UNIVERSITÉ DU LUXEMBOURG

PhD-FSTM-2022-010
The Faculty of Sciences, Technology and Medicine

DISSERTATION

Defence held on 21/02/2022 in Esch-sur-Alzette

to obtain the degree of

DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG

EN INFORMATIQUE

by

Inês PINTO GOUVEIA
Born on 22nd April 1994 in Lisboa, (Portugal)

ARCHITECTURAL SUPPORT FOR HYPERVISOR-LEVEL INTRUSION TOLERANCE IN MPSOCS

Dissertation defence committee

Dr Marcus Völp, dissertation supervisor
Professor, Université du Luxembourg

Dr António Casimiro, Member
Professor, Faculdade de Ciências da Universidade de Lisboa

Dr Gerhard Fohler, Member
Professor, Technische Universität Kaiserslautern

Dr Peter Y. A. Ryan, Chairman
Professor, Université du Luxembourg

Dr Gilbert Fridgen, Vice Chairman
Professor, Université du Luxembourg

To the greatest mother ever

Acknowledgements

First and foremost I would like to thank my supervisor Prof. Marcus Völp for his unending patience, calmness, sense of humor, support and determination in making sure I did not have a heart attack until I finished the PhD. Thank you also, of course, for all the imparted knowledge on such diverse topics.

My deepest gratitude to my co-supervisor Prof. Paulo Esteves-Veríssimo not only for his scientific and academic guidance, but also for the incredible motivational and emotional support throughout this journey. Some life-changing conversations were truly priceless and I learned way more than what goes in this thesis. I do not have words to thank you enough.

A huge thanks to Prof. António Casimiro for all the encouragement and guidance and for helping me navigate and survive the academic storm. I'm not sure I would have finished this thesis otherwise.

I am also grateful to the whole *CritiX* research group, past and current members, for the daily chats, academic discussions, support and shared laughs. A special thanks to Prof. Jérémie Decouchant, Dr. Rafał Graczyk, Natalie Kirf, Christoph Lambert and my "adoptive mother" Dr. Ivana Vukotic.

I would like to extend my gratitude also to the *Navigators* group at the University of Lisbon for being my academic starting home and inspiring me to pursue an academic career. In particular, thanks to Prof. Alysson Bessani for taking the time to review my work and provide feedback and to Prof. Nuno Neves for the valuable advice. A note of acknowledgement to Prof. José Rufino for helping me believe in myself and in achieving my dreams.

The biggest thanks to my parents, Milú Pinto and Henrique Gouveia, and grandmother Dulce Guedes, who always had my back, believed in me and did everything in their power to make me happy. I cannot express how grateful I am to be this lucky in having an amazing family.

Thanks to my greatest friend Paulo Antunes for always being there in the good and not so good moments. For making me laugh in the darkest times and keeping me company in this doctoral journey and everywhere else.

Many thanks to all my other friends in Portugal for spending so much time with me, even from far away, and lighting the mood every time. Thanks to all of you, I did not dive into this adventure alone: Ricardo Costa, Miguel Falé, João Feio, Jerónimo Oliveira, Diogo Ventura, Ana Monteiro, Catarina Jorge, Joaquim Afonso, Henrique Mendes and Rubén Baldé.

Finally, a word of gratitude to all my friends in Luxembourg for being so welcoming and helping me have a life outside of work. A special thanks to Cristiana Silva, Konstantinos Thoukydidis, Viola Glaser, Giorgio Angioletti, Stefano Amodio, Jessica Bonacina, Dr. Mads Engelund and Jacqueline Poupart.

Declaration

I, Inês Pinto Gouveia, declare that the thesis titled, "Architectural Support for Hypervisor-Level Intrusion Tolerance in MPSoCs" and the work presented therein are my own. I confirm that:

- this work was done wholly or mainly while in candidature for the degree Docteur de l'Université du Luxembourg;
- where any part of this thesis has previously been submitted for a degree or any other qualification at this university or any other institution, this has been clearly stated;
- where I have consulted the published works of others, these are clearly attributed;
- where I have quoted from the works of others, the sources are always given;
- where the work presented in this thesis is based on work done by myself jointly with others, I have clearly outlined what was done by others and what I contributed;
- with the exception of such quotations, this is entively my own work; and
- I have acknowledged all main sources of help.

Signed:

Date:

Abstract

Increasingly, more aspects of our lives rely on the correctness and safety of computing systems, namely in the embedded and cyber-physical (CPS) domains, which directly affect the physical world. While systems have been pushed to their limits of functionality and efficiency, security threats and generic hardware quality have challenged their safety.

Leveraging the enormous modular power, diversity and flexibility of these systems, often deployed in multi-processor systems-on-chip (MPSoC), requires careful orchestration of complex and heterogeneous resources, a task left to low-level software, e.g., hypervisors. In current architectures, this software forms a single point of failure (SPoF) and a worthwhile target for attacks: once compromised, adversaries can gain access to all information and full control over the platform and the environment it controls, for instance by means of privilege escalation and resource allocation. Currently, solutions to protect low-level software often rely on an simpler, underlying trusted layer which is often a SPoF itself and/or exhibits downgraded performance.

Architectural hybridization allows for the introduction of trusted-trustworthy components, which combined with fault and intrusion tolerance (FIT) techniques leveraging replication, are capable of safely handling critical operations, thus eliminating SPoFs. Performing quorum-based consensus on all critical operations, in particular privilege management, ensures no compromised low-level software can single handedly manipulate privilege escalation or resource allocation to negatively affect other system resources by propagating faults or further extend an adversary's control. However, the performance impact of traditional Byzantine fault tolerant state-machine replication (BFT-SMR) protocols is prohibitive in the context of MPSoCs due to the high costs of cryptographic operations and the quantity of messages exchanged. Furthermore, fault isolation, one of the key prerequisites in FIT, presents a complicated challenge to tackle, given the whole system resides within one chip in such platforms.

There is so far no solution completely and efficiently addressing the SPoF issue in critical low-level management software. It is our aim, then, to devise such a solution that, additionally, reaps benefit of the tight-coupled nature of such manycore systems. In this thesis we present two architectures, using trusted-trustworthy mechanisms and consensus protocols, capable of protecting all software layers, specifically at low level, by performing critical operations only when a majority of correct replicas agree to their execution: *iBFT* and *Midir*. Moreover, we discuss ways in which these can be used at application level on the example of replicated applications sharing critical data structures. It then becomes possible to confine software-level faults and some hardware faults to the individual tiles of an MPSoC, converting tiles into fault containment domains, thus, enabling fault isolation and, consequently, making way to high-performance FIT at the lowest level.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Computer systems frequently fail on account of a panoply of reasons, ranging from accidental, sporadic faults to well-mounted and persistent cyber attacks. Furthermore, the ways such systems fail are aplenty [Dav16; LAC16; Lee18; Tsi18; Yus19; Pri19], leading to potentially severe consequences, whether their impact resides solely on the cyber realm or extends to the physical world as well. Given the increasing dependency on information and communication technologies, modern societies are, thus, functionally dependent on the correctness of these systems.

Embedded and, later, cyber-physical systems (CPSs) [KS19], for instance, became present in, or the base of, several environments, from industrial/manufacturing facilities, to financing, power grids and autonomous vehicles such as self-driving cars and unmanned aerial vehicles (UAV). This close relation with the physical environments and, in some cases, human lives, raises these systems' cyber-security requirements to unprecedented standards. Their criticality calls for safety- and security-aware designs, fault and intrusion tolerance (FIT) and resilience mechanisms to reduce or completely mitigate the risk of failure[1].

While sometimes distributed [Fen+18], the aforementioned systems are often incapable of tolerating the failure of individual nodes. Moreover, even though some rely on replication and FIT solutions and can, theoretically, tolerate such failures, they may still pose a severe threat if an individual node does fail or falls under control of an adversary. Take the example of a swarm of drones, where a mission may still be accomplished despite the failure of one unit, but the very same failure can lead the affected drone to clash into the others or any other element of the surrounding area. Additionally, full-node replication costs are often high, whether in terms of resources, by replicating entire systems or subsystems (e.g., replicating a whole Electronic Control Unit (ECU)); power

[1] We shall use the terminology and cause-effect fault→error→failure sequence presented in [ALRL04]

consumption; or area.

Simultaneously, with increased dependency on computing comes higher performance and functional requirements. Systems have been progressively pushed to extremes of efficiency through modularity in platform sharing, firstly through virtualization and lately by leveraging the power growth, functional diversity and adaptation flexibility offered by multi- and manycore platforms, namely multiprocessor systems-on-chips (MPSoCs) [DT01; WJM08; BDM09; Ram11; Fur+12]. Such platforms are made possible by the increasing number of smaller transistors able to be fit into a smaller area (Moore's Law [Moo+65]), translating into a larger quantity of components being consolidated into a single chip. MPSoCs have emerged as, not only traditional multiprocessors integrated on a silicon board together with other common resources, but also as a means to fulfill embedded applications' requirements [WJM08].

The increased complexity, reduced transistor sizes and, often, custom off-the-shelf (COTS) hardware bring, however, a greater likelihood of accidental faults. Hardware components are often exposed to radiation, thermal variations, energy variations and some mechanical exertion [Dö14] and smaller transistor sizes make them more vulnerable to such predicaments. At the same time, more critical application domains lead to greater interest in hijacking, i.e., hacking these systems. It is, then, more crucial than ever to advocate for the implementation of fault and intrusion tolerance and resilience, not only in a distributed system environment as heavily discussed in numerous research works [Lam98; CL99; Kap+12; Ver+13; CSK07; Yin+19], but locally in each node, or chip, (i) decreasing both the risk of failure and, consequently, the possible compromise of an entire system; and (ii) reducing replication costs. As we shall describe, taking strategic advantage of the tight coupling of MPSoCs will enable accelerated fault tolerance mechanisms, providing adequate performance for such environments.

Naturally, fault prevention greatly helps reducing the probability of a compromise by, for instance, reducing the attack surface of a system, however, as we shall see, vulnerabilities are hardly ever fully eliminated. As such, a system must be designed to survive even if it gets (partially) compromised.

Solutions have been presented aiming to devise means for fault tolerance in MPSoCs at different levels of the software stack, targeting the operating system (OS), kernel and hypervisor layers [BS95; Bau+09; Dö14; ECP18], i.e., the low-level software layers that support the system - the last line of defense. However, all these solutions were devised under the assumption of a trusted low-level kernel (e.g., hypervisor or platform manager), which, as we shall argue, is a single point of failure (SPoF) itself. Even formally verified kernels, e.g., seL4 [Kle+09a]), may fail due to model/reality discrepancies or hardware faults violating modeling assumptions [BLH18a]. In addition, solutions aiming to conduct safety verification [HHWT97; Fre+11; CAS13] of CPSs tend to have intensive computation costs.

So far, no solution fully eliminates software SPoFs, namely in critical, low-level

management software, meaning they cannot withstand partially successful attacks. Any layer then, even if trusted by the layers above, represents a SPoF, which is detrimental for the system's continued correct operation. If these SPoFs are compromised by adversaries, the latter gain full authority over the platform's privilege-enforcement mechanisms and, through them and resource allocation, access to all information and complete control over all platform resources (e.g., cloud-based systems), including, in the case of cyber-physical systems, extended control over the physical environments on which they act, e.g., nuclear power plants [Das19], power grid stations [Mes07] or contemporary and autonomous cars [Gre15]. It is then in our best interest to eliminate system management SPoFs.

1.1.1 Motivating Example

Consider an autonomous car's collision avoidance system where a replicated collision avoidance application votes, in a triple modular redundancy fashion, to use the car's breaks when the sensors detect an object. Triple modular redundancy (TMR) consists of three replicas executing in lock-step, with the same state. Each replicas "votes" on a result and the majority is applied. If one fails, we can determine which one is wrong, meaning that TMR provides fault masking and diagnosis of up to one fault. So, in this case, even if a minority, one replica, fails, a majority of healthy collision avoidance replicas will apply the breaks by agreeing on the correct value. However, if it is the operating system or overall platform manager that is compromised, then it is possible to exploit vulnerabilities in this layer to access resources such as the breaks and prevent them from actuating independently of the application's decision. Figure 1.1 showcases this example.

This ability to influence resources once the low-level software is compromised is due to privilege enforcement being managed by these layers. Additionally, access control depends on privileges and resource allocation is as well managed by the low-level layers, typically the kernel. So any vulnerabilities in this software gives way for the adversary to take full control. Finally, the lack of proper isolation among system components, which is dependent on access control, makes an fault propagation easier.

1.2 Thesis Purpose and Approach

The common solution to address low-level SPoFs and protect systems' last line of defense has been to introduce a trusted underlying layer, with reduced functionality, e.g., a micro-kernel or micro-hypervisor. However, this new layer suffers from the same issues, as the base mechanisms responsible for isolation, access control and resource allocation, which must be trusted, are still present. So, in turn, this layer becomes the SPoF. Solutions aiming to protect privilege enforcement and other critical operations [NW74;

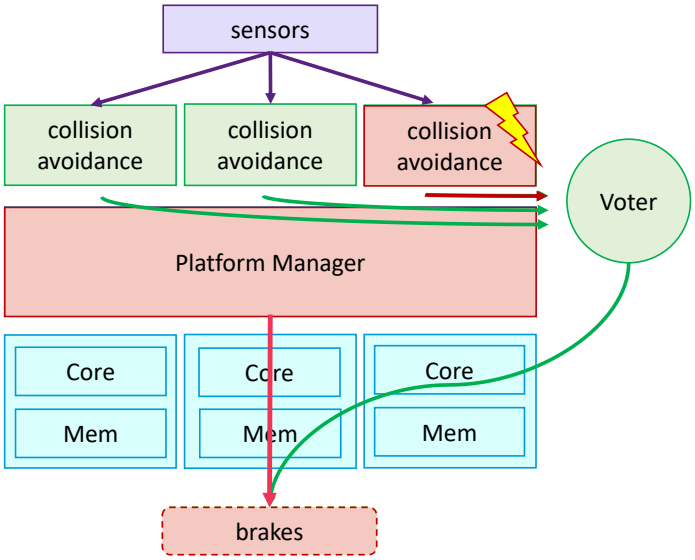


Figure 1.1: Collision avoidance system.

Cha+95; KSRL10; Asm+16] have been implemented, but again run into the same pitfall where the protection mechanism itself becomes the SPoF. As we shall see in Chapter 2, adversaries can exploit vulnerabilities to "taint" whichever resources they desire by attacking system management software, for instance, by reaching and modifying page tables [XCP15], even in trusted enclaves [Fer+17] such as Intel SGX [VB+17].

Fault tolerance has been widely implemented by means of some sort of replication, with TMR being a well-known technique, however, TMR does not benefit from isolation nor from more fine grained forms of agreement. Reaping inspiration from the classical works on Byzantine fault tolerant state-machine replication (BFT-SMR), we can bring MPSoCs to the level of fault and intrusion tolerance and resilience available in current distributed systems and, therefore, prevent any self-contained individual system from failing completely, even if that node is then connected in a distributed network. Performing critical operations in a consensual manner, carried on only if a majority of safely isolated replicas agree to its execution, as in BFT-SMR, while ensuring replica isolation, provides the bases for a system architecture capable of securing the last line of defense.

The perfect example of a critical operation is the ability of replicas to change their own privileges (e.g., access to a resource), which cannot be allowed to be performed single handedly, without the others' consent, at risk of breaking the isolation principle. Faults must be contained to the originating replica, in order to prevent compromised replicas from affecting the others and any critical resources.

The classical BFT solutions are normally based on replication and the masking of a minority of faulty replicas behind a majority of healthy ones working in consensus. As long as a threshold f of faults, whether accidental or intentional, is not exceeded, the system remains operational, as a quorum [MR98] of nodes agrees to modify the state (data) correctly. Interestingly, MPSoCs and classical distributed systems have parallel characteristics. The former can be organized in the abstract concept of tiles [Wai+97], containing a range of resources (e.g., cores, caches) and, thus, closely resembling the notion of nodes in the latter. Thus, instead of achieving fault tolerance by replicating the whole system or MPSoC, one can instead utilize different tiles as fault containment domains and the cores readily available in the chip as replicas for the low-level software we wish to protect. These software replicas can then be augmented with trusted-trustworthy hardware devices, as well replicated, which handle the execution of critical operations, but are themselves trusted not to fail.

The direct translation of distributed BFT protocols and mechanisms to a tightly-coupled environment such as MPSoCs is, however, far from straightforward. The mere replication of low-level software across tiles does not provide the core principle required for FIT to work in the first place: fault containment and isolation. Furthermore, we must achieve this increased level of safety while freeing the system from (i) having to trust any underlying software layer and from (ii) suffering from the performance penalty FIT techniques would have in a tightly-coupled setting due to cryptographic operations and high message transit.

By building new MPSoC architectures that tackle the challenges presented above, it is possible to secure any software layer with minimal overhead, achieving a performance in the order of microseconds, as demanded by a self-contained silicon platform. For the purpose of this thesis, we constructed a minimal functionality micro-hypervisor in two distinct new architectures and showed, on the example of consensual data structures, how critical applications benefit from the presented trusted extensions and the tight coupling of tiles. To develop such new FIT MPSoC architectures that respect fault isolation, we designed low-complexity trusted-trustworthy units that act as the containment edge for each replica. These units are simple to the point of executing no code and instead being the secure means through which access control, voting and platform reconfiguration are handled. Then, high-performing consensus protocols are run among the low-level software replicas, making use of said units, and ensuring FIT properties.

Nonetheless, considering the long-term running period of most systems, FIT is not enough. Eventually, the fault threshold shall be surpassed given enough time or a powerful enough adversary, turning the MPSoC into a liability at some point in time. Ergo, reliability must be considered. The rejuvenation and relocation of faulty replicas has long been implemented in the distributed systems realm [SNV06; Sou+10], returning previously compromised replicas to a fresh, correct state. It is then our intent to, not only protect the systems' last line of defense, but to protect it throughout its entire life-

time by applying these techniques to rejuvenate tiles and relocate their software from permanently damaged ones to spares.

The working hypothesis of this thesis and, thus, its main contribution is, then, as follows. Through low-complexity trusted-trustworthy units, it is possible to confine software-level faults and some hardware faults to the tiles of an MPSoC system, turning tiles into fault containment domains (akin to nodes) and enabling fault and intrusion tolerance and resilience mechanisms at the lowest level. The units' low complexity thereby opens pathways to easily bring these necessarily trusted components close to a zero defect target, easing verification, if necessary. This introduction of independent trusted units allows for retaining the flexibility MPSoCs provide in terms of resource allocation, dynamic re-allocation and recovery both through rejuvenation from transient faults and relocation from permanently ones. We, thus, gain the advantage of a dynamic system with hardened safety and security guarantees akin to those observed in classical distributed systems.

Therefore, in this thesis, we:

- evaluate how low-level software can fail, placing the system contained within the MPSoC (and possibly others) at risk;
- formulate how we can apply concepts from distributed Byzantine fault and intrusion tolerance together with access control mechanisms to devise a system architecture and protocols capable of eliminating all SPoFs throughout the software stack, focusing on the example of an FIT micro-hypervisor;
- develop ways to reap benefit from the tight coupling of such manycore chips to accelerate these solutions; and
- consider how the proposed architectures and devices can benefit applications, providing the example of critical data structure-sharing among replicated subsystems.

1.3 Thesis Scope

The approaches presented in this thesis target, as mentioned, MPSoCs as well as other sorts of manycore-based systems. Such concepts are often present in embedded and/or cyber-physical systems, often critical in totality or partially [Cas+14; CGR17]. As the nature of MPSoC indicates, we have in mind systems containing all or most of their components on a single chip, with a chip-level fast network interconnecting them.

