# Resilient and Secure Programmable System-on-Chip Accelerator Offload

Inês Pinto Gouveia, Ahmad T. Sheikh, Ali Shoker, Suhaib A. Fahmy and Paulo Esteves-Verissimo
CEMSE Division, King Abdullah University of Science and Technology (KAUST)
Thuwal 23955-6900, Kingdom of Saudi Arabia
{ines.pintogouveia, ahmad.sheikh, ali.shoker, suhaib.fahmy, paulo.verissimo{@kaust.edu.sa}}

***Abstract*—Computational offload to hardware accelerators is gaining traction due to increasing computational demands and efficiency challenges. Programmable hardware, like FPGAs, offers a promising platform in rapidly evolving application areas, with the benefits of hardware acceleration and software programmability. Unfortunately, such systems composed of multiple hardware components must consider integrity in the case of malicious components. In this work, we propose Samsara, the first secure and resilient platform that derives, from Byzantine Fault Tolerance (BFT), protocols to enhance the computing resilience of programmable hardware. Samsara uses a novel lightweight hardware-based BFT protocol for Systems-on-Chip, called H-Quorum, that implements the theoretical-minimum latency between applications and replicated compute nodes. To withstand malicious behaviors, Samsara supports hardware rejuvenation, which is used to replace, relocate, or diversify faulty compute nodes. Samsara's architecture ensures the security of the entire workflow while keeping the latency overhead, of both computation and rejuvenation, close to the non-replicated counterpart.**

***Index Terms*—MPSoC, Fault and Intrusion Tolerance (FIT), Rejuvenation, Reconfigurable Hardware, Resilience**

## I. Introduction

The trend of offloading computing to hardware accelerators is gaining more traction in emerging systems and applications such as cyber-physical systems (CPS), Internet-of-Things (IoT) services, automation and space applications [1]–[3]. The purpose is often both seeking accelerated computing and enhanced security. Hardware acceleration is a result of building application-specific logic, e.g., matrix multiplication and cryptographic operations, precluding OS or software interruptions or locking, and taking advantage of fine-grained parallelism; whereas security leverages the hardware immutability properties to provide guarantees like tamper-resistant data processing, isolation and building security abstractions [4]–[10]. To continuously support new use-cases while reducing the burden of hardware fabrication of a monolithic system (costly and slow), these systems are often (1) assembled and deployed as *Multi-Processor System-on-Chip* (MPSoCs) that makes use of modular, cheap *commercial-of-the-shelf* (COTS) components [11], [12]; and (2) leverage the hardware programmablity features of general-purpose reconfigurable hardware like FPGAs and CGRAs [13]–[15]. Unfortunately, this raises new computing integrity challenges against intrusions and faults.

Reconfigurable hardware fabrics allow new computing functionality to be loaded after fabrication [16] by means of a binary file (usually called the *bitstream*) that maps an architecture or module (which we call a *Tile* henceforth) onto the fabric. While this gains the intrinsic performance and security properties of hardware after loading, it inherits some of the mutability weaknesses of software prior to loading (given the nature of the binary file), which can be prone to intrusion and benign faults [3], [17]–[20]. We noticed that little has been done to ensure computing integrity when a tile is faulty or malicious (*Byzantine* [21]). Indeed, literature has addressed this issue through several complementary approaches, e.g., focusing on encryption, isolation, and memory obfuscation [22]–[24]; however, these approaches assume tiles are trusted or rely on software and, thus, mutable implementations. For instance, encrypting the bitstream [22] prevents tampering with it at rest, e.g., in memory; however, it neither withstands a contaminated bitstream at the development phase (which can occur due to the lengthy hardware development process and the use of many tools), nor *Network-on-Chip* (NoC) attacks under execution [25]. On the other hand, containment and memory randomization [23] ensure the execution of deployed co-existing tiles do not interfere, but do not give any guarantees on the output integrity if tiles are Byzantine, e.g., corresponding to a vulnerable bitstream. Finally, *Triple-Modular-Replication* (TMR) is often used as a voting mechanism to ensure computing integrity [26], but it is not adequate for malicious adversaries.

In this paper, we introduce Samsara, the first resilient and secure computing platform architecture for programmable MPSoCs. Samsara hinges on hardware reconfigurability to ensure computing integrity by employing state-machine replication (SMR) [27] and rejuvenation of tiles, managed through a novel lightweight hardware-based Byzantine agreement protocol, called *H-Quorum*. *H-Quorum* is optimized for simplicity and low latency to make it feasible for hardware offload. When *H-Quorum* detects a fault or delay, it recovers via a rejuvenation process that is seamless to the application. Rejuvenation is used to reload (from a bitstream library) a tile facing transient faults or glitches, replace a malicious or faulty tile with a diversified version [28] to improve independence of failures, or relocate the tile to another location to avoid underlying glitches or compromised network routes. Diversity of bitstreams is possible by means of different logic module configurations at design time, through design with hardware description languages (HDLs) or open-source architectures and frame-

works like those of RISC-V [29], [30]. Samsara's architecture is designed to maintain the security of the entire workflow including storing encrypted bitstreams, booting, execution, and rejuvenation.

One of the main challenges addressed by Samsara is circumventing the known complexity and delay of Byzantine agreement, which would represent a significant overhead for hardware accelerators. Hence, we optimize *H-Quorum* for simplicity and latency by inspiring from Quorum [31], which is known to be the simplest classical Byzantine agreement protocol with the lowest theoretical latency possible [31], [32]. Nevertheless, Quorum is known to be impractical in Internet-based settings, as it assumes that clients are not Byzantine and it recovers into a heavy-weight backup phase under faults (see details in Section III-A). We circumvent these limitations by making smart choices: Samsara employs an *Application Specific Integrated Circuit* (ASIC) *Controller* that plays the role of a trusted "semi-leader" replica: it acts as a leader by mediating the application requests sent to the replicated tile and collecting votes, while not participating in the computation phase. This is key to ensure that the Controller exhibits a small fingerprint (a few hundred LoC — of a hardware description language — in our case) to be easily verified, and, thus, considering it a trusted-trustworthy hardware component is a valid assumption.

Another main challenge in Samsara is to ensure secure rejuvenation. Rejuvenation is done while the system is running by reloading a new bitstream to the reconfigurable fabric via booting scripts. To make sure intruders cannot modify, initiate, or interrupt rejuvenation, our MPSoC architecture makes use of a simple microprocessor in which only the Controller can run these scripts. This makes sure that *H-Quorum* maintains the state integrity across rejuvenation, including a fast shared-memory state-transfer mechanism, detailed in Section III-B. We provide a systematic proof sketch in Appendix A to ensure the correctness of the entire workflow phases.

As a proof of concept, we implemented Samsara on a Xilinx ZCU102 FPGA SoC [33]. Our evaluation of accelerator applications shows that Samsara's latency is slightly higher than a non-replicated accelerator, but up to 35.9% lower than state-of-the-art hardware-based Byzantine agreement counterparts, like *iBFT* [34] and a shared-memory implementation of *MinBFT* [10]. Additionally, the evaluation shows that the rejuvenation time is negligible, and 99.89% faster than rebooting the whole platform.

The rest of the paper is organized as follows: Section II discusses Samsara's system and threat models, as well as background on MPSoCs and reconfiguration. Samsara's architecture and *H-Quorum* are presented in Section III. We then provide the evaluation of our proof of concept on an ZCU102 FPGA in Section IV, and discuss related work in Section V. Finally, we conclude in Section VI. We provide correctness and liveness proof sketches in Appendix A, for the reader's convenience.

## II. System and Threat Models

### A. System Model

*1) Generic Programmable MPSoC:* We consider a generic programmable *Multi-Processor System on Chip* (MPSoC) composed of: (1) a processor portion, called the *Processing System* (PS) that has one or more processing cores, used to run application software, alongside the basic I/O and peripherals; (2) a reconfigurable portion (e.g., FPGA, CGRA), used to deploy offloaded functionality, i.e., accelerated tasks. Examples of commercial devices that fit this model include the AMD/Xilinx Zynq-7000 [35] and Zynq UltraScale+ [11], and the Intel/Altera Agilex 7 SoC. All portions of the MPSoC are connected via an underlying reliable hardware Network-on-Chip (NoC) or network bus, such as AMBA AXI or PCIe [1]. The choice of communication medium depends on the platform's fine-grained architecture and application requirements (e.g., bandwidth). Channels may slightly delay circulated messages or data but will eventually deliver them as long as they are not under attack.

*2) Samsara COTS:* In the context of Samsara, we propose an additional *application-specific integrated circuit* (ASIC) component, as part of the MPSoC PS, which implements the *Controller* of *Samsara* (as explained in Section III-A); and a simple COTS microprocessor, *MP-Boot*, to handle booting and configuration of the reconfigurable portion, (see Section III-A)). We propose that the MPSoC is supplied with volatile memory and hardware-assisted *Tamper-Resistant Storage* (TRS) (e.g., *TPM* [36] or *Hardware Credential Storage* [37]), used for tamper-proof storage and compute. Modern MPSoCs allow addition of specific COTS, modules, or *microchips* as we do [38]. Future MPSoC designs leveraging chiplets also provide this flexibility.

In addition to this, as detailed in Section III-A, the reconfigurable portion is divided into partitions, i.e., *tiles* [39], that encapsulate compute functions. A tile is an abstraction of the logic modules containing offloaded and/or accelerated tasks, placed on dynamically reconfigurable hardware regions. Samsara uses these tiles as *H-Quorum* replicas.

