 13, no. 4, p. 55–60, nov 2016. [Online]. Available: https://doi.org/10.1145/3015037.3015046