Performance: Naturally, directly translating state-of-the-art distributed FIT protocols to such systems would result in a prohibitive performance that would be more than suitable for nodes distributed across a traditional network, but severely high in our context. This means we require solutions that perform at the level of a few hundred or

thousand processor cycles and exhibit processing times in the order of nanoseconds or microseconds.

Fault Types: As shall be further detailed in the system 3.4 and fault model 4.2, 5.2 Sections, we consider not only accidental faults, but also Byzantine malicious ones. These faults can then result in a crash or random (Byzantine) behaviour, but such effects shall remain confined to the originating tile, i.e., to the fault containment domain where the fault occurred.

Intrusion Tolerance: Focusing on the FIT mechanisms for the protection of low-level software layers, their performance and scalability, we evaluate resilience by assuming up to f replicas fail arbitrarily and systematically analyze how they may jeopardize the system. As shall be demonstrated when we present the experimental results in Sections 4.6 and 5.6, the system performs just as well even when a minority of replicas are faulty or late.

The frequency and seriousness of hypervisor-, kernel- and RTOS-level faults has been detailed in multiple works [ABWER21; BCP12; PE12; TS13; TN16].

1.4 Overview

This section provides a synopsis for each of the remaining chapters of this thesis, structured as follows:

Chapter 2 presents background information pertaining to the vulnerability of low-level software, the affected mechanisms and how it can give adversaries advantage and control over the whole system. It also discusses MPSoC safety and security and the relevant related work both in low-level software protection and classical Byzantine fault tolerance.

Chapter 3 describes how to construct SPoF-free systems in terms of their replication, fault containment and FIT requirements. We explain how to bridge the gap from current generic MPSoCs to distributed MPSoCs (D-MPSoCs), providing the basis on which to build the envisioned systems, capable of addressing the SPoF problem. We also discuss how to achieve consensus without cryptography for chip-level performance, executing consensus in an equivocation-free manner. Finally, we briefly introduce the two solutions proposed in this thesis and present their system model.

Chapter 4 introduces the *Midir* architecture, an enhanced manycore architecture, effecting a paradigm shift from SoCs to distributed SoCs. *Midir* changes the way platform resources are controlled, by retrofitting tile-based fault containment through well known access-control mechanisms, while securing low-overhead quorum-based consensus on all critical operations, in particular privilege management and, thus, management of containment domains.

Chapter 5 presents *iBFT*, the first hybrid, *Introspection*-based BFT-SMR protocol, crafted for systems where consensus performance should remain close to the speed of

the replica-connecting communication mechanism, i.e., the chip network. *iBFT*'s main concern is acceleration of consensus at low-level, however, it can as well be used to get the system safety achieved by *Midir*.

Chapter 6 discusses resilience for continued unattended operation through rejuvenation and relocation of faulty replicas within the MPSoC.

Chapter 7 applies the given solutions to an application-level use where multiple replicated application subsystems interact by sharing critical data structures.

Chapter 8 finalizes with the conclusions and directions for future work.

1.5 Publications

The list of publications related to this research is the following:

- **Behind the Last Line of Defense: Surviving SoC Faults and Intrusions**
  Inês Pinto Gouveia, Marcus Völp, Paulo Esteves-Verissimo
  Submitted to Elsevier's Computers & Security journal.
- **Behind the Last Line of Defense: Surviving Microhypervisor Intrusions**
  Inês Pinto Gouveia, Marcus Völp, Paulo Esteves-Verissimo
  Intel whitepaper.
- **Introspection: Look What The Other Replicas Did**
  Inês Pinto Gouveia, Marcus Völp, Rafal Graczyk, Paulo Esteves-Verissimo
  Submitted to EuroSys 2022.
- **Introducing Attack Tolerance for Modern Systems on Chip**
  Marcus Völp, Inês Pinto Gouveia, Paulo Esteves-Verissimo
  Intel whitepaper.
- **To verify or tolerate, that's the question**
  Inês Pinto Gouveia, Mouhammad Sakr, Rafal Graczyk, Marcus Völp
  Accepted in PAVeTrust Workshop.

Chapter 2

Background and Related Work

The organization of complex computing resources such as MPSoCs depends on low-level platform management hardware, for instance, memory-management units (MMUs); and software, e.g., firmware, hypervisors and management engines (MEs). However, current MPSoC architectures are such that these management components, which should form a last line of defense against severe accidental faults or adversaries intruding the system (malicious faults), instead constitute a single point of failure, mainly for two main reasons: (i) the way platform privilege-enforcement mechanisms (e.g., MMUs or hardware-enforced capabilities [Woo+14]) are designed allows faults in a core/tile to propagate through MPSoC components; (ii) faults in this lowest-level management software, e.g., hypervisors, configuring these privileges, are bound to propagate across management and managed components, again causing common-mode failure scenarios. This Chapter will be dedicated to the discussion on why these SPoFs exist and why current solutions have not fully addressed the problem of their existence; and to the background explanation of the techniques we will be using to solve it.

We shall provide, in Section 2.1 a description of what privilege escalation is as well as, in Section 2.1.1, an overview of well known access control mechanisms, protecting resources from processes. Next, in Section 2.2, we discuss resource allocation and in Section 2.3 an example of how resource allocation and management is performed within a micro-kernel. Afterwards, in Section 2.4 we discuss the failure risk of low-level software and why it presents a serious concern. MPSoC safety and security is briefly presented in Section 2.5. Then, finally, in Section 2.6, we describe Byzantine fault tolerance and its inner workings, and give an overview of the state-of-the-art.

2.1 Privilege Enforcement

A key piece in system management is, privilege management. In a privilege escalation attack, an adversary can promote the privilege of a process to a higher level by exploit-

ing an OS vulnerability. If the attack succeeds, it is possible to operate the system with a privilege that is higher than the one originally assigned, for instance, avoiding access control and gaining read/write permissions for all system information [Yam+21]. Privilege escalation occurs in two forms: vertically, where a lower privilege user or application accesses functions or content reserved for higher privileged ones; and horizontally, where a normal user accesses functions or content reserved for other normal users. Since a process' privileges are stored in the kernel and cannot be referenced by user applications, they cannot be directly tampered with at application level. However, attacks that issue system calls in a sophisticated manner and exploit vulnerabilities in the kernel space can tamper with process privileges stored in the kernel.

In [Yam+21], an investigation was made on memory corruption vulnerabilities that can allow privilege escalation, as reported on the JVN iPedia Vulnerability Countermeasure Information Database [Jvn]. It was concluded that approximately 89% of the vulnerabilities reported to be capable of privilege escalation were classified as "Critical" or "High" in the CVSSv3 score. CWE-269, for instance, details improper privilege management, with multiple security vulnerabilities associated, such as CVE-2021-42109, CVE-2021-41387 and CVE-2021-40489; and works such as [Sca11] and [Iqb+16] discuss hypervisor-related privilege escalation attacks, namely how a security breach in one hypervisor may break down the whole hypervisor and, consequently, influence the system's guest OSs.

2.1.1 Access Control

Access control is a security technique that regulates which users or system processes are granted access to objects, i.e., which privileges they are given. Multiple techniques have been developed to perform access control, making sure entities have the right privileges to access a certain resource.

Access Control Lists: An ACL represents the permissions attached to an object that specifies which users or processes are granted access to that object, as well as the operations that can be performed on them [Smi19].

Access Control Matrix: An access matrix can be envisioned as a rectangular array of cells, with one row per subject and one column per object. The entry in a cell - that is, the entry for a particular subject-object pair - indicates the access mode that the subject is permitted to exercise on the object. Each column is equivalent to an ACL for the object and each row is equivalent to an access profile for the subject [Lam74].

Capabilities: Capability-based security is based on unforgeable tokens of authority and refers to a value that references an object along with an associated set of access rights, e.g., read/write. A capability is usually implemented as a privileged data structure that consists of a section that specifies access rights and a section that uniquely identifies the object to be accessed, and are generally stored by the OS in a list, with some mechanism in place to prevent a user process from directly modifying the con-

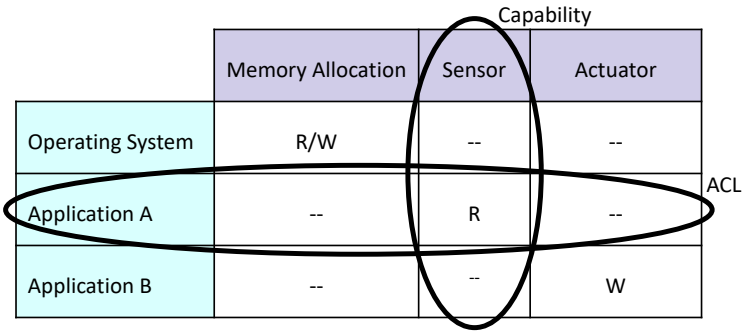


Figure 2.1: ACL - capability relationship.

tents of the capability in order to forge access rights or change the object it points to. Attempts to access a referenced object must then be validated by the OS, typically by means of an ACL. Capabilities and ACLs are in fact related. Figure 2.1 demonstrates this relationship.

2.2 Resource Allocation

Any process requires resources to be allocated to it in order to run, such as access to a portion of the memory's address space. Resources can, however, be any component internal (physical or virtual) to the system or externally connected to it, for example, cores, memory, peripherals, files, etc. The task of allocating these resources generally falls to the operating system or other low-level software, such as a micro-kernel or hypervisor. Let us take the example of address spaces in a micro-kernel.

At the hardware level, an address space is an amount of memory, i.e., range of addresses, allocated to a process. Most systems use, however, the notion of virtual memory, giving a process the illusion of a larger memory space, in turn making it possible to run several processes concurrently and isolated from each other. The mapping between physical and virtual addresses is manged by the OS or a smaller underlying layer and it is implemented by the translation lookaside buffer (TLB) hardware and page tables. Page tables are data structures used by the core to store the virtual to physical address mappings and the TLB consists on a cache containing a subset of the page table and used for recently used translations. The MMU traverses page tables to replace translations in case no mapping is present for the currently accessed page.

2.3 A Micro-Kernel Example

A micro-kernel is the close-to-minimum low-level software providing the mechanisms needed to implement an OS. It provides minimal services of process and memory management and inter-process communication (IPC). It is often the software operating at the most privileged level and, thus, controls critical resources and possesses powerful mechanisms, namely for privilege escalation and resource allocation, that must not fall into the hands of adversaries. A micro-kernel hides the physical hardware concept of address spaces, since otherwise, implementing protection among processes would be impossible [Lie95]. For constructing and maintaining address spaces, the micro-kernel often provides two operations: *grant* and *take*.

Address space operations: The owner of an address space can *grant* any of its pages to another space, provided the recipient agrees. The granted page is removed from the granter's address space and included into the grantee's address space. Similarly, one can make it so that the page can be accessed in both address spaces. Finally, a page can be *taken*, maintaining it accessible in the taker's address space, but removing it from all other address spaces which had received the page directly or indirectly. These are clear examples of critical operations that, once the micro-kernel is compromised, give way for an attacker to control other (critical) resources in the system.

Issues: Even if the micro-kernel was to be replicated, since MMUs implement virtual-to-physical address translation by traversing OS-managed page tables and given that most systems today perform no further checks once resource accesses pass the MMU, compromised replicas (i) installing valid translations or (ii) bitflips in page tables may, therefore, violate fault containment and cause faults in one kernel replica to bring down other replicas. Replication of data structures or consequent application of error-correcting codes (ECC) and memory scrubbing for non-replicated data structures prevents the latter (ii), however, the first obstacle (i) remains, even if memory keeps its value despite bitflips.

Exploitation examples: There are several ways page tables can be altered and/or deviated from their purpose by corrupted management software: (a) accessing pages without going through both the first- and second-level page tables can lead to a page table being interpreted as a page, (b) mistaking second level page tables as first level may lead a page to become a page table, and finally (c) page tables can be mapped as writable pages. Figure 2.2 provides a visualization of the three issues just described. An example of TLB exploitation via a privileged user is detailed in CVE-2019-19339 and a Linux vulnerability that allows access to a physical page after it has been released back to the page allocator and reused is described in CVE-2018-18281. In fact, most reported OS kernel vulnerabilities are due to improper memory management [SPWS13; CDA14].

The need for protection of the means for privilege escalation and, therefore, resource access thus emerges from their frailty. Gaining the means to manipulate address spaces

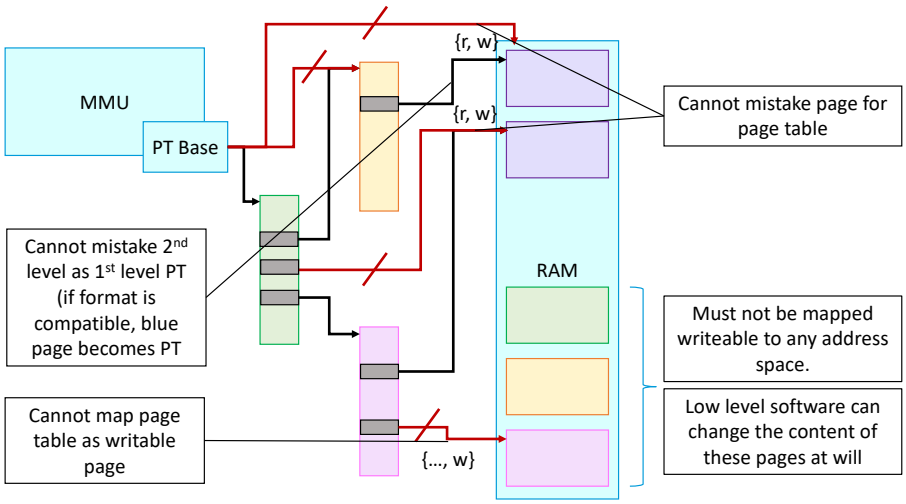


Figure 2.2: Installing malicious valid translations.

at will gives an adversary means to sway the systems to their own profit.

2.4 The Failure Risk of Low-Level Software

On the hardware side, existing mechanisms such as the MMU and MPU (memory protection unit) play an important role in providing memory protection, mainly to prevent a process from accessing memory that has not been allocated to it. However, as we have seen in previous sections, these are not enough for isolation.

On the software side, conventional OS designs equate control over resources with the possibility of directly or indirectly obtaining access to them. For example, by manipulating page tables as mentioned above, an OS kernel can install virtual-to-physical address mappings to any resource that resides in the platform's memory map and, once installed, it may access this resource through this very same mapping. Of course, one can differentiate application and kernel-level access, confining the latter to a boot-time fixed partition. Nonetheless, even restricting the system resources that are available to the kernel, a compromised instance would still be able to indirectly access said resource by mapping it to an application it controls.

Traditionally, the pattern to solve this issue has been to introduce an underlying software layer to perform these resource management decisions on behalf of the virtualized kernel. Examples include hypervisors (and micro-hypervisors), virtualizing guest OSs, but also Intel SGX, preempting management OS access to enclave memory and ensuring data is encrypted and signed before it is returned to this OS. Unfortunately, this layer now becomes the single point of failure and target for attacks.

2.4.1 Is this a real risk?

It is, if the vulnerability rate of these low-level platforms is non-negligible. Recent problems, whether in Intel's CSME [EG17], Xen/Critix [Xen] or concerning Spectre [Koc+18] and Meltdown [Lip+18], have been repeatedly reminding us of how brittle the assumption of "tamperproof and unattackable low-level platform management assets" is.

Numerous vulnerabilities have been reported in real-time operating systems' (RTOSs) source code, namely in IoT devices (e.g., CWE-119, CWE-120, CWE-126, CWE-134, CWE-398, CWE-561, CWE-563) [ABWER21]. Vulnerability analysis of virtualized environments and hypervisor security have shown the various ways these can be attacked [BCP12; PE12; TS13; TN16; ZD20]. As previously mentioned, works such as [Sca11] and [Iqb+16] have already discussed privilege escalation attacks in hypervisors for full compromise. Even micro-hypervisors are vulnerable to attacks. While their assigned features are reduced which, consequently, means fewer lines of code and, perhaps, reduced complexity for formal verification; this also means slimmer or minimal security features, making micro-hypervisors relatively easy to gain access to, as remarked by [TS13].

Furthermore, the vulnerability analysis performed in [BCP12] supports the claim that protecting one system node (e.g., in a network) is essential in preventing other nodes from being tampered with. Attackers can exploit vulnerabilities in virtualization environments to compromise one or more of its software processes, but also exploit a defect in the relationship between two components. Exploiting a defective component to establish a trusted path to a second component, it is possible to compromise the availability of the latter by setting up a malicious alternate component with the same internet protocol (IP) address setting the stage for a man-in-the-middle attack. Even though attacks across a traditional network, in classical distributed systems, is out of the scope and not the purpose of this thesis, this further endorses the need for the protection of individual systems to protect themselves, but also other nodes, e.g., in a CPSoS.

Even formally verified software (e.g., kernels such as seL4 [Kle+09b]), although mathematically proved to be "flawless", may fail due to model/reality discrepancies or hardware faults violating modeling assumptions [BLH18b]. Namely, errors can still be found in software checked by a theorem prover [FZWK17], in particular if the assumptions they are based on are over-simplified. Additionally, the considerable effort that goes into proving new or existing pieces of software questions practicality. For instance, seL4's kernel implementation, correctness proof and binary verification took a total of 24.7 person years to complete.[1]

[1] Of course, the collaboration between fault tolerance and formal verification results in the best combination for the reliability of the system. What we argue here is that, aside from the practicality of verifying even a small micro-kernel, formal verification alone may not be sufficient, namely due to possibly needed simplifications and approximations.

It is a risk because, in essence, compromising low-level software can give attackers (i) access to information; (ii) access to privilege management, resulting in the ability to escalate its own privileges; (iii) ways to perform critical operations; (iv) access to other components of the system or other systems; and (v) the ability to inject, modify and execute code.

2.4.2 Being the risk real, are there no solutions yet?

The solution design space for contemporary hardware platforms' (namely MPSoCs) dependability and security has been unfolding in two directions: (i) application-specific system-level replication (e.g., triple modular redundancy, mainly in CPSs, by means of multiple ECUs), where the lack of flexibility limits the extension to general systems; (ii) manycore-level replica management and consolidation, which then, if on bare MPSoCs, reintroduces the SPoF concern, now for the low-level replication management component.

Moreover, solutions attempting to secure low-level software layers tend to rely on a smaller, but still SPoF, underlying layer. Mitigation measures have been studied for detection and containment of errors in operating systems and manycore-support software [McC+10; SLQP07; Dö14; SYT16] through this underlying, assumed-trustworthy layer. Minotaur introduces a toolkit to improve the analysis of software vulnerability to hardware errors by leveraging concepts from software testing [Mah+19]. However, these have a non-negligible complexity and, in consequence, even a residual fault or vulnerability rate in these supposedly trusted components may breach the platform's dependability and security goal.

In fact, as confirmed by [HDL13], "simple" components with at least a few thousand lines of code (KLOCs) have a non-negligible statistical fault footprint. Other studies [OW02; OWB04] reveal between 1–16 bugs per 1,000 lines of code go undetected before deployment, even in well-tested software. Operating-system kernels form no exception [PG05; Mat+14]. Recent insights [Pal+14] reveal that faults in stateful core subsystems — on which we focus here — outrank driver bugs in severity.

Many approaches target operating systems with the goal of improving their resilience against faults. However, typically they protect applications [DS10; BCM13; Kuv+16] or specific OS subsystems [Sun+10; SABL06; Zho+06; ES13] and only from accidental faults. Efforts for providing whole-OS fault tolerance include [Her+06; NB13; DCCC08; LAK09; Bha+16; GTHR99; Gen18]. Nevertheless, the complexity of these recovery kernels is comparable to that of a small hypervisor. For example, OSIRIS [Bha+16] directs OS recovery to a 29 KLOC reliable computing base (RCB) [ED12], roughly twice the size of modern micro-kernels [Lie95; LWHH18; Kle+09a; Asm+16]. Again, this makes the likelihood of residual faults or vulnerabilities non-negligible.