*3) Reconfiguration Implementation:* There are several possible implementations for the reconfigurable portion. For instance, *FPGA Programmable Logic* (PL) can be used to instantiate hardware accelerators in two phases: (1) loading *full bitstreams* that define the base infrastructure, e.g., determining the number of tiles and their locations, networking (called routing), and other components; and (2) *partial bitstreams* for reconfiguration at runtime, used to deploy new compute logic. Alternatively, CGRAs [40] consist of a large array of function units (FUs) interconnected by a mesh style network; they offer coarser-grained configuration than FPGAs. Yet another possibility is to use multiple connected GPUs or other diverse FUs, e.g., via a secure implementation of PCIe [41].

[1]The main difference between a NoC and a bus pertains to the fact that the NoC uses a conceptually point-to-point approach, while a bus tends to be multipoint.

For the remainder of this paper, we shall focus on an FPGA-based implementation of the reconfigurable portion. For this, we first present a brief overview of MPSoC and FPGAs to ease the understanding of the remaining sections.

### B. Background on MPSoC and FPGA

This section provides a brief background on programmable Multi-Processor Systems-on-Chip (MPSoC) and FPGA Partial Reconfiguration (PR).

*1) MPSoC Architecture:* An MPSoC is a System-on-a-Chip (SoC) which includes multiple processors, often used in embedded devices. A typical modern MPSoC [11] includes multiple fabricated processing cores (called *hard* cores) in a Processing Side (PS) and programmable hardware, based on Field Programmable Gate Array (FPGA) technology. The latter can be reconfigured with arbitrary custom-hardware logic after fabrication.

The cost of FPGA development and deployment is negligible for low to medium volumes. However, per-device cost is expensive. Application Specific Integrated Circuits (ASICs) are chips with immutable logic circuits. They are preferred for large scale manufacturing due to cheaper per-device cost after amortizing the high non-recurring engineering (NRE) costs.

*2) FPGAs:* FPGAs possess reconfigurable programmable logic (PL), consisting of fabric that can be programmed according to a design that is mapped into a configuration file called a *bitstream*. A common case is to design accelerators or softcores, and connect them to other modules such as memory controllers and I/O. The PL can be configured directly through an external interface or via the MPSoC's PS. The PL consists of a sea of building blocks such as Configurable Logic Blocks (CLB), programmable routing, I/O banks, Block RAMs (BRAMs), etc. Designed modules, e.g., accelerators or softcores, and their communication medium are mapped into the aforementioned components.

Traditionally, when the PL is programmed, it becomes immutable, i.e., the free and configured regions of the PL cannot be further changed without reconfiguring the whole fabric. However, dynamic partial reconfiguration (PR) allows partitions of the PL to be reconfigured at runtime [42], without mutating the rest of the design. Thus, a full bitstream configures the whole PL, while partial bitstreams modify only the specified partition(s) without compromising the integrity of applications running on the remaining portions of the PL. One of the biggest advantages of PR is the ability to time multiplex the underlying silicon for various tasks.

To design a system using PR is it necessary to *floorplan* the design: the PL fabric is spatially divided into *Reconfigurable Partitions (RPs)* that are to contain dynamically *Reconfigurable Modules* (RMs). When using PR, floorplans contain a static partition, containing the design modules that cannot or should not be dynamically reconfigured [2]; and one or more RPs. The number and placement of these RPs cannot be changed at runtime, meaning that, in order to change the number of RPs used and the type of RMs they can hold, a full bitstream (configured with a new floorplan) must be loaded to the PL. Compatible RMs, e.g., diverse accelerators, can, however, be swapped at runtime using only partial bitstreams.

We now present the threat model we assume, based on MPSoC and FPGAs.

### C. Threat Model

In this section we address Samsara's hardware, software and network/bus threat models (see Samsara's architecture in Fig. 1 for clarity). The threat model focuses on compute integrity and availability. Confidentially is out of scope for this paper.

*1) Samsara Controller and MP-Boot:* Samsara's *Controller* is a simple and easily-verifiable trusted-trustworthy ASIC and is, therefore, assumed to be tamper proof. Similarly, the microprocessor *MP-Boot* has a small footprint and is used only to run booting and reconfiguration software upon a signal from the *Controller*. This software is stored in encrypted form in, e.g., the TPM. Therefore, this software is assumed to preserve its integrity and not be called by another entity. Both the *Controller* and *MP-Boot* are detailed in Section III.

*2) PL Hardware Logic:* We assume a threat model where an adversary can inject hardware trojans [43], [44] or backdoors (e.g., in coalition with PS software [45]) in PL Tiles, during the design cycle [17], [18]. That is, we assume tiles in the PL and, thus, their bitstreams may contain malicious functionality that impacts *integrity*. Additionally, we assume the possibility of non-malicious threats caused by unintended vulnerabilities created at design time [18]. We do not focus on attacks that have been successful in breaking the *confidentiality* of bitstreams [19], [20] while they are at rest in memory.

To boost resilience against a compromised tile, several redundant tiles are run simultaneously in the PL following the *H-Quorum* agreement protocol — explained in Section III-B. We assume that a majority of these tiles are not simultaneously compromised in a short time window, which means they have diverse configurations or implementations to avoid common-mode failures. For example, diverse configurations of hardware modules are a possibility that is currently supported by some vendor tools, while different implementations are also possible via hardware description languages (HDLs) or with the support of the RISC-V instruction set architecture and respective frameworks [29], [30]. We also assume that tiles have a level of containment or isolation, by having the base PL design use methods like Xilinx's Isolation Design Flow (IDF) [46], which provides fault containment at the FPGA module level, so that an anomaly or vulnerability may not affect other modules of the PL directly.

*3) General Hardware: Application Cores* may fail arbitrarily since these do not have access to the *Controller*, *MP-Boot*, the TPM, or the PL. For further isolation guarantees, solutions like [47] can be used for flexible access control. We do not assume any side-channel attacks on the platform, the failure of the FPGA fabric itself, nor MPSoC-wide failures. This assumption is reasonable since FPGAs and MPSoCs in general are high-end hardware that are subject to rigorous testing. Still,

[2]These can, however, be reconfigured by reconfiguring the whole PL.

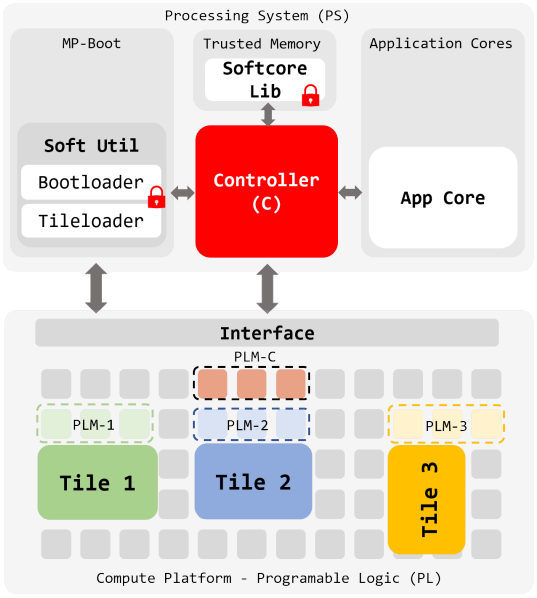


Fig. 1. The architecture of *Samsara* Rejuvenation Fault and Intrusion Tolerant Framework. The three tiles are spawned from three known softcores [28] to demonstrate diversity.

resilient clocks [48] mitigate some chip-wide common-mode faults and the recent trend towards interconnected chiplets further improves the physical decoupling of MPSoC components.

*4) Software:* Similarly, the attacker can compromise any application software that would be executed in the PS's Application Cores (see Section III-A) or in PL soft cores (if used as tiles). However, as we shall explain later, malfunctioning PS application software does not interfere with Samsara, and software running on the PL (if used) has no means to cause trouble beyond providing incorrect output, which Samsara addresses with *H-Quorum*. Furthermore, to safeguard Application Cores further, their core-to-NoC interface can be augmented with solutions such as [47].

*5) Network-on-Chip:* Moreover, via a compromised tile, the adversary may compromise the NoC by dropping or modifying exchanged messages [49], [50], namely by means of software-hardware coalition [45]. The adversary cannot, however, spoof, pretending to be another tile, as shown in Annex A.

## III. Samsara Resilient Computing Platform

### A. Architecture

We present a high-level architecture of *Samsara* in Figure 1. Samsara's architecture is composed of three main components: the *Controller*, the *Compute Platform*, and *MP-Boot Utilities*. The *Controller* manages the *Compute Platform* in the PL and interfaces the application requests from the *Application Cores* with it. Having such a critical role, the *Controller* is implemented in hardware (ASIC) that cannot be tampered with. To enable different configurations and extensions, the Controller stores default PL configurations and security keys in Tamper-Resistant Storage (TRS). The *Compute Platform* implements the compute *Tiles*, i.e., accelerators and critical functions, to be used by the applications. It is composed of FPGA-based tiles in the PL, loaded from bitstreams, encrypted and stored in the tile library (e.g., softcore library) in the PS's memory and later authenticated. Encryption provides basic design security to protect the design from copying or reverse engineering (which is not the focus of this paper), while authentication provides assurance that the bitstream provided for configuration is the unmodified bitstream created by an authorized user. Authentication verifies both data integrity and authenticity of the bitstream. Authorized users can still have corrupt or malicious bitstreams and these can also be subject to faults, hence the need for PL-side Rejuvenation to the same or a diverse instance, depending on whether faults are benign or malicious, respectively, as outlined in Section III-B.