Some systems based on capability-based addressing have hardware support for capabilities. CHERI [Woo+14] complements a page-based virtual memory protection mechanism with conventional MMU-based architectures and hardware-supported descriptions of capabilities. Here, capability addressing occurs before virtual-address translation, such that each process is a self-contained virtual capability system. Since CHERI adds capability protection on top of OS managed page-based protection, it includes the MMU and the OS in each application's reliable computing base (RCB), that is, the set of system components that are assumed trusted not to fail. As we have seen, though, such an assumption is brittle given OSs proneness to vulnerabilities.

Several other works have given early steps in the direction of the solutions we shall advocate for, minimizing the threat surface, or enforcing isolation. Nohype [SKLR11] removes all but a small kernel substrate from application cores, which run functionality-rich OSs in virtual machines (VMs), reducing the threat surface. However, this kernel substrate forms a single point of failure. Cap [NW74] and M3 [Asm+16] exploit hardware capability units, and Hive [Cha+95] and Stanford-Flash's MAGIC a bus-level firewall to isolate VMs at tile granularity. These approaches provide for capability unit/-firewall reconfiguration through the kernel. However, although this avoids trusting local kernel substrates for isolation, their configuration interface, which is necessary to retain flexible resource sharing, turns the configuring kernel into a single point of failure. The Loki architecture [ZKDK08] uses tagged memory to simplify security enforcement by associating security policies with data in physical memory. However, the HiStar OS (and its simplified version LoStar), using Loki, still relies on a security monitor, running underneath the kernel in a special processor privilege mode, to manage tags for tagged memory and protection domains. REBOUND [Gan+21] does not mask faults, but instead aims to return the system to correct behaviour within a bounded time, focusing on recovery, thus allowing the system to go through periods of incorrect behaviour. Additionally, it considers a synchronous system, which although suitable for the CPS targeted, given real-time constraints, is sometimes put to the test as shall be discussed in Section 3.4.

2.4.3 Summary

The realm of low-level software vulnerabilities and attack vectors has been studied and the importance of strongly protecting these last lines of defense has been recognized. Regrettably, although some solutions have been attempted for their safety and security, none so far managed to eliminate the SPoF syndrome that plagues them, with solutions constantly falling under the SPoF themselves. Their merit, however, is relevant and we shall adopt some concepts, such as hardware capability units [NW74; Asm+16] and replication to develop our solution.

2.5 MPSoC Safety and Security

Since MPSoCs are an emerging trend in the embedded and cyber-physical worlds and given our interest in them for their increasing popularity as a platform in the systems we target, it is of interest to discuss their specific safety and security issues already documented in literature.

Research in MPSoCs has been done with special concern on the NoC, which has received significant attention by adversaries due to its connectivity with various components. A NoC can be considered an on-chip version a wide area network and was initially proposed in [DT01; BDM02]. It provides a way of communicating between components in the MPSoC, including features such as routing, flow control, switching, arbitration and buffering.

In [CM21], a thorough survey of NoC attacks and countermeasure is provided. There, it is discussed how a compromised NoC can corrupt data, degrade performance and steal sensitive information; and the most common types of vulnerabilities are analyzed, such as information leakage, denial-of-service and data corruption. Specifically, it analyzes five types of security attacks and, most importantly, their corresponding countermeasures[2]: eavesdropping [ACR14; SZFS17; RP19; CM20b; CM20a], spoofing and data integrity corruption [SK11; KRAT13; YF13; SZFS17; HMGP18], denial-of-service [Was+13; JACR15; BDK16; Sep+18; CLM19], buffer overflow/memory extraction [LC10] and side channel attacks [KJJR11; WS12; Rei+16; IHRS19; Guo+19].

The usage of such countermeasures, or a subset combination of those, can help both NoC manufacturers and embedded or CPS designers to better construct platforms capable of resisting common attacks which, in turn, further supports the fault models presented in Sections 4.2 and 5.2. For example, the solutions presented in this thesis control access to memory, but assume access itself is implemented correctly.

Note, however, that a combination of all solutions may not be possible, feasible, needed or even compatible.

2.6 Fault and Intrusion Tolerance

Given our interest in providing fault and intrusion tolerance at low level, we shall discuss well-known replication-based FIT techniques commonly researched in distributed systems in this Section.

Fault tolerance, i.e., constructing a system in such a way that it retains the ability to sustain correct operation despite the presence of faults, has been used for years, commonly in the form of dual (DMR) or triple modular redundancy (TMR) in CPSs to replicate critical control tasks. Modular redundancy refers to the multiplication of

[2]The citations provided correspond to the countermeasures of each type.

system components, providing redundancy should one fail. These 'cloned' components usually work in parallel (often in lockstep) with the same state so as to make sure at least one keeps operating and achieves the intended result. DMR provides robustness to the failure of one component and error detection, it does not provide, however, error correction, that is, which component is correct and which is malfunctioning (diagnosis and masking) cannot be automatically determined as there is no majority. TMR can determine which of the replicated components is in error, given only one fault occurs. Naturally, TMR allows at most one instance to be faulty and provides no isolation among the components. It consists solely of a voting 'circuit' that applies a majority result, as exemplified back in Section 1.1.1.

CPSs replicate critical control tasks in DMR or TMR systems to ensure safety despite accidental faults in an attempt to protect the cyber and physical assets the system is entrusted with. An example of the use of TMR in highly critical systems can be seen in the primary flight computers of Boeing 777's fly-by-wire (FBW) system [Yeh98]. In a similar context, a form of passive redundancy can also be seen in Airbus' dependability-oriented approach to FBW, where "hot spares" are used in case the active computer interrupts its activity [TLS04]. The concept was extended to multi-phase tightly synchronous message-passing protocols still in the CPS domain [Man86; KB03].

Unfortunately, standard replication through DMR or TMR is not sufficient to eliminate SPoFs as it does not provide any form of fault isolation. That is, for as much as we replicate the components we consider to be SPoF, e.g., a micro-hypervisor, without means of fault isolation to prevent propagation to other replicas, one compromised replica can still bring the whole system under the control of an adversary. Thus, replicated low-level software on its own is still a SPoF.

On the other hand, crash (CFT) and Byzantine fault tolerance (BFT) have been heavily studied in distributed systems. Byzantine fault and intrusion tolerant state-machine replication (BFT-SMR) is a generic technique for hardening systems to tolerate arbitrary faults as well as making them highly available. The idea is to replicate a service across several machines so that if one machine fails in any way (e.g., by crashing or being compromised), then the service is still available through the other machines/replicas. Replication techniques such as this one often rely on agreement protocols (commonly called consensus protocols) to ensure that the different replicas are consistent with each other and that one result is achieved for each agreement instance, i.e., for each request.

If we apply resilient BFT, covering not only accidental faults, but also malicious attacks carried out by a perpetrator, at the low-level software tiers without relying on a complex trusted underlying infrastructure and without applying costly mechanisms that significantly slow down execution, then we can have fault-tolerant MPSoCs whose last line of defense is no longer a SPoF and perpetually protects the system and ensures its correct operation without jeopardizing execution times.

2.6.1 Byzantine Fault Tolerance

BFT-SMR carries the much stronger promise of automated and unattended resilience while the system is under attack, even after classical intrusion detection and prevention mechanisms have failed. In distributed systems, BFT-SMR has been shown to mask the actions of a minority of compromised replicas behind a healthy majority operating in consensus, with rejuvenation techniques maintaining this majority over extended periods of time. The so-called 'Paxos' [Sch+14], and 'Byzantine' [CL99] (BFT-SMR) classes of protocols promote resilience to threats, accidental and both accidental and malicious, respectively, extending the fault tolerance concept to generic classes of applications, namely in loosely-coupled distributed systems.

Building on Lamport's Byzantine generals problem [LSP82], the seminal PBFT protocol [CL99] masks the actions of a minority of up to f compromised replicas, by reaching a majority voted consensus of $|Q| = 2f + 1$ out of $n = 3f + 1$ replicas using a baseline voting mechanism among the values proposed by a pre-defined number of replicated and fault-independent components, thus achieving both safety and liveness. While safety is always guaranteed despite an asynchronous environment, i.e., clients eventually receive correct replies in accordance to linearizability to all their requests; liveness, i.e., progress is eventuallly made, however, requires eventual bounds on message delays, providing a way around the well-known FLP impossibility result [FLP85].

Given that PBFT operates in an asynchronous environments, i.e., where there are no bounds on message delivery delays, the rationale for $n >= 3f + 1$ replicas comes from the fact that f replicas can be faulty and not responding, and f may be correct but late, and therefore f of those that responded may be faulty.

Byzantine dissemination quorums are, therefore, used [MR98], i.e., requests can only be created by clients and these requests and every message passed among replicas can be authenticated with the utilization of message authentication codes (MAC), the keys of which are changed during recoveries to avoid impersonation if an attacker learns the MAC keys. These quorums hold two fundamental properties: any two quorums have at least one correct replica in common (intersection); and there is always one available quorum (availability).

The algorithm's aims is to reach consensus on the order of client operations to execute. All correct replicas execute the same operations in the same order, i.e., they are deterministic. For that purpose, PBFT uses quorum replication and primary-backup mechanisms in a 3-phase protocol based on atomic multicast. The use of 3-phases (pre-prepare, prepare and commit) allows for the establishment of the total order of requests.

BFT-SMR protocols such as PBFT can not only tolerate accidental faults, but also targeted attacks, with other BFT protocols steming mostly from the seminal PBFT [CL99] and from [CNV04], exhibiting a wide range of fault models with tolerance spanning non-byzantine to byzantine faults.

2.6.1.1 Detailed Explanation on BFT

PBFT is considered the first practical BFT-SMR protocol and is the one of the baseline reference protocols of the BFT-SMR research community. As such, we shall recap its inner workings here, giving an overview of how BFT and consensus work in the realm of distributed systems.

PBFT supports the assumption of asynchronous, unreliable networks where messages can be dropped, altered, delayed, duplicated, or delivered out of order; and tolerates independent network failures. Although one expects more synchrony from on-chip networks, attacks on the timeliness of the system may exhibit in the small the similar network properties than one finds in the large. Multiple works discuss NoC performance degradation as a consequence of denial of service (DoS) attacks [FPS08; FLHZ13; PKCC17; CLM19] and, although countermeasures for NoC DoS attacks exist, as enumerated in Section 2.5, other sources of asynchrony may arise, as shall be presented in Section 3.4. Fault tolerance studies in asynchronous NoCs can be found in [Zha16].

Each replica maintains the service state and implements the service operations. Clients send requests to all replicas and await for $f + 1$, i.e., a majority of matching replies from different replicas. Correct replicas are deterministic and execute the same operations in the same order, thus maintaining an equal state. The $3f + 1$ replicas move through a succession of configurations called views. In each view v, one replica ($p = v \ mod \ |R|$) assumes the role of primary (or leader), while others are backups. The primary coordinates consensus, picking the order in which client requests are executed. Every view is marked by a change in the primary replica through a view change protocol, changing every round (marked by a checkpoint creation) or triggered upon a timeout or suspicion of malicious activity. A majority of honest nodes can then vote on the legitimacy of the current primary and replace it with the next leader, ensuring liveness. Spinning [VCBL09] mitigates performance attacks by changing the primary after every batch of pending requests is accepted for execution.

During normal-case operation, i.e., when the primary is not suspected to be faulty by a majority of replicas, clients send requests to be executed, triggering the execution of a consensus protocol among the service replicas to reach agreement on whether to execute the requests and in which order. Consensus consists on a 3-phase protocol based on atomic multicast. The use of 3-phases (*pre-prepare*, *prepare* and *commit*) allows for the establishment of the total order of requests both within the same view and across views.

Request: To initiate agreement, a client c sends a request $< REQUEST, o, t, c >_{\sigma_c}$ to the primary replica, being as well prepared to broadcast it to all replicas if replies are late or the primary changes. This request specifies the operation to execute o and a timestamp t that orders requests from the same client. Replicas will not re-execute requests with a lower timestamp than the last one processed for this client.

Agreement: Then, during agreement, the protocol goes through the following phases:

- ***Pre-Prepare*:** The current view's primary places pending requests in a total order and initiates agreement by sending a $< PRE-PREPARE, v, n, m >_{\sigma_p}$ message to all the other replicas, where m is the nth executed request. The strictly monotonically increasing and contiguous sequence number n ensures preservation of this order despite message reordering.
- ***Prepare*:** Backup replica i acknowledges the receipt of a pre-prepare message by sending the digest d of the client's request in $< PREPARE, v, n, d, i >_{\sigma_i}$ to all replicas, including the primary.
- ***Commit*:** Replica i acknowledges the reception of $2f$ prepare messages matching a valid pre-prepare by broadcasting $< COMMIT, v, n, d, i >_{\sigma_i}$.

Execution and Reply: Replicas execute client operations after receiving $2f + 1$ matching commits, and after having executed all operations with lower sequence numbers. Once a replica i has executed the operation requested by client c, it then sends $< REPLY, v, t, c, i, r >_{\sigma_i}$ to c, where r is the result of applying the requested operation to the server's state. Client c accepts r if it receives $f + 1$ matching replies from distinct replicas.

In order to ensure client and replica authenticity and message integrity, signatures of the form $< m >_{\sigma_i}$ are applied. A replica accepts a message m only if: (1) m's signature is correct, (2) m's view number matches the current view and (3) the sequence number of m is in the water mark interval. PBFT makes use of checkpoints and water marks to limit the size of all message logs and to prevent replicas from exhausting the sequence number space.

2.6.1.2 Safety and Liveness Properties

Safety: A BFT system must return correct results to client requests as long as at most f replicas are faulty. This essentially means the system must ensure *safety*. In order to behave like a centralized service, i.e., for it to appear as a whole (although replicated) and not as separate entities, the state of non-faulty replicas needs to be kept consistent. For a system relying on active replication, this means that a client request that is executed on one correct replica must also be processed on other correct replicas, and that correct replicas must handle client requests in the same order, as mentioned above.

Liveness: Liveness refers to clients eventually receiving replies to their requests, provided at most f replicas are faulty. As proven by [FLP85], consensus in a fully asynchronous system is impossible if one or more nodes may crash. In consequence, a BFT system can only ensure liveness in a partially synchronous environment [DLS88], meaning that there are upper bounds on communication and processing delays. However, these bounds do not have to be known by the system. A message exchanged

between two correct replicas over an unreliable network will eventually arrive at the receiver, even if after several retransmissions [Yin+03]. In sum, the system must be able to make progress at some point in time.

2.6.2 Differentiated Fault Models

With the transition from single-core to multi- and many-core systems, sufficiently many resources became available to sustain in a single node the replication degree required for these protocols. For example, homogeneous BFT-SMR protocols, such as PBFT, require $n = 3f + 1$ replicas to mask the behavior of up to f compromised replicas behind healthy majorities, operating in consensus. Even some small embedded systems offer 4–8 cores, supporting fault thresholds of $f = 1$ or $f = 2$.

However, the BFT protocols mentioned above are expensive in relation to the number messages exchanged, the required number of replicas $(3f + 1)$ and the task of ensuring these are diverse enough to enforce failure independence. Even with enough space for such n, for higher values of f, scalability is largely impacted in respect to overhead, area usage and safety, respectively.

2.6.2.1 Architectural Hybridization

Architectural hybridization [CNV04; Ver06; CNV12], i.e., the inclusion of trusted-trustworthy components, which follow distinct fault models, allows reducing n to $n = 2f + 1$ [Ver+13]. The base untrusted system, subject to faults, is extended with a simple component, on which a higher amount of trust is placed. This component is often distributed and the only one assumed to be tamperproof in the system. The trusted component executes a simple repertoire of functions, similarly to a trusted platform module (TPM) [ACG15].

Several trusted-trustworthy components have been proposed for the realm of distributed systems. MinBFT's USIG [Ver+13] and CheapBFT's CASH [Kap+12] offer trusted counters, whereas A2M [CSK07] and TrInc [LDLM09] offer a trusted log or hash of it. The data they provide and the entailed semantics (e.g., no two messages with the same counter) can be trusted by receiving replicas if they can validate the signature or keyed hashes with which these tokens are protected. As we shall see, however, cryptography is quite costly for tightly-coupled systems and, as such, should ideally be avoided for our goal.

2.6.2.2 Optimistic Protocols

Optimistic protocols [Dis+11; Kap+12; DCK15] operate through error-free phases with only $n - f$ active replicas. In case errors are detected, a switch protocol activates the remaining f replicas (e.g., by involving crash-only hypervisors [Dis+11]) and transfers

the state passive replicas require to catch up to the progress of the active subset. In our work, we shall significantly simplify such switch protocols, namely for late replicas to catch up, without requiring hypervisor or other kernel support.

2.6.3 BFT Over Shared-Memory

BFT protocols have also considered the use of shared memory, an interesting idea to look at considering our goal of on-chip solutions. Fault-tolerant shared memory objects were studied in [MMRT03], built from sticky bits, to coordinate distributed processes in a Byzantine environment where the object state may become corrupted. Given that processes can overwrite gibberish data, classical objects such as read/write or read-modify-write registers that are writable by all processes are useless in a Byzantine environment. Access control lists, persistent objects, bounds on the numbers of faulty processes and redundancy can be used to overcome this drawback. An improvement over this work was presented in [Alo+05], reducing the number of replicas required from $n \geq (2f + 1)(f + 1)$ to $n \geq 3f + 1$. Another solution applies access control lists to constrain which operations Byzantine processes may invoke on the policy-enforced objects they share [Att02], an approach which [BCSFL09; LLOR14] refined to more fine-grain security policies. None of the above can, however, be applied to our target environment, given [MMRT03] and [Alo+05] concern only the Byzantine nature of data, but not of processes; and [Att02], [BCSFL09] and [LLOR14] cannot survive the presence of faulty low-level software, only application-level faults.

In [AMT93] a model was introduced for benign failures in distributed shared memory and [ACKM06] introduced Byzantine Disk Paxos, an asynchronous memory consensus protocol targeted for distributed systems where nodes connect to $n > 3f$ disks. In Byzantine Disk Paxos, up to f memory objects may respond arbitrarily to accesses, but accessing processes are assumed to exhibit only crash faults. Our goal is approximately the reverse: processes (and their local memories) may fail arbitrarily, but the values produced by them must keep on being available. The memory content itself can be protected with ECCs or similar mechanisms.

2.6.4 Tightly-Coupled Systems

BFT protocols have been used in tightly-coupled systems before. For example, in [BS95] replica coordination support was incorporated into a hypervisor for the first time; the crash fault-tolerant Paxos [Lam98] was implemented as a Linux kernel module in Kernel Paxos [ECP18]; replication support in micro-kernel-based systems was introduced in [Dö14]; and in [DGY14] Barrelfish's two-phase-commit protocol [Bau+09] is replaced with non-blocking consensus to tolerate up to $f = 1$ crash faults. With $n = 2f + 1$ replicas, our solutions can tolerate up to f arbitrary faults without requiring a trusted kernel, but while still providing fault isolation among replicas.

In [Agu+20] RDMA is leveraged in the crash fault-tolerant system Mu to bring SMR performance down to microsecond scale, also for BFT [Agu+19]. Mu relies on changing RDMA write permissions to allow the leader to directly write into follower logs. In a manycore context, however, such permission changes would have to involve the OS to, e.g., manipulate page tables, and could induce significant costs, e.g., through translation lookaside buffer (TLB) flushes.