With the benefit of reconfigurable hardware, tiles are subject to updates over time, e.g., to modify functionality, which makes them less secure than fixed function ASIC tiles. Consequently, the Compute Platform supports active replication of tiles to mask and detect faulty or misbehaving ones. This is protected and managed by the Controller. In addition, *Samsara* makes use of *Software Utilities*, i.e., the *Bootloader* and *Tileloader* (see Fig. 1), that are used occasionally by the *Controller* to manage and rejuvenate faulty tiles. These utilities are software-based components that should only interact with the PL at booting and reconfiguration time (detailed in the next section), respectively. Therefore, they are retained in memory in an encrypted form and are executed in a dedicated processor (*MP-Boot*) that is the only software processing unit with PL access. Furthermore, *MP-Boot* is not available for user-level application usage, to prevent malicious code from invoking the API that loads PL bitstreams.

In our architecture, the communication between the Controller and the Compute Platform Tiles can be done, e.g., 1) through a bus such as PCIe, or 2) through shared PL memory - BRAMs. While 1) would require the use of signatures for authentication, 2) does not, as we explain in Annex A. We shall use 2) for the remainder of the paper. As seen in Fig. 2, the Controller holds a BRAM in the PL to which it has read/write access. Here, it writes the Requests and keeps the Log of executed Requests for later state transfer after rejuvenating a tile (in the case of stateful applications). Tiles have read-only access to the Controller's BRAM in order to prevent malicious ones from modifying its contents. Then, tiles hold a BRAM of their own, to which they have read/write access and the Controller has read-only access. In this BRAM, tiles place their Replies as well as their local Log. Requests and Replies are written up to a maximum, after which a checkpoint is taken and the BRAMs reset to give space to new rounds of Requests/Replies. Checkpoints are saved in an on-chip SRAM outside the PL and can be fetched if needed by the Controller. Reset of the BRAMs can be triggered by the Controller or by a simple Reset IP in the PL.

The Controller's memory can instead be kept outside the PL as regular SRAM or as Tamper-Resistant Storage (TRS), which could improve security due to being outside reconfigurable logic, but, depending on the specific implementation

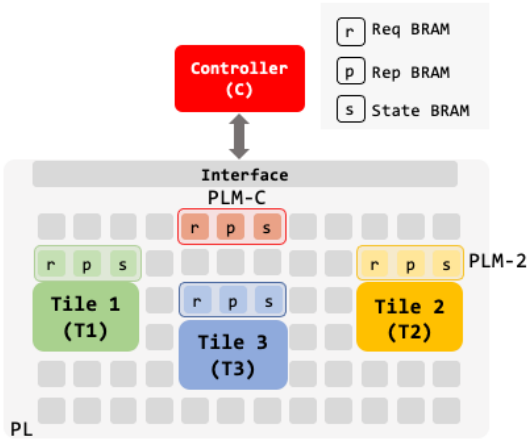


Fig. 2. Memory access in Samsara: each entity has *R/W* access to its own PL Memory (PLM). C has *R* access to all PLMs. Each Tile has *R* access to PLM-C.

(bus, type of storage), could as well increase access times.

### B. Operational Phases

*1) Overview:* Samsara operates in three phases: *Bootstrapping*, *Execution*, and *Rejuvenation*. In a nutshell, in the Bootstrapping phase, the Controller launches the Compute Platform. It executes the *Bootloader* utility that loads the main PL bitstream (i.e., the floorplan of the FPGA) and then the *Tileloader* utility following the configuration stored in the TRS. The configuration decides the types/versions of tiles to load from the Softcore Library as partial bitstreams, how many tile replicas are run, when and how to rejuvenate tiles, etc. When the Compute Platform is ready for use by the applications, the *Execution* phase begins. In this phase, the Controller receives requests from the applications, assigns each request an unique ID, and sends them to all the tiles of the Compute Platform, by running the lightweight quorum-based Byzantine agreement *H-Quorum* (detailed next). The tiles execute the same request simultaneously[3] and reply to the Controller. The latter verifies if a majority of the received responses are matching and forwards the reply back to the application. In case there is any mismatch or delay detected, the corresponding tile is to be replaced. This announces the launch of the *Rejuvenation* phase. In this phase, the Controller destroys the faulty tile and replaces it with another one of the same type (i.e., refreshing it) or by another diverse version. This follows a policy defined by the Controller. Rejuvenation ends by completing the state transfer to the newly loaded tile if the application is stateful.

Next, we explain the three phases of Samsara in detail.

*2) Bootstrapping Phase:* This phase is only launched when the MPSoC is started. It aims at preparing the Compute Platform through loading the necessary main and partial (i.e., tile) bitstreams to the PL section with the following steps:

1) Controller sets its status: $\mathbf{status} \leftarrow \mathit{Loading}$ and accepts no requests from applications.
2) Controller verifies and launches the *Bootloader* in the dedicated microprocessor. This loads the main bitstreams to the PL. The main bitstream represents the basic configuration on which tile bitstreams are loaded afterwards into specific containers defined by the main one.
3) Controller verifies and launches the *Tileloader* using the stored configuration in the TRS. The Controller passes the configuration parameters **Config** to the Tileloader. The retained configurations in the TRS are described in Table I.
4) Tileloader loads the bitstreams to the PL and notifies the Controller when the tiles are ready. This avoids the case when the latter is unresponsive or faulty. The Controller sets a timer while waiting for the tiles *status* to become *Ready*.
5) Tiles notify the Controller when *Ready*. This asserts the info sent by the Tileloader in case it is unresponsive or faulty.
6) If the Controller received the expected number of Ready messages from the tiles, as defined in **Config** before the timer's expiry, it sets its $\mathbf{status} \leftarrow \mathit{Ready}$ and starts accepting application requests. Otherwise, the Controller launches the *Rejuvenation* phase in *partial-mode* if a minority of tiles are faulty or slow, or in full-mode otherwise.

*3) Execution Phase:* This phase is dedicated to the execution of application requests through a novel lightweight quorum-based *Byzantine* agreement protocol, we call it *H-Quorum*. *H-Quorum* has a simple message exchange pattern, depicted in Fig. 3, that is tailored for low latency. In a nutshell, an application sends its requests to the Controller that mediates the requests and responses with the compute tiles. The Controller sends each request directly to all tiles and collects their responses. A reply is sent to the application if a majority, i.e., $f + 1$ out of $2f + 1$, of responses match.

TABLE I
CONTROLLER CONFIGURATION.

| Attribute | Description |
|---|---|
| *softcore* | compute softcore types (stored as bitstreams) |
| *version* | softcore diverse versions that are functionally similar but exhibit different design, implementation, or configuration. |
| *min-tiles* | minimum number of tiles allowed (could be defined by the agreement protocol used). |
| *max-tiles* | maximum number of tiles allowed, limited by PL capacity and by floorplanning[4]choices at design time. |
| *stateful* | boolean is *True* if the application is stateful. |
| *rejuv-policy* | defines the rejuvenation modes: Refresh/Diversify, Replace/Relocate, Scale-in/Scale-out, or Reactive/Proactive. |

[3]Simultaneous here does not mean in lock step.

[4]Floorplanning refers to the set of physical constraints used to control how logic is placed in the PL. The definition of dynamically reconfigurable zones is done at this stage and defines the containers these partitions can be in.

Otherwise it recovers by launching the rejuvenation phase.

*H-Quorum* is inspired by *Quorum* [31], that is proposed for Internet-based distributed service settings. *Quorum* follows a single round-trip direct messaging pattern between a client and all the $3f + 1$ replicas, assuming $f$ faulty replicas, thus making it the Byzantine agreement protocol with the lowest theoretical latency [31], [32]. This makes it the preferred choice in our case since low latency is paramount in hardware accelerator applications. Nevertheless, Quorum has several shortcomings that make it unfeasible for Internet-based settings, and actually impede its adoption [31], [32]. In particular, (i) Quorum assumes the client is trusted since it directly sends requests to all replicas and collects their responses, i.e., without a primary replica as in classical protocols [51], [52]; (ii) requiring $3f + 1$ replicas to maintain safety and liveness under partially-synchronous networks, which is deemed costly; and (iii) requiring a recovery phase to a backup protocol (e.g., PBFT [51]) with an expensive state transfer mechanism due to logs' matching, necessary for safe view-change under failures. Fortunately, by leveraging the hardware environment, *H-Quorum* tweaks Quorum in order to bypass these shortcomings as follows.

**A. Controller acts as a trusted half-primary.** *H-Quorum* avoids the trusted client issue by having the Controller mediating the messages between the application and the tiles. Being an ASIC hardware, the Controller is arguably highly trusted provided that its footprint is not big (as we convey in Section IV) in order to be easily verifiable. For this, we decided to simplify the Controller that can be observed as "half-primary": it mediates the messages and performs the log matching, but it does not compute the requests as the tiles do (since computation logic can have a large footprint in accelerators).

**B. Requiring a majority of correct tiles.** *H-Quorum* requires only $2f + 1$ tiles to tolerate a maximum of $f$ faulty tiles to maintain liveness and safety. This is possible since the communication in Samsara is hardware-based, which provides some network containment and thus exhibits some network synchrony. This allows us to make some time bounds on responses before launching the rejuvenation phase, contrary to Quorum that cannot differentiate between a Byzantine replica and a slow or faulty network. Even in the case where a faulty tile is slow or attempts to launch a DoS attack [25] on the network bus or jam the Controller, the latter can detect this easily and launch the rejuvenation phase that can change the faulty tile and/or the network routes (see next).

**C. Safe state-transfer upon recovery.** *H-Quorum*'s state-transfer is simple as it does not require log-matching, contrary to Quorum. Indeed, Samsara's Controller has a restricted memory in the PL, i.e, PLM-C in Fig. 1, in which it retains the state and logs, that are equivalent to those of *correct* Tiles. This makes state-transfer fast and importantly preserves the correct state across rejuvenation phases (which is analogous to *view-change* in Quorum).

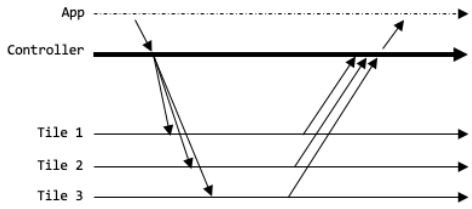


Fig. 3. Message exchange in *Samsara*'s *H-Quorum Execution* phase.

We now present the steps of *H-Quorum* as follows:

1) An application $A$ sends a request $req$ to the Controller $C$.
2) $C$ creates a message $m = uid, req, H(req)$, where $uid$ is the message's *unique ID* representing an incremented sequence number safely assigned by the Controller, and $H(Req)$ is the hash digest of $Req$. $C$ sends $m$ to each tile $T_i$ by placing $m$ message in the read-shared PLM-C memory. The Controller sets a timer $timer_i$ and waits for a reply from at least $f + 1$ tiles before expiry.
3) Upon the receipt of $m$, a tile $T_i$ accepts $m$ after verifying its hash digest and $uid$. Then, $T_i$ computes $req$ and puts the output $rep$ in a response message $r = uid, rep, H(rep), tid$ that uses the corresponding request $uid$, as well as $T_i$'s ID $tid$. The latter "sends" $r$ to $C$ by placing it in its shared $PLM_i$ slot to be read by $C$.
4) For each tile $T_i$, $C$ reads the reply $r_i$ from the corresponding $PLM_i$, it verifies its hash digest and $uid$. Then, $C$ tries to check if a majority of these replies are matching as follows:
   a) If $C$ received matching (i.e., having equivalent output) replies $rep$ from all the $2f + 1$ tiles before the $timer_c$ expires, it forwards the $rep$ to the application $A$. In addition, if $A$ is a stateful application, $C$ *temporally* retains the $req|rep$ in its allocated PL $PLM_C$ slot to be used for state transfer under faults.
   b) If $C$ received only $f + 1$ matching replies $rep$ before $timer_c$ expires, it forwards $rep$ to $A$; however, it also launches the *Rejuvenation* phase with *partial-mode* to replace the faulty or slow tile (see next). In addition, if $A$ is stateful, $C$ *temporally* retains the $req|rep$ in its allocated PL $PLM_C$ slot to be used for state transfer under faults.
   c) Otherwise, if less than $f + 1$ matching responses $rep$ are received before $timer_c$ expires, $C$ launches the *Rejuvenation* phase with *full-mode* to replace all the tiles and their (network) routes (see next).
   d) Finally, if agreement was successful, $C$ marks the request as done.

*4) Rejuvenation Phase:* This phase is launched by running the *Tileloader* to carry on a *partial-mode* or *full-mode*

replacement of the Compute Platform. *Partial-mode* happens through reloading new tiles to the PL, e.g., to refresh or diversify tiles; whereas the *full-mode* rejuvenates the tiles as well as their routes (network), all static and dynamic parts. Both stateless or stateful applications can be considered. For the former, no state transfer upon rejuvenation is required, while for the latter rejuvenated replicas read or copy the state saved in PLM-C, depending on whether they need a local copy. Examples of stateful applications include, e.g., image processing. Rejuvenation steps and modes are explained as follows:

1) The Controller $C$ changes its status to $\texttt{status} \leftarrow Loading$ and stops accepting application requests.
2) $C$ invokes the *Tileloader* in two possible modes: (i) *Partial-mode* where $C$ launches the *Tileloader* to flush the faulty or slow tiles and their corresponding BRAMs in the PLMs. It then reloads the same or diverse bitstreams as new tiles (see next). In the (ii) *full-mode*, $C$ invokes $Tileloader$ on the entire Compute Platform to be rejuvenated, meaning that all PL resources (softcores, BRAMs, Routes, Registers, etc.) are destroyed and reloaded. This is necessary when a majority of matching replies is not achieved or the Compute Platform is slow or unresponsive. In the case of stateful applications, the state saved in the PLM-C BRAMs is first checkpointed and saved in the on-chip SRAM before triggering the Tileloader, to ensure no state progress is lost. The *Tileloader* reloads the Compute Platform following a predefined policy as follows:
    a) Refresh/Diversify: reloads the exact (Refresh) or diverse (Diversify) tile version to the PL. Refresh is convenient for transient faults or attacks while Diversify boosts the Compute Platform resilience through minimizing common-mode failures. The fault threshold $f$ is assumed to hold when Rejuvenating.
    b) Replace/Relocate: reloads the tile in the same or different PL partition location. This precludes potential issues in the PL fabric location and changes the communication routes in the case of AXI issues/attacks. Relocation, however, assumes a block exists for the relocation of the partial bitstream into a new location and that its interface has been taken into account at design time during floorplanning.
    c) Scale out/in: changes the number of simultaneous tiles adapting to severity levels, i.e., as $f$ in $2f+1$ changes. This is bounded by the *min-tiles* and *max-tiles* parameters in configuration (see Table I).
    d) Reactive/Proactive: invokes Tileloader in reaction to a fault raised by *H-Quorum* or in a periodic fashion regardless of faults, seeking proactive resilience (especially useful against *Advanced Persistent Threats*).
3) The *Tileloader* notifies $C$ with *Ready* status.
4) $C$ sets a timer and waits for *Ready* response from the corresponding rejuvenated tiles.
5) If $C$ received the expected number of *Ready* messages from the tiles before the timer's expiry, it sets its $\texttt{status} \leftarrow Ready$, transfers the checkpointed state back to PLM-C, and starts accepting application requests. Otherwise, the Controller launches the *Rejuvenation* phase in *partial-mode* if a minority of tiles are faulty or slow, or in full-mode otherwise.

## IV. Evaluation

The proposed framework was evaluated on a Xilinx Zynq Ultrascale+ ZCU102 FPGA board, running at 100MHz. The choice of frequency has the goal of fair comparison, as the references against which we compare Samsara run also at 100MHz. We instantiated 3 reconfigurable partitions (RP) in the PL (i.e., 3 tiles), each with 2 possible reconfigurable modules (RM). RMs refer to the possible partial bitstreams that fit into the RPs which are dedicated to tile implementation. The choice of the type of IPs to run on the PL, i.e., what tiles contain, is application dependent and not a property of Samsara. RMs can only be loaded into a full bitstream with the RPs designed to contain them. However, by triggering full Rejuvenation and replacing the whole PL, it is possible to install a new full bitstream that can interface with different types of RMs. Namely, a bitstream can be designed to have more generic RP boundaries which interface more easily with different types of tiles. In order to evaluate the most complex and costly case possible for Samsara, we designed the tiles to be softcores (Xilinx's MicroBlazes in standard configuration). Simpler cases include cryptographic or machine learning accelerators, or any other type of module. As a proof of concept, the Controller is simulated in the PL and not implemented as an ASIC (as it is would require manufacturing and be costly to experiment with over several iterations). It is then instead implemented as software running on a PL softcore. Tiles' cores and other components in the PL (e.g., memory controllers, timers, etc) are connected using AXI4-Lite. The Bootloader and Tileloader are run in one of the FPGA's PS ARM cores.

Table II details the architectures/protocols evaluated in this paper. We evaluate two fault models for Samsara, (1) one where we assume the MPSoC NoC and AXI buses to be correct and thus do not use message hashes; and (2) one where we acknowledge network attacks (as described in the threat model in Section II-C) and, thus, use hashes (*Samsara HW_H* in Table II). Both scenario (1) and (2) use 3 partitions as described above. We compare these implementations with a baseline of having just one tile, in this case, one softcore (*SC* in Table II), with a regular triple-modular redundant (TMR) architecture (with and without hashing), the *iBFT* protocol from [34] and a shared-memory-based implementation of *MinBFT* [10] akin to that described in [34]. Using 3 tiles is the minimum to have majority (i.e., if $f = 1$, then $n = 2f+1 = 3$) and allows better comparison to basic redundancy (TMR) and [34], since it uses 3 replicas as well. The comparison with *iBFT* is relevant given it too implements a low-level agreement protocol like *H-Quorum*. The other protocol presented in [34],

TABLE II
PROTOCOL EXPLANATIONS. HW = HARDWARE AND SW = SOFTWARE.

| SC | Single Core (Baseline) |
|---|---|
| TMR | 3 Cores |
| TMR HW_H | 3 Cores with HW Hashing (SHA-256) |
| H-Quorum | 3 Cores |
| H-Quorum HW_H | 3 Cores with HW Hashing (SHA-256) |
| iBFT [34] | 3 Cores |
| MinBFT [10] SW_H | 3 Cores with SW Hashing (SHA-256) |

*Midir* [47], is slower than *iBFT* and, therefore, is not as suitable for comparison. Samsara's version with no hashes also helps to compare with *iBFT*, which assumes a correct network. *MinBFT* is the most well-know state-of-the-art BFT protocol that implements architectural hybridization [53] and, as such, was a good target for comparison.