2.6.5 Resilience

Contrary to what may seem at first sight though, fault-tolerant systems in practice are not able to tolerate an infinite number of faults. For continuous long-running services, the upper bound f poses a problem as the number of faulty replicas is likely to eventually grow beyond any practical maximum number of faults to tolerate. PBFT, as well as other similar protocols, achieve safety and liveness provided fewer than a threshold, f, of nodes are faulty throughout the lifetime of the system, with $n >= 3f + 1$ replicas. However, for continuous unattended support, the ability to sustain tolerance is conquered by means of proactive recovery, i.e., periodically returning replicas to a correct state even if they are not suspected of being faulty. To remain resilient despite persistent attacks, replicas should, however, be rejuvenated proactively and reactively [SNV06; Sou+10], and faster than an adversary can compromise more than f replicas. To ensure the latter, diversification [CMS08; RS10; Gar+11; LHBF14; Gar+14; GBN19] cancels adversarial knowledge on how previously analyzed replicas can be compromised. It was concluded in [SNV06] that $2k$ additional replicas are required to prevent exhaustion failure if up to k replicas are rejuvenated simultaneously. This holds a significant improvement over previous solutions, which tolerated f faults during the whole lifetime of the system.

2.6.6 Conclusion

We must break new grounds and open promising avenues in the applicability and resilience of manycore architectures. For that end, the use of Byzantine fault tolerance together with other sets of mechanisms such as voting, the aforementioned capability registers, and concepts of our own design (which we shall introduce in the next chapter), can help us finally secure systems' last line of defense. Bridging the gap from MPSoCs to distributed MPSoCs (D-MPSoCs) shall give rise to new lines of reasoning, allowing us to see that MPSoCs are, after all, on-chip distributed systems, which, given the proper interplay among the mechanisms enumerated above, can be made secure.

Chapter 3

From MPSoCs to D-MPSoCs

Provided the problem discussion and a background on classical fault tolerance techniques, we now turn to bridging the gap between classical distributed systems and MPSoCs, analyzing how exactly components can take the role of nodes, how distributed protocols can be implemented at low level and how the necessary precondition — fault containment — can be achieved by means of proper isolation mechanisms.

3.1 Gap Analysis

So, what are we missing exactly to transform an MPSoC into a distributed MPSoC (D-MPSoC)? Acknowledging that a manycore can behave as a (closely-coupled) distributed system, allows us to design a set of efficient and low-overhead distributed systems-inspired modular protection and redundancy management mechanisms, e.g., in the fashion of BFT-SMR, for fault and intrusion tolerance.

MPSoCs consolidate in a single chip computing resources that used to reside on multiple chips, with plenty of resources are available for processing with high interconnectivity among them. Shared memory is also becoming increasingly popular in such environments, as evidenced by the AURIX architecture [Tec19] for autonomous driving, turning data sharing among resources easier. Tiles [Wai+97] are placeholders and instantiation points for resources, typically instantiated with cores and private caches, or with slices of shared caches and connected through the NoC with each other and with memory controllers (to reach out to RAM/IO). It is possible as well to cast accelerators, GPUs and FPGAs into the tile abstraction. Figure 3.1 shows an abstract view of an MPSoC with several tiles interconnected by a NoC and their possible contents.

The modularity and networked interconnection of tiles already suggests attributes of a distributed system and has inspired first steps to hardware-enforced fault containment at tile level, as pioneered by Hive [Cha+95] and M3 [Asm+16]. Moreover, tiles favour functional and non-functional diversity since they can host cores from several makers

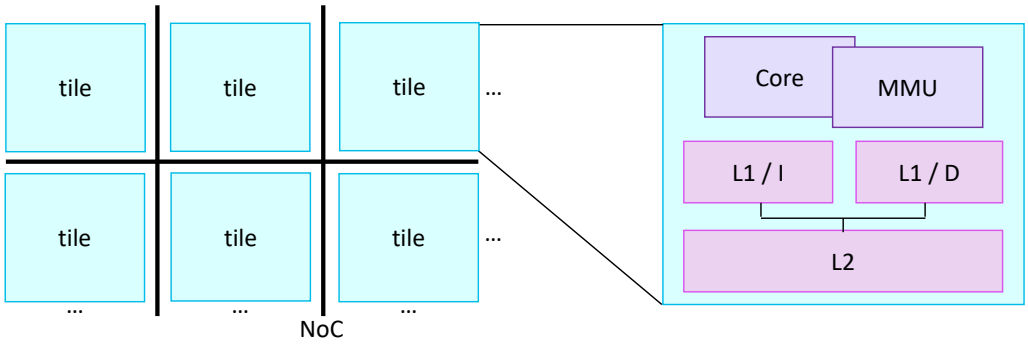


Figure 3.1: MPSoC tiles abstraction, showcasing the possible contents of tiles and how they interconnect.

and can have distinct internal structures (e.g., a different set of components, as long as their ability to host a low-level software replica remains). This improves fault independence through the implied low likelihood of experiencing the same fault in different tiles. Similarly, different versions of the same code can be used at distinct tiles to fulfil the same goal [AC+77; KL86; JA88]. This already provides us the bare bones of a fault tolerant distributed MPSoC: tiles that can host replicas of the low-level software (redundancy), a network interconnecting them and some form of fault isolation (although, for now, incomplete).

As explained in Chapter 2, the issue with Hive and M3, though, is that, although they avoid trusting tile-local kernel substrates for isolation, their configuration interface, which is necessary to retain flexible resource sharing, turns the configuring kernel into a single point of failure. In other words, privilege and access control configuration is still a SPoF, as, say, a hypervisor or micro-kernel replica, operating in one of the tiles, could still reconfigure these at will.

A naive first approach would, then, be to replicate the low-level software we are aiming to protect, e.g., a micro-hypervisor, across different tiles and rely on diversity to justify a fault threshold for compromised replicas. These replicas would then vote on the operations to perform, just like in the aforementioned BFT protocols, and maintain a cohesive, virtually centralized state, as if only one micro-hypervisor instance existed. Unfortunately, even with the replication of low-level software across tiles and the isolation mechanisms provided in Hive and M3, fault containment remains imperfect: potentially faulty or compromised low-level kernels retain control over platform privilege configuration mechanisms and, thus, form a single-point of failure. Consequently, we require simple mechanisms capable of holding the trusted-trustworthy property, that, ideally, run no code and that can guarantee the number one concern in MPSoC fault tolerance: isolation, consensual platform reconfiguration and consensual access to re-

sources. These mechanisms should have a close-to-zero probability of failing and be easily verifiable. Such mechanisms need as well to be replicated, with one provided to each replica much like the USIG in MinBFT.

The modular nature of this solution is crucial not only for its applicability at any layer of the software stack, but also, and most specially, to give latitude to apply an incremental level of protection as desired and as called for given the system's criticality, thus preserving flexibility. We need, then, to investigate novel solutions for the construction of highly-efficient BFT-SMR protocols for tightly-coupled systems, that have performance in the order of micro or nanoseconds to match chip-level latency numbers.

3.1.1 Consensual Updates

Platform reconfiguration (e.g., privilege management or resource allocation) and access to critical resources, including devices, is but reading and writing regions of memory. Privilege information and access policies are stored in some memory addresses that should be protected. It is imperative that updates to these sorts of data structures are done consensually[1] by a trusted-trustworthy mechanism.

Traditionally, locks protect data structures accessed by multiple threads (or replicas) from possible inconsistencies caused by simultaneous updates. Threads are often required to acquire one or more locks to ensure certain operations are executed in a mutually exclusive manner, i.e., in absence of concurrent writes. However, locks do not prevent a faulty replica from corrupting the data being accessed and, most importantly, would not be the correct means with which to protect data as replicas should not access it directly in the first place. As long as a lock is attained, the replica is free to update it at will, and faulty replicas accessing the data could modify it in ways that would not only corrupt it, but also enable privilege escalation, as explained. Not to mention faulty replicas could potentially hold locks indefinitely. In order to protect individual replicas and the system as a whole, shared data must, therefore, be protected from such faulty replicas.

Devising a mechanism capable of performing consensual updates would provide the guarantee that, even if the lowest system layer is compromised, integrity properties are kept for critical data. Consensual updates refer, thus, to voted/agreed-upon updates on shared data, by essentially providing fast enough means of executing on-chip consensus among replicas. From BFT and the notion of majority quorums, voting shall be used to achieve consensus on the critical operations to perform, ensuring a majority of correct replicas agree to their execution.

[1] Consensually means agreed-upon by a majority of replicas.

3.1.2 Equivocation

Sadly, a key factor of BFT, authentication, becomes problematic in the context of MPSoCs. All practical BFT protocols rely on the presence of authentication and, thus, cyrptographic operations in order to ensure replicas do not impersonate others or lie about their votes. PBFT, for instance, relies on digital signatures, requiring that requests and every message passed among replicas are authenticated with the utilization of message authentication codes (MAC), the keys of which are changed during recovery to avoid impersonation if an attacker learns the MAC keys. In MinBFT, the trusted-trustworthy device USIG is in charge of signatures and provides two simple operations `create UI` and `verify UI`. Every message generated by a USIG is tagged with a certificate called UI (unique identifier), containing an ID (the replica's unique identifier), a monotonically increasing counter value and a signed hash of the message; and serves the purpose of uniquely identifying messages. These generated signatures are then verified in other replicas' USIGs.

3.1.2.1 Consensus Without Cryptography

In on-chip environments, however, cryptographic costs, although perfectly acceptable in the context of distributed systems, given their fair performance ratio considering ethernet message passing costs, would not be suitable, since local transfer operations and cross-tile NoC bus costs are in the microsecond to nanosecond domains. Table 3.1 compares these costs for 256 byte and 4KB transfers[2] and HMAC signature generation and verification, a cryptographic operation that, as mentioned, is found at the heart of many BFT-SMR protocols. As it can be seen, HMAC signatures are one order of magnitude more costly than cache-to-cache transfers or tightly-coupled memory accesses (e.g., `memcpy`), even if the read data is compared to a local copy (`memcmp`). As an additional comparison, we also measured the network latency across two machines in the same network.

Close-to-native communication latencies, therefore, require abandoning cryptography and, with this, transferable authentication. Consensus without transferable authentication was first investigated by Lamport et al. in the oral messages (OM) protocol [LSP82]. They identified an impossibility to diagnose errors and hence recover from situations where replicas continue to lie inconsistently to others (i.e., equivocate) in protocols where messages lack authenticators while replicas remain in control of delivered information. In essence, replicas lose the ability to prove the origin of messages once this message leaves the originator's state. There is, thus, the following property.

Non-repudiation: Without cryptographic primitives, state loses its authenticity once it leaves the original writer's memory, e.g., by being copied.

[2] 256 byte sizes correspond to the size of a cache line pair on the x86 architecture and 4KB is the typical ethernet packet size. Additionally, most system calls have parameters with less than 256 bytes.

operation	*256 bytes*	*4096 bytes*
ethernet	153.2 μs (337130 c)	323.9 μs (712698 c)
hmac-sig	1.9 μs (4179 c)	5.2 μs (11362 c)
hmac-ver	1.9 μs (4348 c)	5.2 μs (11492 c)
memcpy	0.1 μs (246 c)	1.1 μs (2331 c)
memcmp	0.4 μs (822 c)	4.9 μs (10680 c)

Table 3.1: Network latency of 256 byte and 4096 byte transfers in relation to local transfers (`memcpy`/`memcmp` using x86' `rep; movsq` `rep; cmpsq` instructions) and to the costs of 256-sha HMAC computation and verification. Measurements are shown in microseconds (μs) and processor cycles (c) of an AMD Ryzen 7 3700X 8-Core CPU (2 threads per core) running at 2.2GHz.

To solve this problem we must rely on architectural hybridization and the introduction of trusted-trustworthy mechanisms, as we shall later discuss in Chapters 4 and 5.

3.1.2.2 Impossibility to Diagnose Faults

Banning cryptographic operations due to their costs leads to an impossibility to diagnose faults as long as replicas remain entitled to change the regions of memory where they write the proposals of their votes. Remember that, in classical distributed protocols, replicas send messages to each other through an ethernet connection. Taking the example of PBFT, once a replica receives a *Pre-Prepare*, *Prepare* or *Commit* message from another with a certain sequence number, it will ignore further messages from the same replica with the same sequence number, forbidding the sending replica from "changing its mind" about the request being voted upon. Inside an MPSoC, however, data must be written in memory, in such a way that replicas can read the votes of others. As such, one must be cautious about an important detail: the time at which a replica reads the memory where the proposals are stored. Additionally, in order to obtain a performance as optimal as possible, reaping benefit of the tight coupling of replicas, one must minimize reads and writes, meaning replicas should be able to just read a memory region whenever they desire, without having to request that information and wait for it to arrive.

Let us assume we have three replicas A, B and C. Replica A is faulty, while the other two are correct. A, being the leader, proposes a request m to be executed next (*Pre-Prepare*). B reads m and, thus, proposes the same (*Prepare*). Then, before C has the change to read A's vote, A changes m to m'. C reads m' and proposes that. When cross checking (*Prepare*), B and C will notice their requests differ. Who is faulty, then? B cannot tell whether it is A or C, and C cannot tell if it is A or B. More precisely, replicas can rely on the authenticity of the information they directly observe from the original

writer (sender), since only a single writer must have access to their own state, but not on information that replicas relay in their state (after having received this information from other replicas).

Specifically, the impossibility applies because faulty replicas may change the information stored in memory before, while or after it is read and, without synchronizing writes with reading operations, they may do so to deliver some information to one group of replicas, while conveying different information to others. It is, therefore, impossible to distinguish a scenario where the sender of a message falsely sends (i.e., writes) some information from one where the receiver (i.e., reader) modifies it.

As such, we must circumvent the above impossibility by not relying on fault diagnosis for reaching agreement.

3.2 D-MPSoC Fault Tolerance Requirements

Replication: Replication is the first step towards a SPoF-free system. As the name indicates, the multiplication of the low-level software into redundant instances maintains the availability of data despite failures (up until the threshold f). In order to guarantee proper isolation, replication needs to happen across tiles, meaning the low-level software replicas must be running on distinct tiles each. Replicas can, however, run alongside other pieces of software, such as applications, in the same tile or even in the same core. Since the tile is the fault containment domain, whatever happens inside its boundaries is irrelevant and does not affect other replicas or system components.

Furthermore, replicas must run the same commands in the same order and start from the same initial state, as in classical BFT protocols. Nevertheless, it is not required for them to operate in lockstep. The latter refers to running the same set of operations at the same time in parallel. However, we allow replicas to be late and then catch up. This shall be further discussed in Chapters 4 and 5.

Finally, replication must be applied only to low-level software and to the trusted-trustworthy components we shall introduce. Replication of every system component is not needed and would, in fact, result in already used costly and inefficient techniques that use the replication of the whole system for fault tolerance.

Consensus and Voting: So that replicas can agree on what operations to perform, some form of consensus protocol must be implemented and, as such, a voting mechanism.

Isolation: Consensus is, however, not enough. In order to enforce fault isolation, nodes, i.e., tiles, require (i) privilege enforcement mechanisms to be reconfigurable only through voting, thus ensuring a majority of correct replicas agree to granting different privileges; (ii) some form of access control so that replicas are locked out of certain resources unless it is otherwise consensually agreed upon; and, naturally, (iii) trusted means for the replicas to perform voting. This shall be further discussed in Chapters 4

and 5.

Consensual Critical Updates: Performing updates only if a majority of healthy replicas agree to their execution is the core of fault tolerance. This is especially important for critical operations, be it in application execution or in platform reconfiguration. No low-level software replica shall conduct critical changes by itself, thus preventing the minority of compromised replicas, if any, from single-handedly changing its privileges or obtaining access to critical resources. Consensual privilege reconfiguration is one of the main contributions of this thesis. This shall be further discussed in Chapters 4 and 5.

Persistent Consensus: Consensus must be, to some extent, persistent. Otherwise, the information concerning agreed upon requests would only be available to the agreeing quorum of $f + 1$ replicas and, if faulty replicas are participating, but refuse to execute the request later on, too few correct replicas would have obtained this knowledge to complete said request. Storing agreed upon system calls in a form of log allows lagging replicas to catch up with the requests they missed. This shall be further discussed in Chapters 4 and 5.

Performance and Shared Memory Communication: Given the tightly-coupled nature of an MPSoC, communication must be carried by the proper means. Performance must be in the microsecond to nanosecond mark. As we saw in Section 3.1.2, this means cutting off the cryptographic costs present in most BFT protocols. It also means message exchange and overhead must be cut to a minimum. To this end, replicas should be able to read the information they require without having to request it, be it agreement data from other replicas, the log of executed operations or error records. Hence, the choice of (protected) shared memory is the most fitting.

Equivocation Prevention: With the lack of transferable authentication, usually provided by cryptography, replicas must still not be able to lie to each other by changing their votes at strategic points in time. Instead, other trusted-trustworthy means of protection must be in place. This shall be further discussed in Chapters 4 and 5.

Trusted-Trustworthy Mechanisms: Each node must leverage a trusted-trustworthy component capable of performing the tasks described above: voting, access control and isolation. Meaning, these components must not fall into the SPoF syndrome, being as simple as possible and running no code. The location and nature of these mechanisms will determine performance, hardware integration and the solution's reliable computing base (RCB). This shall be further discussed in Chapters 4 and 5.

3.2.1 Nature of the Presented Solutions

The possible solutions for the problems presented in this thesis are then distinguishable in a number of ways, particularly in terms of how they deal with:

- The type of trusted-trustworthy mechanisms used (see Sections 6.2 and 5.7.7);

- Where to locate the memory that is going to hold the consensus values, which shall not be modified at will by the replicas to prevent equivocation (see Section 6.2);
- How to make sure lagging replicas, i.e., correct replicas that are late, are able to easily and efficiently catch up and how to ensure they get the correct, latest state (see Chapters 4 and 5). This is related to the previous item;
- Simplicity and optimization (see Sections 6.1 and 6.2).

3.3 Solutions

In this thesis, we present two distinct fully-fledged solutions for the problems defined in Chapter 1, while incorporating the requirements enumerated in Section 3.2: *Midir* and *iBFT*. These solutions differ in the ways described in Section 3.2.1 and have different advantages and trade-offs. In order to provide a more comprehensive view of both, we shall dedicate Chapters 4 to 7 to the several aspects of D-MPSoC construction and analyze how these solutions contribute differently to that goal.

3.4 System Model

Target platforms: As described when discussing the motivation for this thesis, tightly-coupled systems are the target of the solutions presented here. We refer to systems where resources are, generally speaking, closed together and highly interconnected, thus also making them quite dependent on each other. Such is the case of most embedded and cyber-physical systems. Namely, we consider tiled manycore systems, integrated entirely on chip, such as MPSoCs.

Network-on-chip: In our system model, we assume thus a fully connected tiled system, where on-chip network components offer the abstraction of a correct network, interconnecting all tiles to one another. Messages sent are eventually delivered, unchanged, to the destination, but possibly only after several retries. This is fair as network coding [OAHY08], multi-tenant [CMDTM] and adaptive routing techniques [Yan+16] increase the coverage of this assumption. We leave, however, coverage of network attacks and their mitigation for future work.

Diversity: We shall further assume tiles are instantiated with heterogeneous processing elements and will hence exhibit a certain level of fault independence through the implied low likelihood of experiencing the same fault in different tiles, i.e., fault independence through diversity.

Tile-internal functionality: Note that, emulating the spacial isolation of distributed system nodes, we are agnostic about the semantics and interplay of tile-internal and/or

core-level components, e.g., MMUs and their virtualization, copy-on-write, memory protection or recovery functionalities.

Chip-wide failures: Conventional multi- and manycore designs retain the possibility of common mode failures in central hardware components (e.g., the clock or power distribution network), which must be addressed differently. Resilient clocks [SS10] mitigate some of these common-mode faults and the recent trend towards interconnected chiplets further improves the physical decoupling of tiles. Once the physical (hardware) effects of a fault are retained to the causing tile and the signals it exhibits to the system, any remaining faults can be contained through trustworthy tile-level privilege enforcement (as we shall latter describe in Section 4.3 for *Midir* and Sections 5.4 and 5.7.7 for *iBFT*).

Synchrony: In the time domain, although manycores might seem the perfect example of a (closely-coupled) synchronous (distributed) system, reality is a bit different, there are several possibilities for instability. For example, (i) excessive resource use raises the temperature and causes thermal managers to throttle the speed of tiles near this hot spot; (ii) interfering access patterns reduce memory bandwidth by evicting cache lines from shared caches; and (iii) NoC-level bursts may cause noticeable and, with unfair arbitration, unbounded message delays. Faulty behavior (accidental or malicious) might further worsen these negative time-domain effects. A strict synchronous model would not reflect reality and thus be proved brittle. In fact, as mentioned in [FS12], modern VLSI (Very Large Scale Integrated) chips can no longer be viewed as monolithic blocks of synchronous hardware given today's deep submicron technology with clock speeds in the GHz levels, with wiring delays dominating transistor switching delays, and electrical signals not being able to traverse the whole chip within a single clock cycle any more. We, instead, rely on a partially-synchronous model [DLS88] and prepare for possible delays (notably by using buffering). Two particularities exist in these closely-coupled environments, in contrast to large-scale distributed systems, which play in our favor: (i) barring delay variations, liveness is normally guaranteed; and (ii) the infrastructure is plastic in terms of timeliness trade-offs. Therefore, as in most contemporary BFT approaches, we consider asynchrony for safety and partial synchrony for liveness. The structure of our protocols is time-free, and as such they remain safe in the presence of delay oscillations, provided that the fault assumptions hold (no more than f tiles get compromised). Then, the protocols inherit whatever synchrony they achieve from the timeliness of the infrastructure they are immersed in: the manycore works with high performance, in execution and communication, exhibiting short and bounded delays during long enough periods of time, but can exhibit significant variations in these bounds. These are fair expectations, considering the nature of these systems.

RCB: Finally, note that *Midir* and *iBFT* represent different possible configurations with different implications on the RCB, which we shall discuss in the following Chapters, namely in Section 6.1.

Chapter 4

Midir

Our first solution to the issues presented thus far is *Midir*.

Midir is an architecture that constrains the connection of all tiles to the network-on-chip (NoC) through simple and self-contained hardware-based trusted-trustworthy components, which we call T2-H2. Exploring the concept of architectural hybridization [Ver06], we consider those components to be ultra-reliable and to not fail arbitrarily (they may crash, deeming the associated replica faulty), while being agnostic about the reliability of individual tiles and their contents, which may be compromised or fail. The assumption is justified by the simplicity of T2-H2, which, in turn, promotes verifiability. The T2-H2s implement the functionality required for fault independence, containment and tolerance mechanisms described in Section 3.2, and ensure platform reconfiguration is done safely, that is, changes in access control, resource allocation and other critical operations cannot be performed unless the right conditions are met within T2-H2. In consequence, tile-internal software or hardware faults are contained in the tile and the objects the tile can access as specified by the trusted component's permissions.

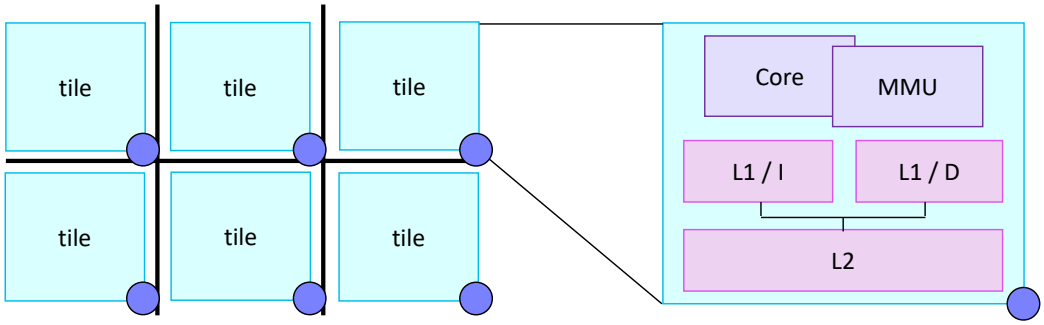


Figure 4.1: Isolation of tiles by means of a trusted component, the T2-H2, placed at the tile-to-NoC interface.

Figure 4.1 shows T2-H2's location between the tile and the NoC interconnect, which not only provides a clear pathway for integration by chip manufacturers and integrators, but also allows drawing from many well-understood building blocks (e.g., region protection, capabilities [NW74], and other chip-level resource management mechanisms [ARJS07], capable of isolating tiles and the resources they can access).

The novelty of *Midir* lies in T2-H2's placement to avoid SPoFs, even while they are reconfigured. These units contain hardware-only logic containing *capability registers* for access control and *voters* for voting on critical operations, namely the reconfiguration of the privileges held in the former.

Low-level software replicas running on the tiles then make use of T2-H2 to access resources, directly (provided the necessary permissions in the capability registers) or through voting, and to vote on critical operations.

Furthermore, the baseline mechanisms for protection and redundancy management provided by T2-H2 can be extended and recursively applied at any software layer, giving the designer ample latitude for crafting resilience into systems, both "horizontally" (incremental power of defense mechanisms) and "vertically" (depth of defense).

4.1 The Midir Architecture

In essence, *Midir* is an architectural concept based on augmenting manycore systems in a minimally intrusive way through strategically placed, simple and self-contained trusted-trustworthy components (T2-H2). In fact, T2-H2 provides just two generic baseline functions staged *in hardware* at the tile-to-NoC interface: access control (capability registers) and quorum-based consensus (voters).

Figure 4.2 depicts one possible layout, of a stereotypical hypervisor-based system, where the hypervisor is replicated for fault/intrusion tolerance, serving virtualized operating systems and applications: hypervisor replicas are distributed across tiles, so that each replica executes on a different tile, separate from applications (although the latter condition does not necessarily need to apply); tiles and software therein interface with each other through the NoC; and T2-H2 are the "blue dots" performing that interconnection. The hypervisor represents an use-case example for *Midir*, however, other sorts of low-level software could be used. Note that replication is only required for the low-level software, which occupy the pre-existing tiles and are equipped with a T2-H2 each.

As long as the execution in a tile remains within the resources associated to this tile (local caches, memories, accelerators, etc.) no overhead occurs, since T2-H2 is not involved in authorizing or denying these accesses. In fact, we remind that it is not the purpose of *Midir* to provide fault containment between software components co-located on *the same tile*, as isolation is done across tiles, deeming tiles the fault containment domain. This resembles the internal behavior of nodes in a distributed system, where nodes are the unit of fault containment.

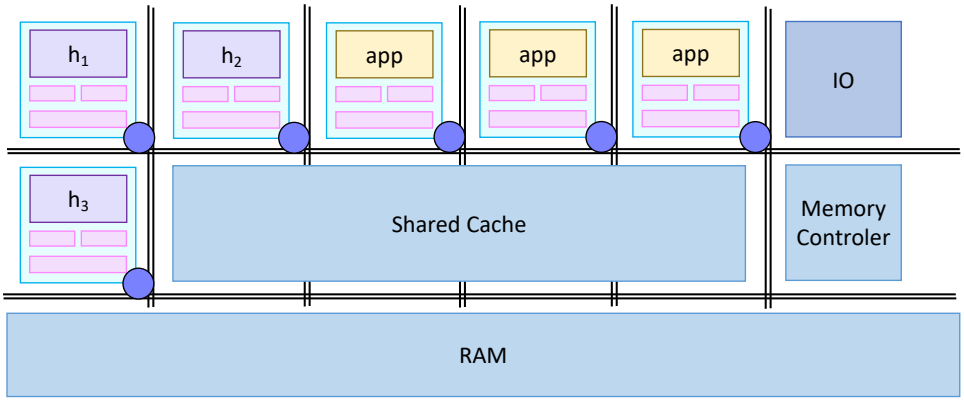


Figure 4.2: Overview of the Midir architecture: a multi-/manycore system augmented with T2-H2 hardware capability units (blue dots) at the NoC interface.

Once software components are spread across tiles, they interact through external operations (e.g., via a resource in another tile, via shared on-chip memories or via external memory or I/O). In this case, T2-H2 interposes such accesses and validates that each of them has sufficient privileges. Consequently, hardware faults inside a tile or accidental or malicious faults in any part of the software it executes are limited in propagation to the objects authorized by these capabilities, which are only modified consensually, as we shall explain later on.

Midir's concept of controlling the tiles' lowest-level privilege enforcement mechanism is agnostic to the mechanism used. However, the simpler such a mechanism and the closer it can be implemented to the tile's NoC interconnect, the more architecture-level faults *Midir* will be able to tolerate.

4.2 Fault Model

Midir's fault model considers software-level compromise at all levels, including in the hypervisor, in the firmware, and, more generally, in any critical software component. This assumption is consistent with our aim of tolerating an incremental level of threat, up to advanced and persistent threats, such as sophisticated attacks mounted by highly skilled and well-equipped adversaries, on tiled manycore systems, often deployed entirely on-chip. Moreover, we consider a limited set of hardware-level faults and attacks: precisely those whose physical effects are confined to a tile (e.g., trapdoors in a core, but no hardware faults that cause a chip-wide collapse).

We strive to establish the tile as a unit of component failure. There is no guaranteed

fault containment inside tiles. That is, adversaries (or accidents) will be capable of compromising the whole software in any tile (e.g., but not only, a hypervisor replica). Once that happens, we no longer make any assumptions about the correctness of any software in that tile, until it is recovered (see Chapter 7). However, we also consider that tiles themselves are fault containment domains.

The system is composed by a set of n low-level-replicas $\mathbb{N} = \{s_0, \ldots, s_{n-1}\}$ executing on different tiles, such that $n = 2f + 1$, following a model based on architectural hybridization. We assume that no more than f tiles are compromised during a reference time T_a. Note that this supports the classical fault and intrusion tolerance fault bound, but also opens the way to promoting resilience [SNV06]. In fact, classical hardening, diversification and intrusion prevention help in putting barriers in the adversaries' way [Gar+14], ensuring that T_a has a usefully large value and shrinks no further.

The generic system components (including low-level platform management ones) can be hardened as needed, down to a residual fault and vulnerability rate. This is good, but not enough, especially under malicious threats. We, thus, leverage architectural hybridization to amplify the coverage of the assumptions made in this threat model, by allowing differentiated strategies towards the fault rate targets across system components. We shall use trusted-trustworthy components, which fall under a more restricted fault model, failing only by crashing, much like USIGs in [Ver+13].

We assume it is unfeasible to construct and/or verify software or hardware of reasonable dimensions, to a 0-defect goal. However, we stipulate that it is possible to design ultra-reliable, ultimately trusted-trustworthy *simple* components to a 0-defect target. The consequence is that these will remain correct and operational, despite compromise of the local tile. As discussed in [Ver06], this is an extremely powerful combination in algorithmic terms: trusted components used routinely to assist critical mechanisms and algorithms (e.g. privilege enforcement, redundancy management) overcoming the residual fault and vulnerability rate of most system components, in order to achieve correct operation with extremely high probability. This is only possible if we strive for absolute simplicity (for verifiability, e.g., by proof assistants) of these trusted-trustworthy components.

4.3 T2-H2

Midir's simple and self-contained hardware-based trusted-trustworthy components are called T2-H2. Exploring the concept of architectural hybridization [Ver06], whilst we consider those components to be ultra-reliable and not fail, we are agnostic about the reliability of individual tiles, which may be compromised or fail. The assumption is justified by the simplicity of the former, promoting verifiability.

Placing the trusted device, T2-H2, at the tile-to-NoC boundary, delimiting the tile as the fault containment domain and ensuring all faults occurring within one tile do

not overflow to other tiles or MPSoC resources, allows us to already achieve some of the desired properties of a D-MPSoC: fault isolation, access control and, given T2-H2's composition, consensual reconfiguration of privileges and consensual execution of critical operations. Figure 4.1 provided a view of this device in the spacial context of the MPSoC. In this Section, we shall discuss this trusted device in detail.

Capability registers: In order to interact with resources outside the tiles and with other replicas, the replica on a tile must first go through this barrier-like component. Each replica has one. Then, first of all, what T2-H2 needs to do is to provide some form of access control. We chose capabilities, which are data structures usually implemented as a privileged data structure that consists of a section that specifies access rights and a section that uniquely identifies the object to be accessed. It shall tell whether or not a process has access to a resource, making it a logical choice. These capabilities allow, then, to access or update a critical resource. Cap [NW74] and M3 [Asm+16] have implemented capabilities in hardware before, albeit with the aforementioned predicaments. These capabilities are placed in the T2-H2 hardware unit in the form of capability registers (small memory units) containing a base address, a range of protection, the permissions the tile has for the resources in that range and the tile's ID.

Capability space: Typically the systems maintain much more capabilities than fit in capability registers. The data structure used for keeping large amounts of capabilities is a capability space, which is typically an OS-maintained array. We shall then use a capability space in memory (RAM), where the list of *granted* capabilities will be stored. Granting a capability assigns that capability to a resource and places it in the capability space, preparing them for later revocation. However, a replica should only be allowed to use capabilities in the trusted component, in the capability registers. As such, the next requirement is to find a way to install capabilities into the registers.

To configure/reconfigure these registers, a capability must be *primed*, meaning copying a capability from the requester's (low-level software replica) capability space into a capability register of its own T2-H2. Once there, it is then ready to be invoked by the corresponding tile. Any reconfiguration of these capabilities, whether modifying an existing one, inserting a new capability or removing, must be a prime operation consensually agreed upon by a majority of replicas. Remember that the capabilities will be giving access to critical resources as well and, as we shall see, to the ability to vote on critical operations; and that they interface with hardware configurations, such as generally critical MMIO like a car's brakes, as exemplified in Section 1.1.1.

Voters: This brings us to consensual updates, of capabilities and other critical data, so that no replica can escalate its privileges single-handedly or modify any critical resources. This means the reconfiguration interface of T2-H2's capabilities must be accessible only through a voter and not invoked directly. Although of course, some accesses to other less critical resources in the system may be allowed to be accessed directly. Thus, the final elements of T2-H2 are voters for consensual updates. Remember that

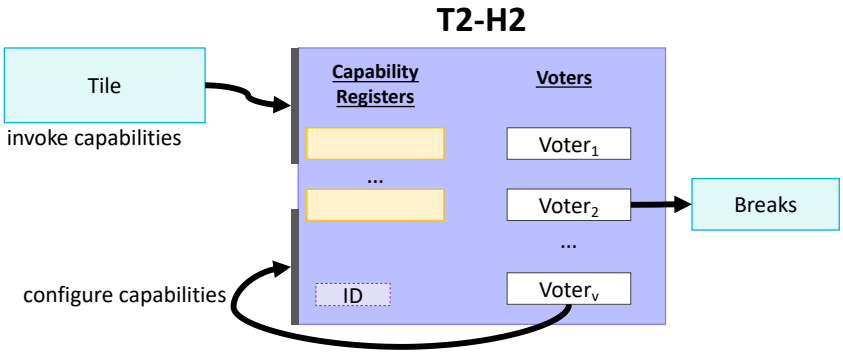


Figure 4.3: Simplified look into T2-H2's contents.

this is all a hardware unit, so nothing in the T2-H2 runs code. Voters must consist of write-once memory locations, where votes are cast and not changed until the voting round is complete or until the voter is reset, this can be accomplished by means of the hardware logic, which ignores modifications to a vote during a voting round. Details on the voter implementations are provided in Section 4.3.3.

Voting can then happen in any tile's T2-H2, depending on the nature of the operation. When tiles invoke the T2-H2, capabilities are checked for access control and votes are issued through the network-on-chip to the proper voter (of course it can also happen that the tile invokes its own voter). Voting is, however, not as easy as simply voting on a final result. We shall discuss the details of reaching agreement on operations and executing them in Section 4.5.

So, in addition to capability checking, *Midir* is capable of subjecting resource accesses to voting by means of distributed components, the T2-H2s, in different tiles. This is especially important for critical operations, be it in application execution or in platform reconfiguration, in order to achieve some form of fault/intrusion tolerance, from error detection or self-checking by comparison, to error masking by consensus. To vote, tiles must hold a capability to the corresponding voter, which authorizes this tile to make proposals as one of these distributed components. Voting is mandatory to install new or change existing capabilities, in order to prevent faulty replicas from bypassing the aforementioned fault containment when reconfiguring the resources a tile can access. Figure 4.3 shows a simplified look on T2-H2's contents, with the example of consensually accessing a car's breaks and configuring capabilities. This figure shows only a single tile invoking a capability. Naturally, to consensually reach a decision within the voters at least $f + 1$ replicas would need to vote, this is just a schematic simplification.

Simplicity also governs our voter design. *Midir*'s voters merely collect and act upon proposals of related operations from different components, letting the voted-upon op-

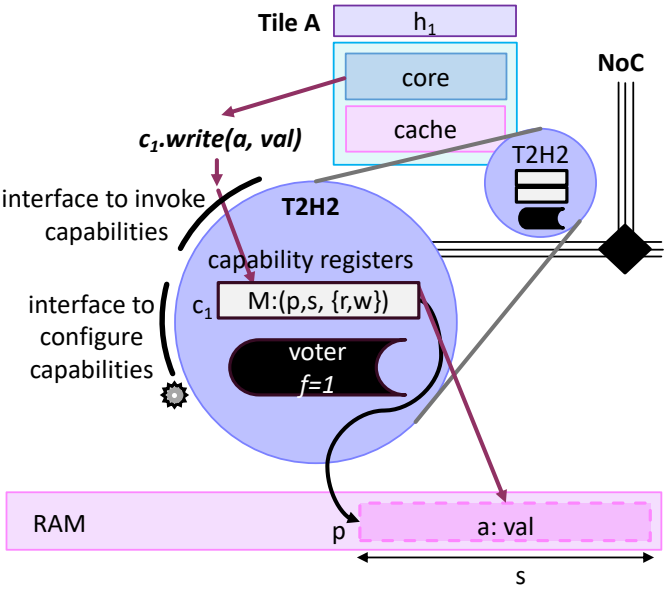


Figure 4.4: Capability-mediated access of tile-external resources. Invoking capability register c_1, application A invokes memory capability $M : (p, s, \{r, w\})$ to write val to location a in region $[p, p + s]$.

eration proceed. Because tile-external resources are typically memory mapped, these operations are normally simple writes. The voters themselves implement no error handling or diagnostics functionality, but provide information for the voting replicas to perform these tasks. More precisely, voters suspend voting on disagreement, freeze the proposals made and expose them for diagnosis. Moreover, they implement a sequence number seq_i for progress tracking, which they increment after each vote unless the vote gets suspended. A voted upon `voter-reset` operation resumes voting and, as well, increments seq_i.

4.3.1 Voted and non-voted operations

To retain the flexibility of the software in a manycore system, allowing it to dynamically adapt resource-to-application mappings as needed, T2-H2 supports direct access to tile-external resources. This way, applications possessing a capability can directly invoke operations on external resources (e.g., to access read-shared or private data in RAM or to interact with non-critical devices). The scenario in Figure 4.4 illustrates a non-voted (write) memory access by Tile A, performed by invoking a capability in this tile's T2-H2. Since T2-H2's capability register c_1 holds a read-write capability to the memory region $[p, p + s]$, the operation to write value val in variable a is authorized.