The protocols are compared in terms of agreement latency, reconfiguration latency, latency footprint, area usage, power consumption and communication steps. Area usage and power consumption are SWaP (space, weight and power) metrics, which are relevant in the construction of on-chip, often resource restricted systems like cyber-physical systems (CPS) and IoT.

### A. Latency

Latency is measured with a combination of an AXI Timer and an AXI Interrupt Controller module in the PL, and a Xilinx timer API that outputs the cycle count.

*a)* ***Protocol Latency****:* Fig. 4 compares Samsara's *H-Quorum* (H-Q in the figure) with 256-bit messages (with and without hashing) with our baseline (*SC*), *iBFT* and the shared-memory implementation of *MinBFT*, in logarithmic scale. It is observable that the use of 3 agreeing tiles as opposed to 1 only has a latency overhead of 1054 cycles (i.e., 10.54 ms, considering a core running at 100MHz). Additionally, *H-Quorum* with hardware-based hashing is still 3241 cycles faster (and 4834 cycles faster without hashing) than *iBFT* (no hashing), meaning it is 21.6% (and 32.2% respectively) faster than *iBFT*. In Samsara, we use a SHA-256 PL module that we connected with each core tile (so 3 SHA-256 modules, one for each tile) to perform the hashing of the messages. The lower cycle count in comparison with *iBFT* is expected, as *iBFT* has more communication steps and number of exchanged messages, as seen in Table III. All measurements are done for a null operation, meaning the depicted latency represent the protocol latency (i.e., message exchange, which in this case is done through reads and writes from/to BRAM) and does not include request execution latency.

To fairly compare with MinBFT which has hashing implemented in software, we present the results for an *H-Quorum* implementation that uses SHA-256 in software rather than hardware, *H-Quorum* SW_H. As can be seen, *H-Quorum* performs faster due to having less protocol steps.

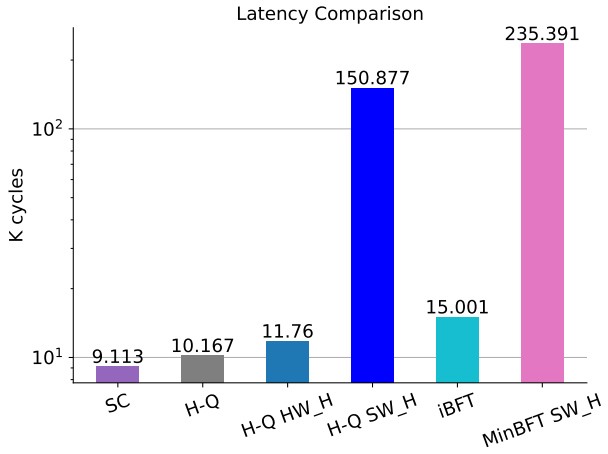


Fig. 4. Latency comparison of Samsara's H-Quorum (H-Q in the graph) with and without hashing (null operation) against a baseline (single core), iBFT and MinBFT. The graph is presented in logarithmic scale, but the values seen atop the bars are the true cycle count.

TABLE III
COMPLEXITY ESTIMATION OF TMR, SAMSARA (H-QUORUM), IBFT AND MINBFT.

| Protocol | Communication Steps | Number of Messages |
|---|---|---|
| TMR | 1 | O(n) |
| H-Quorum | 2 | O(n) |
| iBFT | 5 | $O(n^2)$ |
| MinBFT | 4 | $O(n^2)$ |

In the FPGA realm, with hardware accelerators significantly reducing latency (e.g, SHA-256 in hardware takes on average 1593 cycles, while in software, running on a MicroBlaze core, it takes 128643 cycles), the biggest sources of latency are clock domain crossing and the nature of the bus protocol implementation (for simplicity, we used Xilinx's implementation of AXI4-Lite, which is slower than AXI4-Full and does not allow burst access) for memory access. With more complex implementations of the AXI4 bus and higher clock frequencies, better performance can be achieved.

*b)* ***Protocol Footprint****:* Fig. 5 a) shows how *H-Quorum*'s latency (null operation) becomes more negligible as application complexity scales. We ran SHA-256 on a 256-bit message in software (i.e., running on the MicroBlaze core and not the hardware accelerator used in Samsara) and performed complex double-precision multiplication on a one-dimensional array of very small size (10) for comparison; and observed that running the *H-Quorum* agreement protocol brings no significant overhead to operation execution. Naturally, these operations can be run on dedicated accelerators, which would replace cores in the tiles and have lower latency. Namely, *H-Quorum* itself will have better performance with the envisioned ASIC controller, as it will execute as hardware logic. However, given we are emulating the *Controller* in software running on a PL

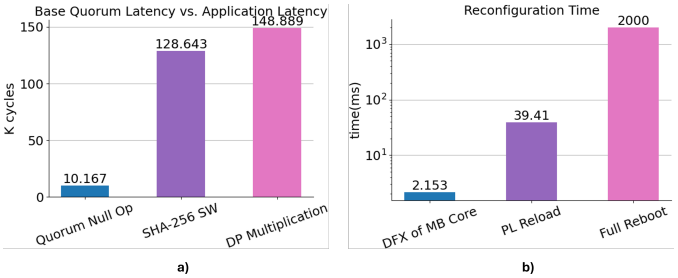


Fig. 5. a) Influence of application complexity on agreement latency. The null operation represents the latency of the Quorum protocol itself. DP Multiplication means double precision multiplication. b) Reconfiguration time in milliseconds (ms) of DFX (dynamic partial reconfiguration) vs. total PL reboot vs. total board reboot, in logarithmic scale.

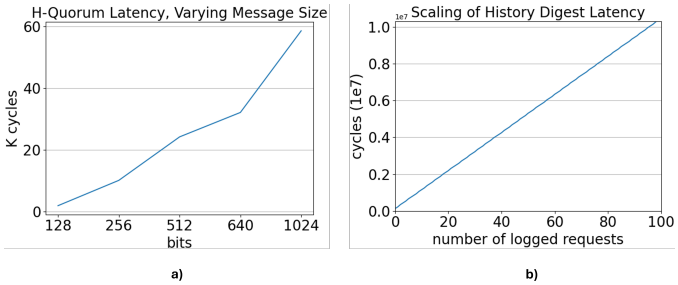


Fig. 6. a) Latency of *H-Quorum* with varying message sizes for null operation. b) Scaling of history digest latency (each round), according to the number of logged requests (up to a checkpoint of 100) for stateful applications.

core, it is fair to compare against software implementations of the aforementioned operations. Even with, e.g., a hardware implementation of SHA-256 (which we use in Samsara, and takes 1593 cycles for a 256-bit input message), *H-Quorum* is, not only still fast, but does not incur great overhead based on replication or agreement, as previously seen in Fig. 4 which shows single core execution as taking only 1054 cycles less.

*c) **Reconfiguration Latency**:* We then analyzed reconfiguration time, measuring the time to partially reconfigure the PL (i.e., replacing a tile, in this case, a MicroBlaze tile) against the time to fully reconfigure the whole PL, and a full platform reboot (i.e., including running the Bootloader again and initializing the PS), as seen in Fig. 5 b). The partial and full PL reconfiguration are measured by the PS's ARM core, while the full platform reboot is measured with the help of the Xilinx application development tool, Vitis which logs the time elapsed from the beginning of the reboot process until it has successfully finished, excluding the time it takes to download the bitstreams and `.elf` files to the board through JTAG. Given that, by rebooting the whole board we do not have any cores available to perform the latency measurement, we could not check the time elapsed by booting the board from the FPGA's SD card (as done in the PL reconfiguration measurements), and therefore we required the time measurement provided by the tool. As such, for the full reboot we programmed the board via JTAG to use the tool's time measurements, which gave us very precise values. Note that the full reboot latency presented in Fig. 5 b) does not account for applications loaded into the PS, besides the Bootloader and Tileloader.

As can be seen, there is a significant decrease in the time required to reconfigure a tile versus the full PL reload, which results from replacing only a small part of the architecture. One MicroBlaze core represents only 4.9% of the Samsara design, i.e., of the full base bitstream. The full reboot, on the other hand, requires reset and initialization of the PS, running the FSBL and programming of the PL. The presented results were obtained with un-encrypted bitstreams, meaning they do not account for bitstream decryption, however, the reduced latency gained from PR would still hold. For stateful applications that need a local state copy, state hashing (digest) and transfer latency depends on message and checkpoint size. Fig. 6 a) depicts *H-Quorum*'s latency as a function of varying message sizes, again using a null operation. As expected, latency grows linearly with the rising memory accesses. Our AXI4-Lite interface performs writes of 32 bits, with number of accesses scaling linearly with message length. Other bus implementation options include burst or streaming. Similarly, Fig. 6 b) shows hashing latency (computed in software) with a growing checkpoint size, up to 100.

### *B. Resource Usage and Power Consumption*

Next we look at resource usage, as seen in Fig. 7. The metrics presented in the graph are LUTs (Look-Up Tables) and registers, which are the basic units of area measurement on the FPGA fabric. We divided each architecture into 3 categories: base architecture (tiles, memory controllers, timers, interrupts, etc), AXI Interconnect and DFX[5] Controller (not to be mistaken with Samsara's *Controller*). The DFX Controller in the PL manages the low-level loading of hardware bitstreams from memory. The separation of the AXI from the rest of the architecture has the intent to show the large occupation of the bus in comparison to the rest of the design, it is still, however, part of the PL along with the DFX Controller. The AXI Interconnect module connect several PL modules and thus multiplexes accesses among them, consuming a large number of LUTs and registers. This is the reason why AXI increases significantly from using a single tile (*SC*) to using 3 (Samsara). For simplicity and easy comparison with the selected state of the art, we implemented Samsara with a main AXI Interconnect for connecting most modules and used the AXI4-Lite interface. Separating it into multiple smaller ones can lower resource consumption as there is less multiplexing, while using AXI4-Full can bring lower latency. DFX is naturally only present in Samsara, as it is the only solution presented here that does dynamic partial reconfiguration, i.e., swapping tile contents at runtime. This represents a trade-off of between a higher area and resource usage, and the flexibility and speed of hardware reconfiguration. Additionally, the area increase in TMR and Samsara versus *SC* is a trade-off as

[5]DFX stands for Dynamic Function eXchange and refers to the Xilinx functionality used to perform dynamic partial reconfiguration at runtime.