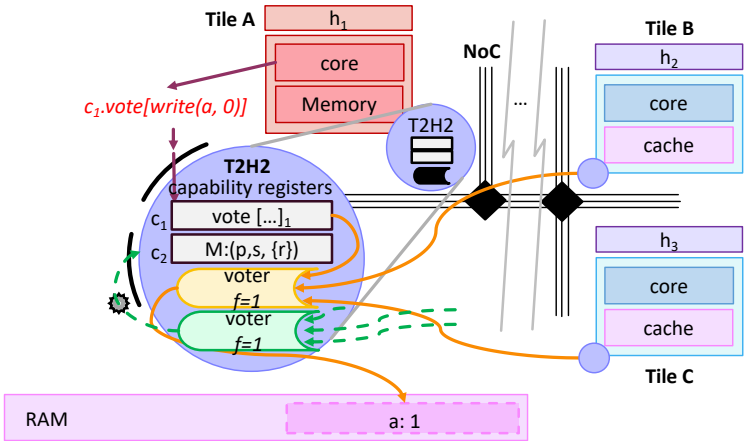


Figure 4.5: Consensual update of location a in the tile-external memory block (upper voter) and consensual reconfiguration of capability register c_2 in the T2-H2 of tile A. Reconfiguration is always consensual (requiring agreement of a majority of the tiles A, B and C); tile-external resources may be optionally treated in that manner (by granting access to a voter, but no direct access). The voter installs the majority decision (e.g., it updates location a with the consensual value 1 or the capability in c_2 with the agreed upon read-only memory capability).

However, T2-H2 also supports voting, particularly useful when, e.g., platform management software or hypervisor replicas must execute critical operations (e.g., privilege change or critical device accesses). These operations are voted upon, within preconfigured detection or tolerance mechanisms, to prevent compromised components from causing harm. Several strategies may be served by *Midir*, such as self-checking, recovery blocks, or *-out-of-n* error masking by majority voting in the presence of f faulty components, but they are all supported by the same baseline voting mechanism. Figure 4.5 represents a similar operation as in Figure 4.4, but in voted access form. The hypervisor replicas in Tile B and C vote to write value 1, while the one in Tile A, being faulty, votes to write value 0. In order to perform these votes, all tiles invoke a capability on their local T2-H2 to access the designated voter (in this case, the upper voter (orange) residing on Tile A's T2-H2). Given that a majority of tiles voted to write 1, value 1 will be written to variable a.

4.3.2 Consensual Privilege Change

One particularly relevant scenario for voted access is consensual reconfiguration of the T2-H2 instances themselves. T2-H2's reconfiguration interface (see Figure 4.4) is accessible only through a voter and cannot ever be invoked directly.

Let us understand why this is a relevant innovation. In conventional OS design, any single kernel instance can directly or indirectly enforce modifications on platform resources. So, even in fault tolerant designs, a faulty or compromised kernel instance could still be able to threaten the platform correctness. For example, by manipulating page tables, any low-level OS kernel instance can install virtual-to-physical address mappings to any resource in the platform's memory map and access it through this mapping. Of course, a trusted underlying layer could solve this issue (e.g., by mediating page-table access). However, whether this layer is software, as in the Inktag kernel [Hof+13] or firmware, as in Intel SGX [CD16], it becomes a single point of failure for the platform.

Midir provides an additional level of protection, whereby the designer can constrain access to the platform reconfiguration, by allowing a particular mechanism, its registers and data structures to be only effected in a consensual manner, through a voter. As with general voting, discussed in Section 4.3.1, these voted accesses will normally correspond to the implementation of detection or tolerance strategies, in this case, directed to the protection against threats on the platform itself. In Figure 4.5, in green colour (lower voter), we represent such a flow of reconfiguration of a platform capability register in tile A's T2-H2. Exemplifying with *f-out-of-n* error masking in a replicated low-level hypervisor, several replicas make the reconfiguration request, which is voted (green voter). The result from the voter is wired through a special T2-H2 capability configuration interface to the concerned capability register, masking the presence of up to f faulty replicas.

Midir does not constrain how systems are configured and hence what faults are tolerated. Instead it provides the means to tolerate an incremental quality of faults, including, for highly critical systems, up to f faults in system management software (e.g., the hypervisor), by providing $n = 2f + 1$ hypervisor replicas and by subjecting all critical operations to voting.

4.3.3 Implementation

The implementation of capability invocation is standard (c.f.[NW74]): T2-H2 is invoked by tiles to perform external operations, then it looks up the capability in the capability register file, and forwards the operation to the NoC after the privilege check succeeds, silently dropping the operation otherwise. Replica IDs are communicated as labels in the capability [Har85], which T2-H2 inserts as an additional parameter into the operation.

Our voter implementation is driven by the following considerations and their impact on functional simplicity.

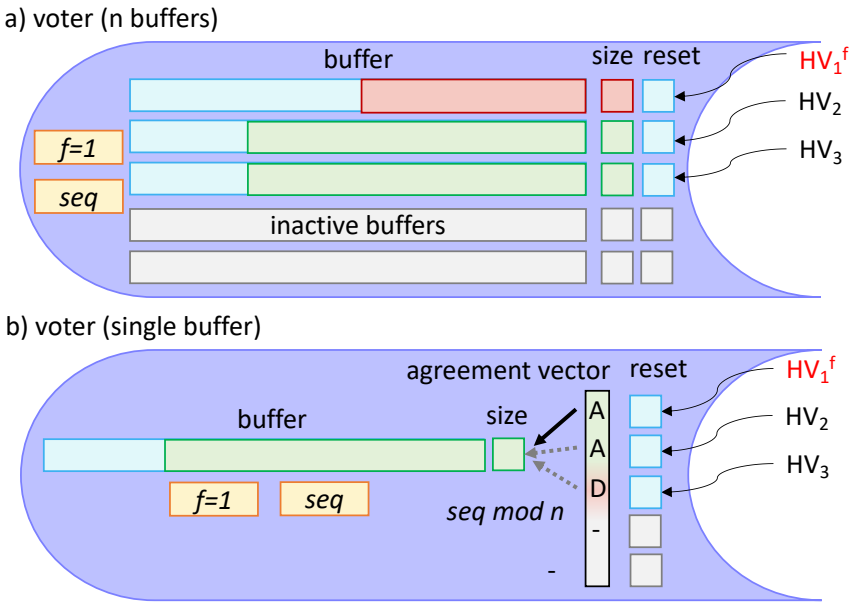


Figure 4.6: Internal structure of a voter. n (a) or a single (b) buffers hold the replicas' message to be voted upon and its length $size$. f defines the fault threshold, seq is a voter maintained sequence number. The agreement and reset vector are described below.

4.3.3.1 Buffered vs. Unbuffered Votes

Perhaps most impactful is the decision to buffer votes to allow replicas to make their proposals without first having to synchronize on the time when the signal for such a vote must be held. Although buffering increases the complexity of the voter, it decouples replicas, allowing them to act in a partially synchronous fashion and, as long as different voters are used, even partially out-of-order[1]. Buffering votes is ideal in a NoC architecture, since votes are transmitted as normal messages (e.g., writes to the memory mapped registers of the voter). Tiles can continue executing once the message is sent. We therefore implement voters to contain buffers (memory registers) for storing proposals from the different replicas for the current vote executed with this voter.

[1]To simplify monitoring of the progress of a system call, we have required that all replicas execute the critical operations of each system call in the same order. Operations of different system calls need not be constrained in this way, and, at the cost of a more complex progress tracking, this requirement can be further relaxed to: same order as far as a single voter is concerned.

4.3.3.2 Immediate vs. Deferred Masking

A similarly impactful decision is whether voters should be able to mask faults immediately. Alternatively, voting can be repeated until a valid proposal is made. The consequences, besides time to agreement, are the amount of memory needed for buffering votes vs. the complexity of the voter logic.

To mask faults and reach agreement immediately after $|Q| = f+1$ matching proposals arrive, the voter needs to buffer suggestions from at least $f + 1$ replicas. Since up to f such messages may be wrong and because the voter can only find out after receiving $f + 1$ matches, buffer space for at least $f + 1$ messages is needed to prevent having to repeat the vote.

We implemented two variants of T2-H2 voters to evaluate the resource/performance trade-off at the two extremes of this spectrum. The n-buffer variant (Figure 4.6 a) implements one message buffer per replica. Each time a message arrives, it is compared against all other stored messages and the operation applied once $f + 1$ buffers match. The single-buffer variant (Figure 4.6 b) trades agreement time for a more resource-efficient implementation: there is only one buffer; and only the current leader replica is granted write access to this buffer. The single-buffer voter follows a leader-follower voting scheme, with the leader proposing a vote and followers validating this proposal. To prevent inconsistency, the voter prevents modification of the leader proposal once the leader marks the proposal as ready, meaning that further proposals for the same sequence number will be ignored by the voter. This allows follower replicas to observe the stored message and express their agreement/disagreement. For this purpose, the single-buffer voter implements an agreement vector with one (initially empty: $-$) tri-state cell for each replica to express agreement A or disagreement D. Now, one of three things may happen when replicas propose:

(i) a majority of $f + 1$ or more replicas disagree with the leader proposal. In this case, the leader proposal is considered invalid and the operation is not applied; or

(ii) a majority of at least $f + 1$ replicas agree. In this case, the proposal is accepted and the voter applies the operation in its buffer.

(iii) the operation times out without a majority of replicas agreeing / disagreeing. In this case, the replicas record this error and repeat the vote after rotating to the next leader.

The n-buffer version requires logic circuits for pairwise buffer comparison, whereas in the single-buffer version a 2 data-bit majority gate over the agreement vector suffices, deeming the latter more resource efficient. On the other hand, although the single-buffer voter guarantees that, latest after repeating the vote f times, a healthy replica is elected as leader and makes a valid proposal, the n-buffer version may proceed as soon as it finds $f + 1$ matching proposal, making it more efficient in terms of execution time.

4.3.3.3 Internal vs. External Error Handling

The third question is whether the voter itself should include provisions for diagnosing errors and for informing replicas about them. Errors are detected when one replica diverges with the majority decision. Voter-initiated error handling translates to the voter tracing back to the voting replicas' cores to identify where to deliver error-handling interrupts. The expected complexity discourages such a solution. We therefore offload error handling to software and support replicas with means to track progress (the sequence number seq) and by suspending voting after detecting a mismatch. In this situation, seq does not advance, but the voter may still apply the operation (in case of $f + 1$ agreement). Replicas read the voter registers and buffers to diagnose the error, by looking for divergences.

To resume execution of suspended voters, replicas reset the voter, which clears all buffers and the agreement and reset vectors and advances the sequence number by one. Reset itself is a voted operation over the reset vector, which contains one bit per replica. The voter resets once $f + 1$ bits in this vector are set. Although this quorum guarantees that at least one correct replica agrees to resetting the voter, it does not prevent faulty replicas from resetting the voter prematurely, that is, before all correct replicas were able to retrieve the error state. The protocol in Section 4.5.4 handles this corner case.

4.3.3.4 Dimensioning Voters

The last question we discuss here is: for how many faults should the voter hardware be laid out. Since we aim at implementing voters in silicon, we have to make this choice at system design time to dimension buffers and vectors large enough for the maximum number of faults to tolerate (f_{max}). However, to not always have to execute at this maximum replication degree, a fault threshold $f \leq f_{max}$ of voters can be configured at boot time. For instance, if the system should tolerate up to $f_{max} = 3$ faults, it needs to be dimensioned to have $n_{max} = 2f_{max} + 1 = 7$ fields in the vectors (and an equal amount of buffers in the n-buffer variant). This voter can be operated at any fault threshold $0 \leq f \leq f_{max}$.

The voter design has been kept simple enough, and decoupled enough from the surrounding logic. As such, we can expect with high confidence that T2-H2 can be implemented and shown correct, as well as stay functional even when the tile it is associated with fails. A crashed T2-H2 prevents its tile from invoking any operation on tile-external resources, in particular from issueing votes. *Midir* ensures safety and liveness as long as the overall number of faulty tiles (including those with a crashed T2-H2) does not exceed f.

4.3.3.5 Voting Interface

In addition to the questions discussed above, one crucial point remains in the construction of a safe voter: its lock-free interface. Faulty replicas must not prevent correct ones from issuing votes, by locking the I/O through which votes are sent to the respective buffers (or agreement vectors). The voting interface must then provide separate channels for each replica to use and adapted internal logic to handle each channel, checking sequence numbers and storing the incoming vote in the appropriate location.

4.4 Properties

In this section we discuss some properties we are able to obtain from *Midir*'s design.

4.4.1 Privilege Reversion

Kernel (or any other low-level management software) replicas can agree to equip user-level applications with the resources an untrusted resource manager selects for them. However, because no healthy replica will agree to kernel replicas granting themselves or untrusted components access to the same memory pages, the recipient will be more privileged than the manager. We call this phenomenon *privilege reversion*.

Privilege reversion has virtuous consequences, it gives rise to interesting design patterns, which were previously available only at application level (assuming a trusted-trustworthy kernel), but which can now be exploited inside the kernel. For example, by agreeing to grant read access to memory, but never direct write access (i.e., write access only through a voter), we obtain the notion of consensually-updateable memory, which we shall use for the bulk of read-most shared kernel data structures. Moreover, granting a single application write access to a page while denying the same for all other components (including the kernel) creates an authentic buffer, since only this application can write. We shall use this for communicating with the kernel replicas.

4.4.2 Protection

Midir gives the designer latitude to use incremental protection, not preventing, in one extreme, configurations where a single instance controls T2-H2's privilege enforcement mechanism (by setting $f = 0$). It is even possible to disable the mechanism entirely by installing capabilities to cover the entire host physical address space. On the other extreme, it provides full protection, eliminating all software-level single points of failure, if the system is configured such that the following properties are preserved during

execution (starting from an appropriately initialized system[2]):

1. Impersonation prevention: no healthy replica agrees to installing in different tiles capabilities to the same voter with the same replica identifier (see Section 4.4.2.1).

2. Bypass prevention: no healthy replica agrees to installing capabilities to directly access a T2-H2 configuration interface, which would bypass voting.

3. Replica integrity preservation: under no circumstances healthy replicas agree to installing capabilities that convey direct write access to the code and local state of a replica, respective to critical data.

4.4.2.1 Replica Identifiers

There are two types of replica identifiers: The ID encoded in the capability and the T2-H2 ID used for immediate revocation.

The ID in the capability identifies towards the voter which replica executes on the tile. For example, if it is replica 2, then it will get the second slot in the agreement vector of the voter (or the second request buffer in the n buffer case). We encode this in the capabilities to allow replicas to have different IDs at different voters. If two tiles would hold such a capability to the same voter, then one could assume the role of the other when voting at this voter.

The one fixed identifier is the T2-H2 Identifier, which is an unique identifier value encoding which software is currently running on the tile. If other tasks have capabilities to invoke this software (e.g., by writing to a memory buffer they read), they will no longer be able to do so if this ID changes.

4.5 Fault and Intrusion Tolerant Micro-Hypervisors

We now turn our attention to the construction of *Midir*-aware FIT micro-hypervisors, such as suggested in Figure 4.7. We discuss this topic in terms of *Midir* and T2-H2, however, this could as well be implemented equivalently with *iBFT* (see Chapter 5), reaching consensus on system calls and using the trusted copy presented in Section 5.7.7 to apply the sensitive operations. Additionally, we use the hypervisor as an example, given its guest OS isolation-oriented nature and the criticality it represents. The remainder of this Section could equally be applied to another kind of low-level software. Note

[2]We boot all kernel replicas simultaneously, ensuring an initial capability setup that allows them to reach a voter. Otherwise, the setup disables all outstanding capabilities to the replicas' code and data segment, only granting the respective replica access to these memory regions. From there on, following properties 1–3, the replicas boot into an intrusion-tolerant system state, consensually giving themselves further privileges as needed.

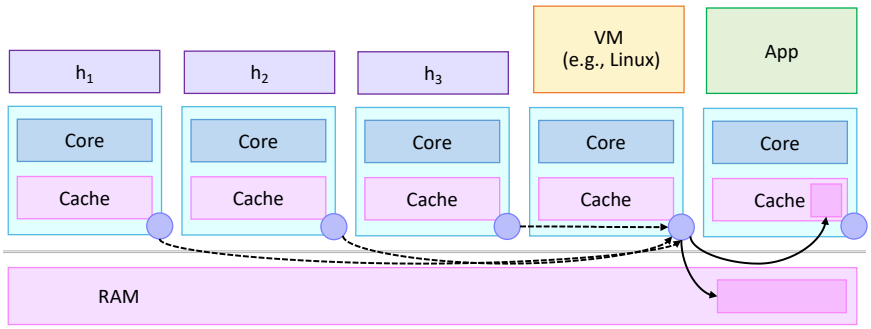


Figure 4.7: Overview of the *Midir* architecture: a multi-/manycore system augmented with T2-H2 hardware capability units (blue dots) at the NoC interface. Access to tile-external resources is subject to privilege confirmation in T2-H2 and possibly voting. Here, the hypervisor replicas $h_1, \ldots h_3$ consensually reconfigure the privileges of the VM on the 4th core, which in turn obtains access to a region of memory in the scratch-pad memory of the application on tile 5. Privilege change is a voted upon operation, indicated by dashed lines.

as well that, throughout this section we shall be using the example of a single buffer voter implementation and, thus, referring to the usage of a leader replica.

Hypervisor replicas execute on dedicated tiles, from where they remotely configure the privileges of applications executing on other tiles. Most of the other common OS-functionality (e.g., context switching, inter-process communication, (non-critical) device access, etc.) can be left to the application and its kernel-support libraries.

Midir gives the designer latitude to use incremental levels of protection for individual operations or sets thereof. On one extreme, configurations may be allowed where all accesses are direct, and thus unprotected by voting (setting up voters for direct pass-through of proposals, i.e., $f = 0$, to reconfigure capabilities).

On the other extreme, the highest level of protection, while retaining the flexibility of a manycore system, eliminates all software-level single points of failure[3] by subjecting all critical operations to voting. We focus on this facet. The replicated micro-hypervisor offers a system call interface executed by its replicas, entering a service loop and maintaining data structures used to handle system call requests, which they receive from applications, other replicas (e.g., requesting a privilege they lack for executing a system call) or from hardware (e.g., triggered by device interrupts). We provide an informal argument of the protocol's safety and liveness in Section 4.7.

Remembering that the unit of fault containment in *Midir* is the tile (equivalent to a node in a distributed system) the essential requirement for a fault tolerant micro-

[3]Modulo *Midir*'s T2-H2, which, justified through its simplicity, we assume will not fail.

hypervisor design is that the replicas behind critical operations are placed in different tiles, such that they communicate by messages, are subject to T2-H2 access control, and converge on the necessary votes as dictated by the algorithm. In order to fully enjoy the baseline functionality provided by *Midir*, a few additional design principles should be followed:

- **P.1** ***Impersonation prevention:*** Correct replicas must deny any operation with a replica identifier that is already in use (T2-H2 voting relies on identifying the individual replicas through their capability; no two replicas should have a capability to the same voter with the same identifier).

- **P.2** ***Bypass prevention*** Correct replicas must deny any operation attempting to grant direct write access to a consensual-update-only object.

Let us illustrate the design with the example of reallocating the tile to a different application. Signaling the tile, an application-specific library may save the state necessary to resume execution (e.g., utilizing memory assigned for this purpose). The actual switch then proceeds by resetting the tile followed by installing the capabilities the new application's library needs, in order to load its state. Obviously, reset and, as we have seen, privilege change are critical operations, which must be performed consensually to prevent compromised kernel replicas from prematurely stopping applications. Channeling such critical operations to voters and confining access with capabilities prevents faulty replicas from causing harm, since, as long as no more than f replicas become compromised, a correct majority out of the $n = 2f + 1$ replicas will outvote these operations. This turns system call execution into updates of replicated state and a sequence of voted operations, which we shall later call *subordinate votes*. This works as well with any other replicated critical software, even firmware such as in SGX (e.g., preventing enclave misconfiguration) or device drivers, when interacting with the physical world. Replies to system calls must also be voted upon, given that hypervisor replicas, by nature, act on behalf of multiple applications, possibly storing information of one that must not be revealed to others.