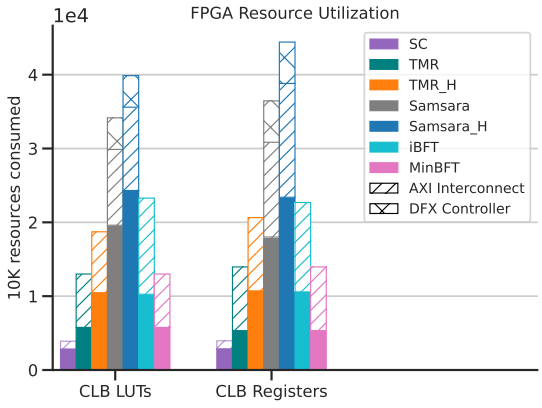


Fig. 7. FPGA resource utilization.

TABLE IV
TOTAL ON-CHIP POWER CONSUMPTION IN WATTS (W).

| Scheme | Power (W) |
|---|---|
| SC | 3.62 |
| TMR | 3.77 |
| TMR HW_H | 3.81 |
| Samsara | 5.06 |
| Samsara HW_H | 5.11 |
| iBFT | 3.97 |
| MinBFT | 3.77 |

well, i.e., it is the cost of resilience. In the evaluated board (ZCU102), the utilization for Samsara still represents only 12.68% of available LUTs and 6.65% of available registers, with 3.24%/1.92% (respectively) for the DDR4 memory where bitstreams were stored, 2.38%/1.63% for the AXI Interconnect, and 1.59%/1.02% for the DFX Controller.

Table IV shows the predicted power consumption for all evaluated architectures. This power analysis is taken from the implemented netlist outputted by the Xilinx design, synthesis and implementation tool Vivado. It represents activity derived from constraint files, simulation files and vectorless analysis and is provided as expectations, not physically tested results.

As expected, Samsara consumes approximately 1.297 W more than the average of the other architectures, which comes from the usage of DFX. Nevertheless, in Samsara, 2.744 W out of the 5.064 W come from the PS and not the PL where the tiles reside. It is also noticeable that the usage of 3 tiles (TMR and Samsara) versus 1 tile leads only to an increase of 0.146 W, meaning that replication or multiple tiles participating in *H-Quorum* does not massively affect power consumption. This, however, depends on the contents of the tile.

## V. Related Work

TMR has been used in the realm of critical embedded systems, e.g., in the primary flight computers of Boeing 777's fly-by-wire (FBW) system [54]. Similarly, passive redundancy can also be seen in Airbus' dependability-oriented approach to FBW [55]. The concept was extended to multi-phase tightly-synchronous message-passing protocols in the CPS domain [56], [57]. Unlike TMR, Byzantine fault tolerant (BFT) SMR algorithms [21], [51] aim to tolerate both accidental and malicious faults, by reaching consensus with $|Q| = 2f + 1$ out of $n = 3f + 1$ replicas. Architectural hybridization [58] proposes an additional trusted-trustworthy component to further reduce the size of $n$ and $Q$ to $2f + 1$ and $f + 1$, respectively. This technique has been used in protocols such as MinBFT and CheapBFT [10], [59]. BFT algorithms have traditionally been implemented on distributed systems and have seen little work in emerging MPSoC critical systems like CPS and IoT, due to their added latency and replication costs. *Midir* presents an architecture and an on-chip BFT-like protocol for improving the safety and resilience of low-level software running on MPSoCs as well as their access control mechanisms. It does so through minimalist hardware logic, T2H2, that provides secure voting on critical operations and access control. Similarly, *iBFT* [34] is an efficient consensus algorithm that leverages shared memory. Nevertheless, neither of these works allow the flexilibity of accelerator reconfiguration nor dynamic rejuvenation. In [24], a partitioning technique enabling the use of COTS NoC-based MPSoC for mixed criticality systems is proposed, however, it is intended to have a purely software implementation as a module of a real-time operating system. Another work [60] investigates and evaluates fault-tolerance techniques for the UltraScale+ MPSoC FPGAs, but it targets only accidental faults in the form of Single-Event Upsets (SEUs). FPGA partial reconfiguration was demonstrated for recovering failed ECUs in automotive networks in [61]. Contention on shared resources in the context of MPSoC-based safety critical applications is explored in [62]. Alcaide et al. [63] develop safety measures in the PL of COTS MPSoCs, but do not consider PL-side faults and intrusions. In [64], a secure framework to implement logic-locking, extended with secure boot for Xilinx FPGAs is presented. Furkan [65] provides a survey on secure FPGA configuration and security of FPGA modules.

## VI. Conclusion

We introduced *Samsara*, the first secure and resilient platform for programmable hardware. Our work shows that leveraging the hardware properties of programmable hardware, like FPGA and GPU, paves the way to design lightweight and low latency BFT variants, such as *H-Quorum*. Interestingly, we show that hardware rejuvenation is also possible at a negligible latency overhead. To improve independence of failures, Samsara supports the rejuvenation to diverse implementations from a pool of versions. This is possible either by simple reconfiguration tweaks or via using off-the-shelf implementations. In particular, typical compute IP implementations, e.g., SHA-256, are available as open source. Samsara, however, imposes an additional resource utilization overhead over non-replicated systems, which we believe is a reasonable price for security and resilience. This is not as critical as replicated computers since a programmable fabric might often be under-utilized.

## REFERENCES


[1] I. M. Safarulla and K. Manilal, "Design of soft error tolerance technique for FPGA based soft core processors," in *IEEE International Conference on Advanced Communications, Control and Computing Technologies*, 2014, pp. 1036–1040.

[2] B. Shashidhara, S. Jadhav, and Y. S. Kim, "Reconfigurable Fault Tolerant Processor on a SRAM based FPGA," in *IEEE International Conference on Electro Information Technology (EIT)*, 2020, pp. 151–154.

[3] H. Feng, W. Li, L. Chen, S. Wang, J. Zhou, C. Tian, and Y. Zhang, "Precise Fault Injection and Fault Location System for SRAM-based FPGAs," in *IEEE 10th Joint International Information Technology and Artificial Intelligence Conference (ITAIC)*, vol. 10, 2022, pp. 2371–2376.

[4] V. Costan and S. Devadas, "Intel SGX explained," *Tech. rep., Massachusetts Institute of Technology*, 2016. [Online]. Available: https://eprint.iacr.org/2016/086.pdf

[5] ARM, "ARMTrustZone," *ARM*, Accessed on August 2022. [Online]. Available: https://www.arm.com/products/security-on-arm/trustzone

[6] S. L. Kinney, *Trusted platform module basics: using TPM in embedded systems*. Elsevier, 2006.

[7] M. Sabt, M. Achemlal, and A. Bouabdallah, "Trusted execution environment: what it is, and what it is not," in *IEEE Trustcom/BigDataSE/ISPA*, vol. 1. IEEE, 2015, pp. 57–64.

[8] T. Distler, C. Cachin, and R. Kapitza, "Resource-Efficient Byzantine Fault Tolerance," *IEEE Transactions on Computers*, vol. 65, no. 9, pp. 2807–2819, 2016.

[9] J. Liu, W. Li, G. O. Karame, and N. Asokan, "Scalable byzantine consensus via hardware-assisted secret sharing," *IEEE Transactions on Computers*, vol. 68, no. 1, pp. 139–151, 2018.

[10] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Verissimo, "Efficient Byzantine Fault-Tolerance," *IEEE Transactions on Computers*, vol. 62, no. 1, pp. 16–30, 2013.

[11] AMD, "Zynq Ultrascale+ Device Technical Reference Manual," *Xilinx*, 2020. [Online]. Available: https://docs.xilinx.com/v/u/en-US/ug1085-zynq-ultrascale-trm

[12] A. Shoker, P. Esteves-Verissimo, and M. Völp, "The Path to Fault- and Intrusion-Resilient Manycore Systems on a Chip," in *53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks-Supplemental Volume (DSN-S)*. IEEE, 2023, pp. 157–162.

[13] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, N. Pemberton, E. Amaro, C. Schmidt, A. Chopra *et al.*, "FireSim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud," in *ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 29–42.

[14] P. Mantovani, D. Giri, G. Di Guglielmo, L. Piccolboni, J. Zuckerman, E. G. Cota, M. Petracca, C. Pilato, and L. P. Carloni, "Agile SoC Development with Open ESP : Invited Paper," in *IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2020, pp. 1–9.

[15] H. Singh, M.-H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. Chaves Filho, "MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications," *IEEE Transactions on Computers*, vol. 49, no. 5, pp. 465–481, 2000.

[16] A. Boutros and V. Betz, "FPGA architecture: Principles and progression," *IEEE Circuits and Systems Magazine*, vol. 21, no. 2, pp. 4–29, 2021.