The above is of course true provided replicas have reached agreement on the system call to execute and on the parameters with which the client has invoked this call. Clients are applications and other low-level software (e.g., hypervisor) replicas that invoke system calls. A further role of the service loop is therefore to reach consensus on system call execution order and parameters. From our evaluation (Section 4.6) we found that *Midir*'s support for consensually executing critical operations also provides for accelerating the BFT protocol that the replicas must execute to reach agreement.

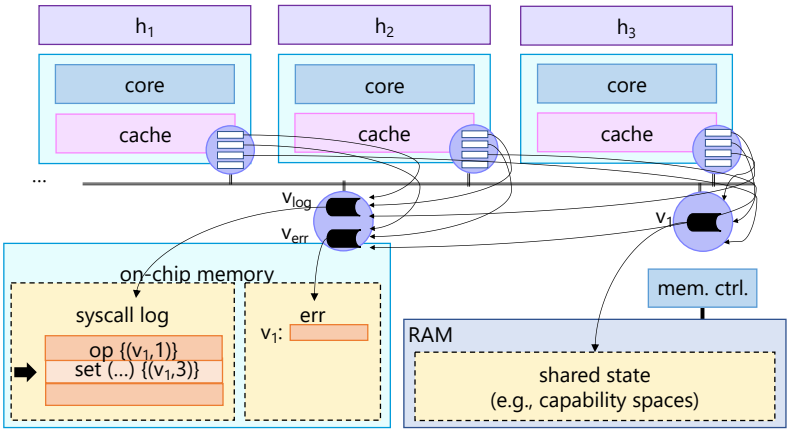


Figure 4.8: Read-shared, consensually updated data structures used by the hypervisor replicas: system calls are recorded in the syscall log, the error log keeps voting error information, and a capability space holds an application's capabilities.

4.5.1 Consensual System Calls

Figure 4.8 provides a more detailed picture of how T2-H2's voters and capability registers contribute to reaching consensus about the system call to execute and its parameters. The service loop of FIT hypervisors needs to reach consensus before it can start executing operations that may have critical side effects when misused.

The service loop utilizes two data structures: a *syscall log* and an *error log*.

System call log: A consensually updated ringbuffer — the *syscall log* — records agreed upon system calls and their parameters to give kernel replicas the opportunity to learn about those agreed upon. Otherwise, this information would only be available to the agreeing quorum of $f+1$ replicas and if faulty replicas participate in the agreement, but refuse to execute the system call later on, too few correct replicas would have obtained this knowledge to complete the system call. Storing agreed upon system calls in the log allows lagging replicas to catch up with the system calls they missed.

Error log: Similarly, the service loop utilizes an *error log* to protect error information from getting lost if the voter is reset prematurely before all replicas have learned about this error. Updates of the syscall and error logs are made through dedicated voters: v_{log} and v_{err}, respectively.

Macroscopically, clients place system call requests in authentic buffers, which the hypervisor replicas poll[4] for new requests. Consensual privilege change allows creating such buffers by granting write access to a single client, but to no hypervisor replica. The

[4]Sleep/wake protocols can be used in periods where no requests are pending.

leading hypervisor replica proposes one such system call by initiating a vote with v_{log}, which followers observe and agree or deny. Once written to the syscall log, replicas proceed by executing the system call and the votes for its critical operations, as well as responding to the client. We call these *subordinate votes* as they depend on the main vote, logging the system call. That is, no correct replica will engage in a subordinate vote unless the system call has been logged. Subordinate votes include at least replying to the client and advancing the syscall log to the next free slot. They are performed utilizing a set of voters $V = \{v_1, \ldots\}$ that is disjoint from $\{v_{log}, v_{err}\}$.

We make no assumptions on the order in which replicas update their local state (even transactional or speculative updates are imaginable). However, to simplify tracing the progress of the system call (and, in turn, the code that late or rebooted replicas have to execute to catch up), we require subordinate votes to be executed in the same order by all replicas and assume that this order is completely specified by the system call parameters.

Our rationale for agreeing on the system call first is to circumvent a fundamental problem of consensus protocols without authenticators: the impossibility to diagnose faults if messages can be altered during multicast operations [LSP82]. In our setting, cryptographic operations would come at overproportionally high costs relative to the speed of the transport medium (the NoC). Since consensus adds to system call execution times, having an execution time close to the NoC's speed is a desirable property. We therefore avoid sending unforgeable authentication tokens (e.g., HMACs) and instead exploit the authentication we obtain from a client being the single writer of its request buffer. However, given clients maintain write access to their request buffers, they can change the request after the leader has proposed it, but before followers validate it, which makes it impossible for followers to distinguish whether the leader proposed a wrong system call or whether the leader proposed the client's original suggestion, but the client changed it afterwards. In consequence, they cannot differentiate faulty clients from faulty leaders to provably identify the leader as faulty. We omit this form of error diagnosis for the system call vote to regain this property when we need it: in the subordinate votes for reaching agreement on critical operations.

Leaders tricked into such a fault are rotated and the new leader proceeds with all other pending requests before returning to the suspicious client.

The following details the protocols the hypervisor replicas execute to reach consensus on and execute system calls. Leveraging the generic voting pattern in Figure 4.9, replicas first reach agreement on the system call (Figure 4.10) to then consensually perform critical updates during its execution (Figure 4.11).

4.5.2 Generic Voting Pattern

Figure 4.9 shows the generic pattern and how replicas interact with voters. Evaluating the sequence number $v_i.seq$ of voter v_i, replicas identify the leader as the replica with

```
1   agreement:
2     seq_i := v_i.seq
3     if (replica = seq_i mod n) {
4       // leader
5       v_i.propose(op, seq_i)
6     } else {
7       // follower
8       wait for leader proposal: op
9       validate op
10      if (valid) v_i.confirm(op, seq_i)
11        else     v_i.decline(op, seq_i)
12    }
13    // all
14    wait for f+1 replicas to
15        agree/disagree/timeout
```

Figure 4.9: Generic voting pattern used in the service loop and when executing system calls.

identifier $v_i.seq \bmod n$ [5] in its capability. The leader proposes a request by invoking its vote capability to write operation op to its voter buffer, which the voter prevents from being changed once the leader marks this proposal as complete. Followers wait for the leader to complete its proposal to then validate the operation and express their agreement/disagreement (by submitting the operation they saw or by writing the corresponding value to the agreement vector (see Section 4.3.3)).

4.5.3 System Call Vote

In Phase 1, replicas first agree on the system call to execute following the generic pattern above. In Phase 2, they then vote on critical operations. Figure 4.10 shows the pseudocode for system call agreement. Lines 16–23 illustrate the client invocation pattern discussed above. The leader selects a pending system call (Line 26) with a valid opcode (Line 27) and prepares the entry to log. To prevent equivocation during subordinate votes (e.g., attempts to trick a replica into proposing the next system call without completing the current one), we enforce some additional principles:

- **P.3** ***Coordinated subordinate votes:*** correct replicas vote only on subordinate voters ($v_i \in V$) to execute the current system call.

[5] As long as enough tiles are available, n and f can be reconfigured, namely when adopting optimistic voting schemes. Such changes can namely de done on the go, provided a safe initialization, rejuvenation and relocation protocol. However, we leave the dynamic modification of these parameters and associated advantages for discussion in future work.

```
16  client c_k:
17    write m := syscall opcode + parameters
18        to c_k's request buffer
19    wait for reply in c_k's response buffer
20  hypervisor replica HV_i:
21    service loop:
22        poll all client buffers
23        remember new request (m, c_k) as pending
24      on pending request:
25        // leader
26        (m, c_k) := pending.remove_head
27        if (m is invalid syscall)
28          skip to next pending request
29        VS := ∅
30        for each voter v_i used to execute m
31          // collect voter sequence numbers
32          introspect v_i to read seq_i := v_i.seq
33          VS := VS ∪ {(v_i, seq_i)}
34        // follower
35        if (pending requests ≠ ∅)
36            set timeout
37        // all
38        v_log.agree_on (``write(log, ⟨m, c_k, VS⟩)'')
39           with validate :=
40             (m ≠ request from client c_k) ||
41             (v_log.seq ≠ seq_log) ||
42             (seq_v ≠ v.seq, where (v, seq_v) ∈ VS))
43        if (at least one replica disagrees)
44          v_log.vote_for_reset()
45        if (not f + 1 agreement)
46          repeat vote
47        execute m
```

Figure 4.10: Service loop - Phase 1: agree on next system call to execute

- **P.4** ***Presence of correct replica:*** no voted operation succeeds without at least one correct replica.

We enforce P.4 by requiring quorums of at least $f + 1$ matching votes, while preventing impersonation (P.1). In combination, these principles ensure that subordinate voters $v_i \in V$ will keep their state while in Phase 1 (including their sequence numbers). By agreeing, alongside the system call, on the first sequence number of all voters used in this system call (collected in Lines 29–33 in the set VS and validated in Line 42), we ensure that all replicas know all sequence numbers to start with in subordinate votes, even if they have been lagging behind. In the absence of errors, the j^{th} subordinate vote on v_i will be executed with sequence number $seq_i + j$, assuming $(v_i, seq_i) \in VS$ was the start sequence number of v_i. This agreement on the initial sequence number then allows for a simpler progress tracking in Phase 2, when executing subordinate votes.

Because of the impossibility in Section 4.5.1, system call votes operate with reduced error diagnostics: replicas reset v_{log} if it got suspended after disagreement (Lines 43, 44) and repeat votes for pending system calls unless they fail for all client-leader combinations, in which case they exclude this client.

4.5.4 Subordinate Votes

The code for executing subordinate votes in Figure 4.11 has to solve two problems:

1. Preserve determinism despite errors.
2. Prevent replicas from prematurely resetting voters.

From reaching agreement on the system call, we know that the first subordinate vote on v_i starts with seq_i because $(v_i, seq_i) \in VS$. As such, without errors, the j^{th} subordinate vote on v_i happens with sequence number $seq_i + j$. The same applies to votes with at least one disagreeing replica that all received $f + 1$ agreement because, after the voter resets (Line 62), they are not repeated (Line 66). The key for lagging replicas to catch up in case of error is to make sure they learn about all errors, so that they know how many times a vote was repeated and when it was successful. Assume the k^{th} subordinate vote ($k < j$) was the last to fail with seq_i^k, then k completed with $seq_i^k + 1$ and the system call progressed to subordinate request j if $v_i.seq - seq_i^k = j - k$.

Solutions to the second problem address the point that all replicas must learn about errors. With $n = 2f + 1$ and $|Q| = f + 1$, up to $n - |Q| = f$ replicas may lag behind while the remaining $|Q|$ progressed to another subordinate request or even to another system call. In particular, faulty replicas may fail a subordinate vote, but agree to reset the voter, which erases the error information about the failed vote from the voter and leaves behind as few as a single correct replica to know about the error. This scenario occurs if f faulty and one correct replica resets the voter before others diagnosed it.

```
48  HV_i.vote (log, v_i, seq_i, req, m, dest) {
49    if (syscall_log.log ≠ log)
50      return success
51    if (v_i.seq ≠ seq_i)
52      if ((err[v_i].log ≠ log) ||
53          (err[v_i].req ≠ req) ||
54          (err[v_i].eseq > seq_i + 1))
55        return success
56      push_error_and_reset_voter
57      if (!err[v_i].success)
58        repeat vote with seq_i + 1
59    // HV_i is up to speed with the others
60    v_i.agree_on(``write(dest, m)'') with seq_i
61         and validate := (m, dest) is valid
62    if (at least one replica disagrees)
63      push_error_and_reset_voter
64      initiate recovery
65    if (f + 1 agreement)
66      return success
67    repeat vote with seq_i + 1
68  }
69  push_error_and_reset_voter:
70    error := introspect(v_i)
71    v_err.agree_on(``write(err[v_i], error)'')
72        with validate :=
73          adjust own error information
74          (proposed error = own error)
75    if (error vote fails)
76      v_err.vote_for_reset(eseq)
77      repeat pushing the error
78    v_i.vote_for_reset(seq_i)
```

Figure 4.11: System call execution - Phase 2: subordinate votes and error handling

Clearly, without costly cryptographic information, the honest replica cannot convince others about what has happened. The following design principle solves this problem by preventing premature resets before error information is pushed to the error log.

- **P.5** ***No reset before error logging:*** correct replicas reset subordinate voters only after the error got logged.

This error state contains information about the current system call, i.e., the system call entry log; the subordinate vote req; the sequence number of the voter v_i; the point where it failed $eseq$ and which replicas agreed/disagreed. In consequence, lagging replicas can validate if the current subordinate vote succeeded (Lines 52–55) and, if not, who was responsible for it to fail. Voter v_i prevents destructive writes until it is reset, which P.5 and P.4 ensure happens only after error information was written to the log. Non-destructive writes are updates of empty buffers, respectively, updates of the agreement vector from timeout to agree/disagree and from empty to any of these three.

The argument for why the problem does not recur with the nested vote for logging the error state is as follows:

1. The state to push is held in the voter v_i. Therefore, even if a replica lags behind, finding v_i suspended, it knows what information to write to the log.

2. Because of P.5, and because at least $f + 1$ replicas are required (P.4) for votes to succeed, the only way to make progress is by writing correct error information.

Therefore, either faulty replicas agree to writing correct error information or eventually correct replicas catch up and write correct information. The exact information seen by the replicas may differ depending on the time they read it, i.e., in late reads, more replicas may have expressed their consent or disagreement. However, it will always contain at least the consensual result of the vote, i.e., whether $f + 1$ replicas agree, disagree or timed out, and, in the former two error cases, it identifies at least one replica that diverges from the majority (the leader, in case of $f + 1$ disagreement). This replica is proven faulty. Followers, reading error information after the leader and finding proposals of additional replicas, downgrade their own information to that of the leader after validating it as described above (Line 73). Repeating the vote while rotating the leader ensures that valid error information is proposed latest after f retries. It then suffices to reset v_{err}, whenever it becomes suspended (Line 76). Once error information is pushed, replicas vote to reset the voter v_i for the subordinate vote (Line 78) and continue executing it.

4.6 Experimental Results

We have implemented T2-H2 with both voter variants in VHDL on a Zynq-7 ZC702 Evaluation Board. We instantiated 3 Microblaze cores as tiles, running at 50 MHz, each

with one T2-H2, connecting the tiles through T2-H2 with an AXI interconnect (serving as the NoC). We have implemented and measured the performance of the service loop of a fault- and intrusion- tolerant hypervisor (Figure 4.10). The service loop is used to agree on and execute client-invoked system calls for two critical operations: granting and priming capabilities. Grant (`L4.map` [Lie95]) copies capabilities between capability spaces and prepares for later revocation. Prime consensually copies a capability from the client's capability space into a T2-H2 capability register, where it is ready for invocation. We have measured the performance of grant and prime in two different implementations of capability spaces, a container object for the capabilities an application possesses:

(i) as a private data structure in each replica (Section 4.6.1), requiring, in the case of prime, only the vote to install capabilities and two further to reply to the client and mark the system call as finished; and

(ii) as a read-shared, consensually-updated data structure (Section 4.6.2), trading off speed for a smaller memory footprint by introducing additional votes for track keeping.

As baselines, we compare to a cross-tile invoked singleton hypervisor (horizontal line), executing the same system calls on its private state (with no agreement or usage of the T2-H2s), with 1637 cycles for *grant* (1977 cycles for *prime*); and to a shared-memory variant of MinBFT[6] requiring 242824 cycles to agree on a system call. Our agreement protocol outperforms MinBFT by one order of magnitude.

A comparison to a cross-tile invoked singleton hypervisor allows us to understand the overhead the T2-H2 introduces in remote memory block access, which is present only in the execution of critical operations. The presented values for this baseline are justified by the absence of caches, as we want cores to be as decoupled as possible. The choice for comparison with MinBFT relates to its high efficiency and state-of-the-art popularity in hybrid BFT solutions.

4.6.1 Per-Replica Capability Space

Figure 4.12 shows the average performance of the ***grant*** and ***prime*** system calls in a per-replica capability space implementation relative to two baselines: ***null*** and a singleton hypervisor instance performing these system calls in a non-consensual manner (horizontal red line). Shown are the system calls broken down into individual votes and the Q5 / Q95 percentiles of the overall measurements.

[6] We omit client signatures in favor of authentic buffers, but implement UIs with HMACs. USIGs can be accessed without overhead.

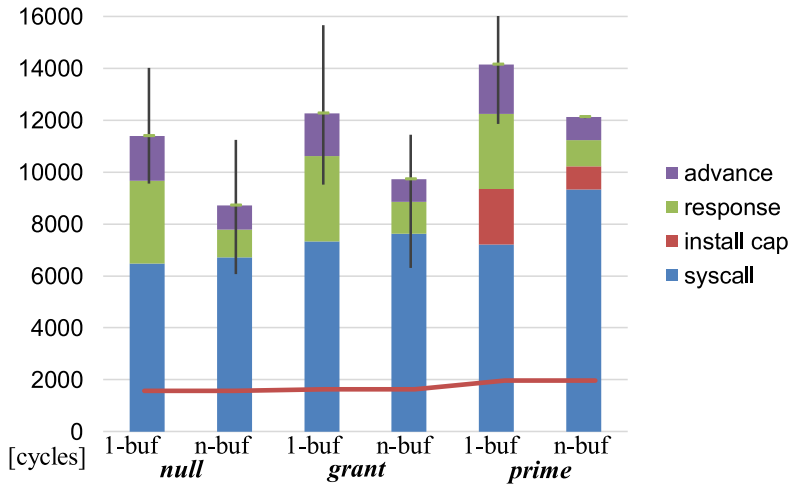


Figure 4.12: Average execution times of the three consensual system calls — ***null***, ***grant*** and ***prime*** — when executed on a per-replica capability space implementation. System calls are broken down into the individual votes for agreeing on the system call and for performing the critical updates required. Shown are also the Q5 / Q95 percentiles and the average costs of executing the respective system calls on a singleton hypervisor.

The minimal costs for learning about a system call request and executing it are 1571, 1637 and 1977 cycles on average for null, grant and prime, respectively. System calls for the single-buffer version have a factor of 8.9 (null) to 9.6 (grant) increase, which can be explained due to the voter not benefiting from caching. Whereas the singleton hypervisor merely has to copy one request from the memory where the client core places it, missing in all caches in the process, follower replicas have to poll the voter to wait for the leader to make a proposal and then confirm (or reject) the proposal made. Each such voter access amounts to costs equivalent to a cache miss.

As can be seen, reaching agreement on the subordinate votes is much faster, which is due to the fact that replicas already align themselves when reaching agreement on the system call to execute.

In the n-buffer version of the voter, higher costs occur during the agreement on the system call, which is due to the writing of the complete request to the voter, not just setting a bit in its agreement vector. However, subordinate votes are much faster, since replicas no longer wait for the leader to make a proposal. Instead, they just propose what should be written as critical operation.

4.6.2 Consensually-Updated Capability Space

Figure 4.13 shows a similar diagram as Figure 4.12, this time, however, for consensually-updated capability spaces. Granting and priming capabilities now require additional

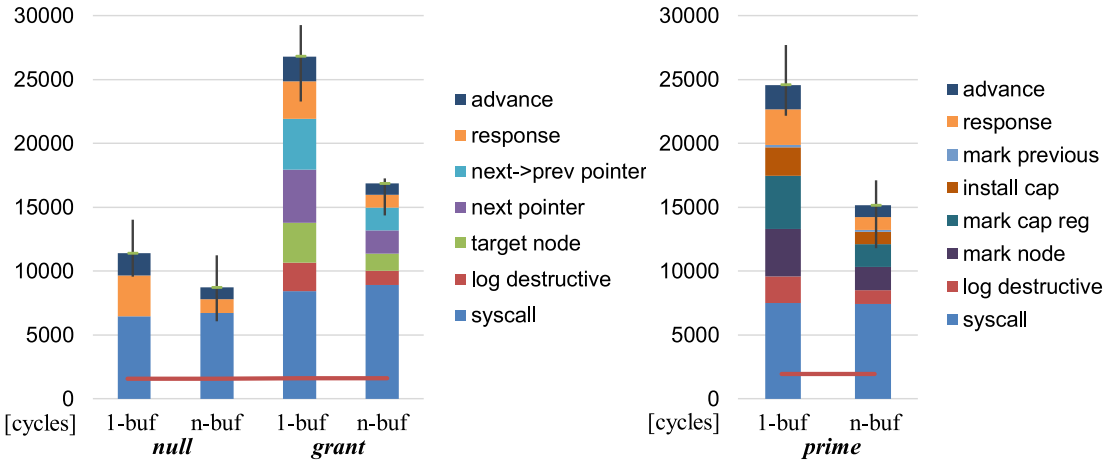


Figure 4.13: Average execution times of the three system calls for consensually-updated capability spaces.

votes to update the data structure.