[17] S. Mal-Sarkar, A. Krishna, A. Ghosh, and S. Bhunia, "Hardware Trojan Attacks in FPGA Devices: Threat Analysis and Effective Countermeasures," in *Proceedings of the 24th Edition of the Great Lakes Symposium on VLSI*, 2014, pp. 287–292.

[18] A. Duncan, F. Rahman, A. Lukefahr, F. Farahmandi, and M. Tehranipoor, "FPGA bitstream security: a day in the life," in *IEEE International Test Conference (ITC)*. IEEE, 2019, pp. 1–10.

[19] M. Ender, A. Moradi, and C. Paar, "The unpatchable silicon: a full break of the bitstream encryption of Xilinx 7-series FPGAs," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1803–1819.

[20] M. Ender, G. Leander, A. Moradi, and C. Paar, "A cautionary note on protecting Xilinx's UltraScale+ bitstream encryption and authentication engine," in *IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2022, pp. 1–9.

[21] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, p. 382–401, jul 1982. [Online]. Available: https://doi.org/10.1145/357172.357176

[22] AMD, "Using Encryption and Authentication to Secure an UltraScale/UltraScale+ FPGA Bitstream," *Xilinx*, 2022. [Online]. Available: https://www.xilinx.com/content/dam/xilinx/support/documents/application_notes/xapp1267-encryp-efuse-program.pdf

[23] T. Zhang and R. B. Lee, "Secure cache modeling for measuring side-channel leakage," *Technical Report, Princeton University*, 2014.

[24] S. Avramenko and M. Violante, "RTOS solution for NoC-based COTS MPSoC usage in mixed-criticality systems," *Journal of Electronic Testing*, vol. 35, pp. 29–44, 2019.

[25] S. Charles and P. Mishra, "A survey of network-on-chip security attacks and countermeasures," *ACM Computing Surveys (CSUR)*, vol. 54, no. 5, pp. 1–36, 2021.

[26] R. E. Lyons and W. Vanderkulk, "The use of triple-modular redundancy to improve computer reliability," *IBM journal of research and development*, vol. 6, no. 2, pp. 200–209, 1962.

[27] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM (JACM)*, vol. 32, no. 2, pp. 374–382, 1985.

[28] M. Makni, M. Baklouti, S. Niar, M. W. Jmal, and M. Abid, "A comparison and performance evaluation of FPGA soft-cores for embedded multi-core systems," in *11th International Design & Test Symposium (IDT)*. IEEE, 2016, pp. 154–159.

[29] A. Waterman, Y. Lee, D. Patterson, K. Asanovic, V. I. U. level Isa, A. Waterman, Y. Lee, and D. Patterson, "The RISC-V instruction set manual," *Volume I: User-Level ISA', version*, vol. 2, pp. 1–79, 2014.

[30] I. Marques, C. Rodrigues, A. Tavares, S. Pinto, and T. Gomes, "Lock-V: A heterogeneous fault tolerance architecture based on ARM and RISC-V," *Microelectronics Reliability*, vol. 120, p. 114120, 2021.

[31] R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić, "The next 700 BFT protocols," in *Proceedings of the 5th European conference on Computer systems*, 2010, pp. 363–376.

[32] J.-P. Bahsoun, R. Guerraoui, and A. Shoker, "Making BFT protocols really adaptive," in *IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2015, pp. 904–913.

[33] AMD, *ZCU102 Evaluation Board User Guide (UG1182)*, 2023.

[34] I. Pinto Gouveia, "Architectural support for hypervisor-level intrusion tolerance in mpsocs," Ph.D. dissertation, University of Luxembourg, Esch-sur-Alzette, Luxembourg, 2022.

[35] AMD, *Zynq 7000 SoC Technical Reference Manual (UG585), version 1.12.2*, 2018.

[36] T. C. Group, *TPM Main, Part 1 Design Principles, Specification Version 1.2, Revision 116*, 2011.

[37] A. K. Sharma, "Onboard credentials: Hardware assisted secure storage of credentials," Master's thesis, Helsinki University of Technology, 2007.

[38] D. Cock, A. Ramdas, D. Schwyn, M. Giardino, A. Turowski, Z. He, N. Hossle, D. Korolija, M. Licciardello, K. Martsenko, R. Achermann, G. Alonso, and T. Roscoe, "Enzian: An Open, General, CPU/FPGA Platform for Systems Software Research," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 434–451. [Online]. Available: https://doi.org/10.1145/3503222.3507742

[39] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua *et al.*, "Baring it all to software: Raw machines," *Computer*, vol. 30, no. 9, pp. 86–93, 1997.

[40] L. Liu, J. Zhu, Z. Li, Y. Lu, Y. Deng, J. Han, S. Yin, and S. Wei, "A survey of coarse-grained reconfigurable architecture and design: Taxonomy, challenges, and applications," *ACM Computing Surveys (CSUR)*, vol. 52, no. 6, pp. 1–39, 2019.

[41] I. Corporation, *PCI Express® Device Security Enhancements*, 2018.

[42] K. Vipin and S. A. Fahmy, "FPGA dynamic and partial reconfiguration: A survey of architectures, methods, and applications," *ACM Computing Surveys*, vol. 51, no. 4, pp. 1–39, 2018.

[43] R. S. Chakraborty, I. Saha, A. Palchaudhuri, and G. K. Naik, "Hardware Trojan insertion by direct modification of FPGA configuration bitstream," *IEEE Design & Test*, vol. 30, no. 2, pp. 45–54, 2013.

[44] P. Swierczynski, G. T. Becker, A. Moradi, and C. Paar, "Bitstream fault injections (BiFi)–automated fault attacks against SRAM-based FPGAs," *IEEE Transactions on Computers*, vol. 67, no. 3, pp. 348–360, 2017.

[45] D. M. Ancajas, K. Chakraborty, and S. Roy, "Fort-NoCs: Mitigating the threat of a compromised NoC," in *Proceedings of the 51st Annual Design Automation Conference*, 2014, pp. 1–6.

[46] S. Pitaka, "Isolation Design Flow for Xilinx 7 Series FPGAs or Zynq-7000 SoCs (Vivado Tools)," AMD, Tech. Rep., 2020, https://docs.amd.com/v/u/en-US/xapp1222-idf-for-7s-or-zynq-vivado.
[47] I. Pinto Gouveia, M. Völp, and P. Esteves-Verissimo, "Behind the last line of defense: Surviving soc faults and intrusions," *Computers & Security*, vol. 123, p. 102920, 2022.
[48] U. Schmid and A. Steininger, "Decentralised fault-tolerant clock pulse generation in VLSI chips," Sep. 7 2010, uS Patent 7,791,394.
[49] J. Sepúlveda, A. Zankl, D. Flórez, and G. Sigl, "Towards protected MPSoC communication for information protection against a malicious NoC," *Procedia computer science*, vol. 108, pp. 1103–1112, 2017.
[50] N. Srivastava and K. Banerjee, "Performance analysis of carbon nanotube interconnects for VLSI applications," in *ICCAD-2005. IEEE/ACM International Conference on Computer-Aided Design*. IEEE, 2005, pp. 383–390.
[51] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, ser. OSDI '99. USA: USENIX Association, 1999, p. 173–186.
[52] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative Byzantine Fault Tolerance," *ACM Trans. Comput. Syst.*, vol. 27, no. 4, jan 2010. [Online]. Available: https://doi.org/10.1145/1658357.1658358
[53] P. Veríssimo, "Uncertainty and predictability: Can they be reconciled?" in *Future Directions in Distributed Computing: Research and Position Papers*. Springer, 2003, pp. 108–113.
[54] Y. C. Yeh, "Triple-triple redundant 777 primary flight computer," in *1996 IEEE Aerospace Applications Conference. Proceedings*, vol. 1. IEEE, 1996, pp. 293–307.
[55] P. Traverse, I. Lacaze, and J. Souyris, "Airbus fly-by-wire: A total approach to dependability," in *Building the Information Society*. Springer, 2004, pp. 191–212.
[56] L. Mancini, "Modular redundancy in a message passing system," *IEEE Transactions on Software Engineering*, no. 1, pp. 79–86, 1986.
[57] H. Kopetz and G. Bauer, "The time-triggered architecture," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 112–126, 2003.
[58] P. E. Veríssimo, "Travelling through Wormholes: A New Look at Distributed Systems Models," *SIGACT News*, vol. 37, no. 1, p. 66–81, mar 2006. [Online]. Available: https://doi.org/10.1145/1122480.1122497
[59] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schröder-Preikschat, and K. Stengel, "CheapBFT: Resource-efficient Byzantine fault tolerance," in *Proceedings of the 7th ACM european conference on Computer Systems*, 2012, pp. 295–308.
[60] G. Pagonis, V. Leon, D. Soudris, and G. Lentaris, "Increasing the Fault Tolerance of COTS FPGAs in Space: SEU Mitigation Techniques on MPSoC," in *International Symposium on Applied Reconfigurable Computing*. Springer, 2023, pp. 215–229.
[61] S. Shreejith, K. Vipin, S. A. Fahmy, and M. Lukasiewycz, "An approach for redundancy in FlexRay networks using FPGA partial reconfiguration," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2013, pp. 721–724.
[62] R. Tabish, "Next-generation safety-critical systems using COTS based homogeneous multi-core processors and heterogeneous MPSoCs," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2021.
[63] S. Alcaide, G. Cabo, F. Bas, P. Benedicte, F. Mazzocchetti, F. Cazorla, and J. Abella, "Unboxing the Sand: on Deploying Safety Measures in the Programmable Logic of COTS MPSoCs," in *11th European Congress Embedded Real Time Systems (ERTS 2022)*, 2022.
[64] G. S. Nicholas, A. S. Siddiqui, S. R. Joseph, G. Williams, and F. Saqib, "A secure boot framework with multi-security features and logic-locking applications for reconfigurable logic," *Journal of Hardware and Systems Security*, vol. 5, no. 3, pp. 260–268, 2021.
[65] F. Turan and I. Verbauwhede, "Trust in FPGA-Accelerated Cloud Computing," *ACM Comput. Surv.*, vol. 53, no. 6, dec 2020. [Online]. Available: https://doi.org/10.1145/3419100
[66] T. Güneysu, I. Markov, and A. Weimerskirch, "Securely sealing multi-FPGA systems," in *International Symposium on Applied Reconfigurable Computing*. Springer, 2012, pp. 276–289.
[67] D. Owen Jr, D. Heeger, C. Chan, W. Che, F. Saqib, M. Areno, and J. Plusquellic, "An autonomous, self-authenticating, and self-contained secure boot process for field-programmable gate arrays," *Cryptography*, vol. 2, no. 3, p. 15, 2018.

## APPENDIX

### A. Safety

In the context of our protocol, proving that Samsara is safe refers to ensuring integrity is preserved.

*a) Bootstrapping:* The Controller, being a simple and easily-verifiable trusted-trustworthy ASIC, cannot be tampered with and follows only the designed logic at all times. During the boot phase, it signals the MP-Boot to start execution of the Bootloader code. This microprocessor is the only software processing unit capable of accessing the PL and its memory and is triggered only by the Controller (*Bootstrapping phase, step 2*). Since no other software is allowed to run on MP-Boot, the configuring API is only called by the Bootloader and Tileloader, which are kept in Tamper-Resistant Storage in encrypted form and authenticated by the Controller. Therefore, the initial configuration of the PL is only executed at boot time as demanded by the Controller and not triggered arbitrarily by malicious code. The same applies to the Tileloader during Rejuvenation. (*Bootstrapping phase, step 3*).

Full and partial bitstreams are stored in encrypted form in the Softcore Library and authenticated when loaded into the PL by modules such as the AES-GCM engine, present in, e.g, Xilinx UltraScale+ devices as part of the Configuration Security Unit (CSU). Without knowledge of the AES-GCM key, a bitstream cannot be modified or forged. Given authenticated bitstreams can still be malicious in nature or suffer faults when deployed into the PL, the necessity for Rejuvenation remains.

Furthermore, techniques such as those presented in [66], [67] utilize the Xilinx Internal Configuration Access Port (ICAP) to read FPGA configuration memory at runtime and generate a hash for comparison against an expected hash with the goal of detecting runtime tampering, which the Bootloader and Tileloader can use for further verification.

Since the Controller, MP-Boot and Softcore Lib are tamperproof and the MPSoC designed to grant only access to the required components, it follows that the Compute Platform and tiles work as defined.

*b) Execution:* During PL-side executions, to ensure message requests are not dropped, the Controller assigns an *unique ID* to each message, which is incremented by a hardware logic counter it implements. This counter is monotonic and used to associate sequence numbers to each operation (*Execution phase, step 2*), similarly to USIG in MinBFT [10] . Given only the Controller has write access to its specific set of BRAM memory (PLM-C) from which the tiles will read requests from, it is guaranteed that requests are originated from the Controller. Additionally, requests are hashed for integrity and verified by the tiles, which, upon execution of the request, place a reply with the same ID in hashed form in their own BRAM memory to which only they have write access, proving its origin. Only replies with the corresponding ID in the corresponding address offset (dictated by the ID) are accepted by the Controller. The fact that messages are hashed means they cannot be forged by the bus or a malicious PL module.

Finally, the Controller matches replies with the same *unique ID* from all tiles, forwarding the result to the invoking application only in case of majority ($2f + 1$ or $f + 1$). Otherwise, Rejuvenation is triggered based on the chosen policy. Given this processing is done by hardware logic alone, the Controller is trusted to provide the correct result to the application (*Execution phase, step 4*).

*c) Rejuvenation:* Rejuvenation works similarly to Bootstrapping, as the Tileloader is triggered by the Controller upon mismatch and uses a similar API to load the partial or full bitstreams into the PL. The Controller uses the parameters defined in Config that dictate the type of softcore/accelerator and version to load, which is stored in the TRS (*Rejuvenation phase, step 2*). Even in the eventuality that a wrong bitstream version was loaded, the execution phase would trigger a mismatch in the Controller, leading to another Rejuvenation. Softcore/accelerator digests are also registered in the Controller TRS to ensure no foreign bitstream is loaded.

Finally, state transfer is guaranteed to be correct since the Controller stores the hashed state during the Execution phase, after successfully matching each request reply, in the PLM-C BRAMs to which only it has write access. The state retains all the previous matching replies delivered, while non-delivered requests should be replayed by the application. After a successful Rejuvenation, tiles read the hashed state from the Controller's PLM-C into their own BRAM so that they can keep building upon it if required by a stateful application. No other tile can forge state in another Tile's BRAM given no read or write access is provided. Furthermore, the Controller issues new Requests only when tiles are loaded and state transfer ends, as signaled by the Tileloader and the tiles themselves. In the event of a full PL Rejuvenation, which wipes the PLM-C BRAMs, the Controller first takes a Checkpoint, saves it to the on-chip SRAM and only then triggers the Tileloader.

### *B. Liveness*

We detail now the informal proof that availability and termination are achieved.

*a) Bootstrapping:* The Bootloader and Tileloader are executed as bare-metal applications. No other application is allowed to execute in the MP-Boot, therefore they will never be preempted. The Controller signals when execution shall take place via an interrupt and MP-Boot's interrupt input port is connected only to the Controller. Similarly, the Config TRS is connected only to the Controller, which is trusted not to flood it, and configurations are sent to the MP-Boot by an exclusive channel connecting only the Controller and the MP-Boot (*Bootstrapping phase, step 3*). Since no other element of the MPSoC has access to these configurations, they shall always be available. The Controller only triggers the Bootloader at boot time, before the PL starts receiving requests; and only signals the MP-Boot to execute the Tileloader after detecting a mismatch in replies. A mismatch is observed after the PL tiles have executed and replied, as such, before sending new requests, the Controller triggers the Tileloader, which reconfigures the PL (partially or fully) while tiles are waiting for a new request (*Bootstrapping phase, step 6*). Reconfiguration is preemptive and will not fail even if the tile is executing some rogue code or finishing logging the request in its local memory (in stateful applications). A corrupted log is also irrelevant as it is never sent to any other tile or the Controller. The latter records its own log, which is used for state transfer and is always available in the PLM-C or the on-chip SRAM.

In order to prevent Rejuvenation from stalling, the Controller sets a timer and waits for Ready responses from the corresponding rejuvenated tiles and the Bootloader, Tileloader and tiles notify the Controller as being ready when Rejuvenation has finished. This ensures normal operation is resumed as soon as Rejuvenation is done and that after a pre-defined time limit, if a tile is not ready, Rejuvenation is triggered again and the tile is swapped for a diverse one.

*b) Execution:* The Controller uses parallelized logic to receive requests and place them in queue while it processes replies from the tiles. Nevertheless, to ensure correct operation, the Controller only starts accepting requests when its status is Ready, meaning requests are not accepted while the PL is Rejuvenating. Jamming the Controller may be attempted by faulty applications, however, due to hardware parallelism, the input ports and request verification logic are independent of the logic used to handle requests (i.e., sending requests to replicas, waiting for their reply, verifying agreement, etc) and trigger Rejuvenation. The Controller can ignore requests sent by an application if they are sent with a time interval less than a determined delta or above a maximum threshold number within a time window.

Additionally, requests, when issued by the Controller, are readily available to the tiles, due to being written in the PLM-C, to which tiles have read access. Each tile has its own read-only access channel to the PLM-C memory controller, granting them exclusive use of the bus and ensuring there is no contention. The same goes for tiles writing replies in their own local BRAMs and the Controller reading them. No replays happen since each message has its own *unique ID* and is placed in a specific address offset according to the said *unique ID* (*Execution phase, step 3*). The Controller sets a timer to wait for responses, as highlighted in step 2 of Section III-B3, to make sure faulty replicas to not delay agreement indefinitely.

The Controller considers an operation successful only if it finds majority matches, which happens since a maximum of $f$ faulty tiles are assumed. Otherwise Rejuvenation will happen if one of the tiles is faulty and another is slow. The Controller always delivers the matched reply to the waiting application (*Execution phase, step 4*).

*c) Rejuvenation:* This is only invoked by the Controller that is trusted either in Reactive or Proactive mode. Thus, the system cannot keep rejuvenating except when the Controller notifies the MP-Boot, when faults happen or on well-defined periods (in case of proactive-mode) (*Rejuvenation phase, step 2*). In the former, *H-Quorum* invokes the Tileloader using the given Config (that is always available and secure) and parameters (full-mode or partial-mode). The same is observed

in Bootstrapping with the exception of state-transfer. This is available since the Controller stores the state in a dedicated BRAM to which tiles only have read access through dedicated channels. Furthermore, state is hashed so that it is copied intact to the tile's BRAM. Finally, tiles signal they are ready when state transfer is done before the Controller's timer is out, in which case it Rejuvenates. After Rejuvenation completes, the Controller's status is set to Ready and the Compute Platform is ready to execute new requests (*Rejuvenation phase, step 5*).