This time, the 6.7 (single-buffer) and 7.3 (n-buffer) times slower performance relative to the singleton hypervisor can, once again, be explained due to the voter not benefiting from caching:

Singleton hypervisor: System call execution is triggered by the client writing to shared memory on one core and the hypervisor (on another core) reading it. From then on, all the operations happen locally in the core of the hypervisor without any interaction with the outside. Therefore, all memory operations aside from the invocation and reply hit in the core's cache, which, in our setting, responds within 1 cycle. The cross-core operations (invocation (1) + reply (2)) dominate these costs.

Replicated hypervisor: System call execution starts as well with invocation (1), but then, the leader needs to propose the request (2), followers validate it (3) and express agreement (4) upon which the voter updates the memory and all replicas wait for the vote to reach agreement (5). In the case of the per-replica capability space (case (i)), we then execute locally, but for replying (to not introduce storage channels) we have to repeat at least (4) + (5), assuming n-buffer voters. As such, even without any delays, we have 7 cache misses vs. 2 in the singleton hypervisor execution, hence a factor of 3.5. Additionally, more voter accesses are performed to read the sequence number, which we need for flow control.

To confirm that variations in fact originate from the agreement on the system call to execute, we have broken down system call execution into their individual votes and measured their Q5 and Q95 percentiles. Figures 4.14 and 4.15 show these values for single- and n-buffer voters, respectively. As expected, subordinate votes remain close to

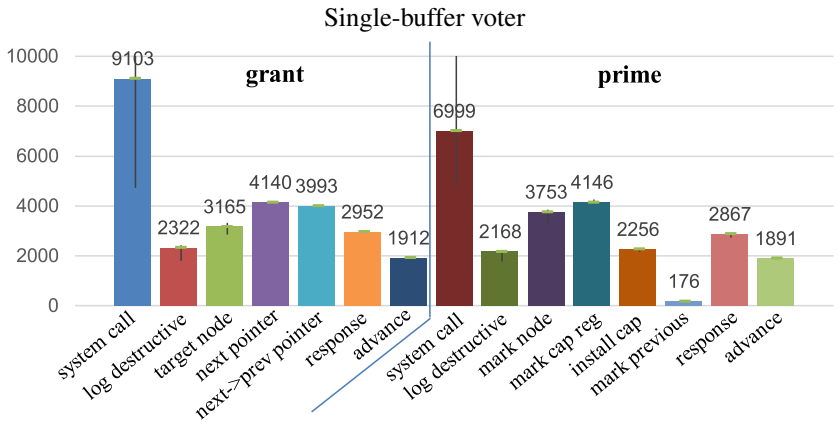


Figure 4.14: System calls broken down into individual votes. Shown are the Q5 and Q95 percentiles for the main system call vote and each subordinate vote for single-buffer voters.

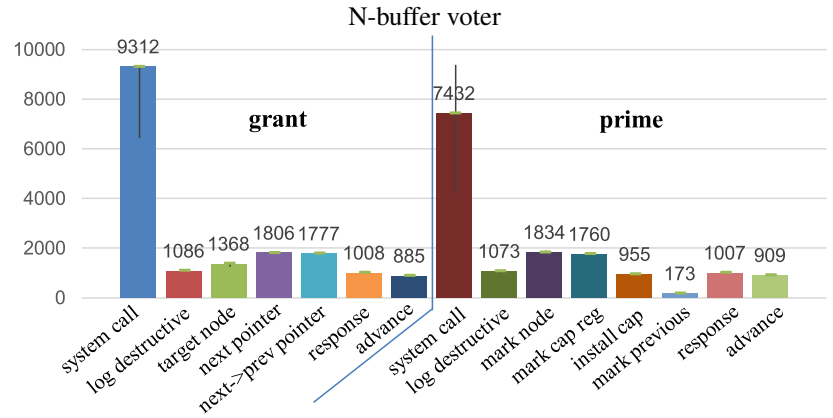


Figure 4.15: Same as Figure 4.14 for n-buffer voters.

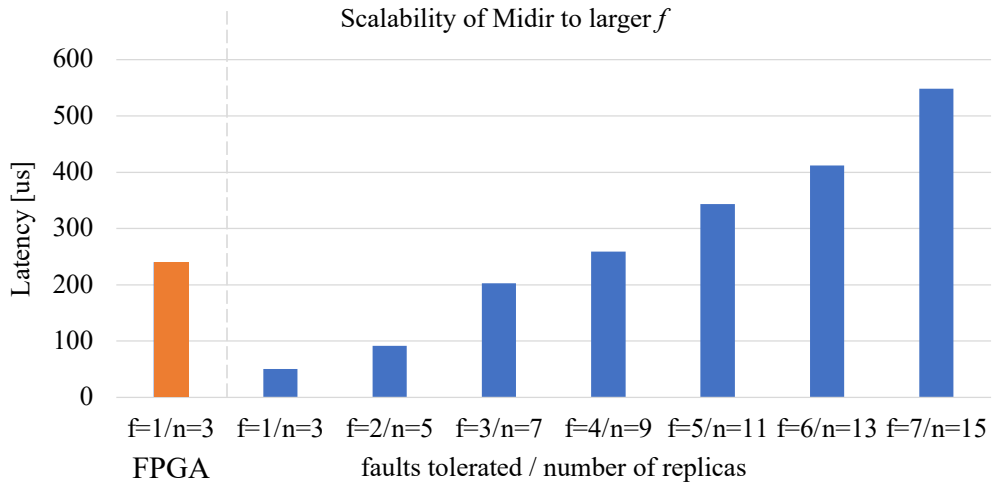


Figure 4.16: Latency of the `null` system calls for increasing number of replicas in microseconds.

their average execution times, whereas agreement on the system call varies significantly.

4.6.3 Scalability

Since our FPGA board's resource limitation prevents us from instantiating more replicas, we confirm the scalability of our approach in an emulation on x86. Hypervisor replicas are pinned as the sole application on the cores of a 24-core Intel Xeon CPU E5-4650 system, running at 2.10 GHz. They execute the same server loop like on the FPGA, but emulate voters in software.

Figure 4.16 shows the latency results of scaling the null system call to an increasing number of replicas and hence an increasing fault threshold from $f = 1$ to $f = 7$. Also shown (although not directly comparable) is the performance of the FPGA implementation, converted to microseconds. As can be seen, the execution of the null system call scales linearly with the number of replicas, which in part is due to the emulation having to acquire a lock during voting. We expect a similar, though less steep, linear increase in a larger scale FPGA implementation due to the additive effects of having to wait for the agreement of an increasing number of replicas with fluctuating system-call execution times.

	Single Buffer	N-Buffer
Common Definitions	129 Lines of C++	
T2H2 Interface	142 LoC++	134 LoC++
Service Loop and Subordinate Votes	311 LoC++	309 LoC++
Capability Space (per replica)	242 LoC++	
Capability Space (consensual)	314 LoC++	
Capability Registers	46 / 605 Lines of VHDL	
Voter	187 / 1512 VHDL	176 / 1703 VHDL

Figure 4.17: Code size in lines of C++ / VHDL code (logic / total).

4.6.4 Code Size and Hardware Utilization

Figure 4.17 lists the code size (excluding initialization) for the service loop, for consensually executing critical operations and for interfacing with the capability registers. Also shown are the VHDL source lines of code for the logic only and for the overall design (including I/O declaration) of the voter and capability unit (containing the capability registers). As can be seen, the amount of code that each replica executes for the above grant and prime system call is well below 1000 lines of code. Faults in this code are masked by the majority of replicas outvoting faulty replicas in critical operations. Similarly, the hardware overhead is just above 400 lines of VHDL code for the logic plus 2411 lines of VHDL for connecting the logic to the AXI interface I/O and for mapping the corresponding internal signals. VHDL simply defines the logic to be programmed in the board, it is not executed by the voters or capability units.

Figure 4.18 shows the FPGA resources of the (post-synthesis) implementation of our components. LUTs are units with no state, used to implement the combinatorial logic; while registers hold state, e.g, to keep buffer contents, but implement no logic. Each F7 Mux (wide multiplexer) combines the outputs of two LUTs together, while F8 Muxes combine the outputs of two F7 Muxes.

Notice that the absolute resource requirement of T2-H2 will not increase significantly if more complex cores are to be controlled. The amount of registers needed for both the capability units and the voters, as well as the logic required for the voters, will increase linearly with higher numbers of n, but it will not increase in function of the complexity of other components, such as cores. Hence, the relative overhead will shrink when more complex tiles are considered.

	Capability Unit (20 cap. regs)	Voter Single Buf (f_{max} = 1)	Voter N Buf (f_{max} = 1)
Slice LUTs	750 / 1292	2230 / 3532	4438 / 5365
Slice Registers	3367 / 4351	3994 / 5983	6228 / 7702
F7 Muxes	115 / 307	0 / 290	0 / 736
F8 Muxes	42 / 138	0 / 97	0 / 352

Figure 4.18: FPGA resources required by T2-H2 (without / with AXI interface).

4.7 Midir Discussions

In this section, we argue about the safety and liveness of the BFT protocol for processing system calls (as shown in Figures 4.10 and 4.11). That is, any two healthy replicas execute the same system calls in the same order (safety) and all correct system calls will be eventually executed (liveness). We assume the combination of a sleep-wake notification mechanism and polling (summarized in Line 22) reveals any pending system call to all replicas. However, before we start arguing about safety and liveness, let us see why faulty replicas cannot trick healthy ones into participating in votes with a wrong sequence number.

System call execution involves as set of voters: the subordinate voters v_i mentioned in VS, plus v_{log} and v_{err}. By construction, voters ignore proposals and confirmations for all sequence numbers other than the current one and only if voting is not suspended. That is, a voter v_i will only react to commands with a sequence number seq if $seq = v_i.seq$. Sequence numbers advance only if $f + 1$ replicas agree to a proposal and no replica disagrees, or if $f + 1$ replicas agree to reset the voter due to some error case (e.g., one or more replicas disagreeing with the proposal).

From property $P.3$ and $P.5$ and the arguments we have given in Sections 4.5.3 and 4.5.4 we know that no healthy replica participates in reset before error information has been confirmed by such a replica and logged through v_{err}. Moreover, we know that healthy replicas will engage with subordinate voters only for executing the current system call they process. This means either the replica is participating in the current system call or it was lagging behind other replicas. In the latter case, the sequence numbers it will use to invoke the voter are smaller[7] and the voter will ignore the request without any effect.

[7]We assume sequence numbers used by lagging replicas will never be overtaken by the current write and say that such a sequence number is smaller, despite possible wrap-arounds of the used integer. We substantiate this assumption by implementing a large enough sequence number space and recommend rejuvenating replicas before their sequence numbers could be overtaken.

4.7.0.1 Safety

Proposing VS as part of the system call (Line 38) and including this as part of the agreement (Line 42) means a fault-threshold exceeding quorum of replicas agrees to the starting point of subordinate votes and from there we know, from the arguments given in Section 4.5.4, that without errors the j^{th} subordinate vote on a voter v_i is executed at $seq_i + j$ where $(v_i, seq_i) \in VS$ (and similarly with errors, by recording and acknowledging the number of retries). Therefore, if a healthy replica votes for a subordinate vote, it will always vote with the correct sequence number, which implies faulty replicas cannot leverage this vote/agreement to confirm a different request.

From the above, we can conclude safety holds, by seeing that replicas will not agree on different system calls for the same sequence number. The voter will only write system calls to the log which received $f + 1$ agreement, and the log position is advanced consensually and in a way that allows all replicas to learn about updates (last subordinate vote of the previous system call). The voter itself thereby prevents equivocation by freezing the proposal the leader makes for the current sequence number, i.e., by preventing it from being overwritten for the current sequence number. Additionally, sequence number will not be reused for the same vote since both successful requests and reset advances this number.

From safety of the logged system call, its parameters and VS, then follows the safe execution of the subordinate votes, given that the j^{th} subordinate vote on voter v_i is completely defined by these aspects. Notice that all healthy replicas execute the logged system calls, including their subordinate votes. In particular, Line 50 will not lead to skipping the execution of the remaining system call, but only short cuts through the subordinate votes when realizing that the system call has already been completed. Healthy, but lagging replicas therefore first update their state with logged system calls before engaging in new system call requests.

Notice also that, while it is possible for faulty clients to trick leaders in proposing a system call that followers will not confirm, the consequence of this is merely a rotation of the leader (by reset of v_{log} in Line 43) and the next leader continuing with another client.

4.7.0.2 Liveness

What remains to be seen is why the system is live (i.e., why it will eventually process all requests from correct clients). The combination of sleep-wake and polling in Line 22 will iterate through all client/replica combinations. Therefore, each valid client will repeatedly find a correct leader who proposes the request. Partial synchrony then ensures that during the long enough periods of synchronous behavior, healthy replicas engage in processing this request. Let us therefore, for the following argument, assume request processing happens in such a good phase and will not time out. Then latest after rotating

through f leaders, the client will find a healthy leader to propose the request.

As shown in Line 14 and 15, replicas will wait for either $f + 1$ replicas to agree, $f + 1$ replicas to disagre or $f + 1$ replicas to time out. Thus, if the request is proposed by a healthy leader (or by a faulty, but stealthy leader in a correct manner) at most f (respectively $f - 1$) replicas can disagree and, in the absence of timeouts, $f + 1$ agreement will be reached. Then, even if the vote is suspended due to a disagreeing replica, the voter will record the system call in the log and all healthy replicas will proceed by executing the logged call (after resetting v_{log} in Line 44 to return this voter into a state where it accepts further votes, including the next system call).

For subordinate votes, a similar argument applies. In the absence of timeouts during long enough phases of synchrony, when a replica proposes an operation for a subordinate vote, replicas wait until either $f + 1$ replicas agree to the proposal (in which case the voter executes the operation, e.g., by writing to the specified destination), even if a minority of replicas disagree; or $f + 1$ replicas disagree. Disagreeing replicas causes an error to be recorded and the vote to be repeated. From the arguments in Section 4.5.4 we know that error logging makes progress latest when a healthy replica proposes a valid error record and when lagging healthy replicas catch up to find the error information in the voter (remember $P5$ prevents premature reset before the correct information is logged). As such, latest after rotating through f faulty leaders a healthy leader will propose and reach $f+1$ agreement (from healthy followers or from stealthy faulty replicas responding correctly). This ensures that each subordinate vote gets executed and, consequently, the system call as a whole.

Having seen that the proposed BFT protocol for system call execution is in fact safe and life, we now focus on the implementation of the voters and how it ensures the behavior we require, namely freezing proposals and suspension until consensual reset.

Chapter 5

iBFT

Our second solution to the stipulated problem is *iBFT*. This approach is not merely a different idea to the implementation of SPoF-free systems, instead it presents different advantages and drawbacks which shall later be discussed in Chapter 6. Nevertheless, *iBFT* is primarily concerned with accelerating consensus in a tightly-coupled environment, so in that sense, it differs from *Midir*'s primary goal of eliminating all SPoFs and *not* using, in any possible configuration, a trusted underlying layer. *iBFT* can still, however, be used for the same goal as *Midir*.

5.1 The iBFT Architecture

iBFT is a consensus protocol, augmented with trusted-trustworthy devices (such as *Midir*), for tightly-coupled systems which focuses on a mechanism — called *Introspection* — that grants replicas the ability to observe the internal state and progress of their peers without requiring the peers' active involvement or consent. Rather than having to request information and wait for the other replica to send it, introspection allows information-providing replicas to remain passive (or even become faulty) without preventing other replicas from reading (but not writing) their state. In essence, *iBFT* is an accelerated form of consensus that takes advantage of the low overhead of operations like `memcpy` and `memcmp` to achieve an efficient form of fault tolerance. In order to avert equivocation without having to rely on cryptography, replicas' local memories, used to store voting and progress information, are hardened with a trusted-trustworthy mechanism that transforms regular memory into read-shared *write-once memory*, preventing votes from being changed until reset in a consensual manner.

Figure 5.1 gives a general overview of an *iBFT*-supporting architecture. Shown are the containment domains (tiles), including a core and a memory whose write ports are exclusively connected to this core. Other cores are connected through the NoC only to the read ports of this memory. The same holds for slices of tagged memory (t-mem

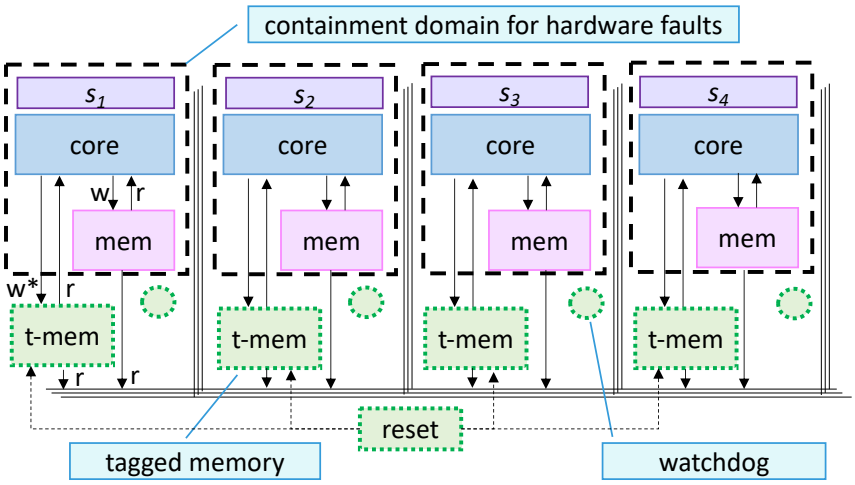


Figure 5.1: *iBFT* architecture overview.

in the figure), that cores use to implement the trusted-trustworthy write-once memory abstraction (see Section 5.4 for an in-depth explanation).

Replicas leverage shared memory for communicating with local clients and *wo* memory for communicating with their peers. Remote clients can be supported through local proxies and user level networking or through the operating-system kernel invoking all replicas with incoming client requests. As a hybrid BFT-SMR protocol, *iBFT* requires $n = 2f + 1$ replicas to tolerate up to f faults.

5.1.1 Setup

Figure 5.2 shows the basic setup of shared memory buffers connecting the server replicas among themselves and to their clients. Each client c_i has a request buffer (req) mapped writable to its address space and read-only to the address space of all replicas. Conversely, service replicas (s_1, s_2, s_3 for $f = 1$ and $n = 3$) use per-client writable reply buffers, which are mapped read-only into the client address space.

Each replica receives restricted write access ($rw*$) to its memory and read-only access to the memories of other replicas[1]. Replicas organize memory in slots. Each slot is comprised of one character string, used by the leading replica to record the client request m to execute, a client sequence number seq and the identifier c_i of the client, and of n bitfields (flags), one for each replica and used to store status information and to express agreement. The tagged-memory device ensures that the character string is sen-

[1] Write access is restricted in the sense of allowing values to be written exactly once in between resets (see Section 5.4).

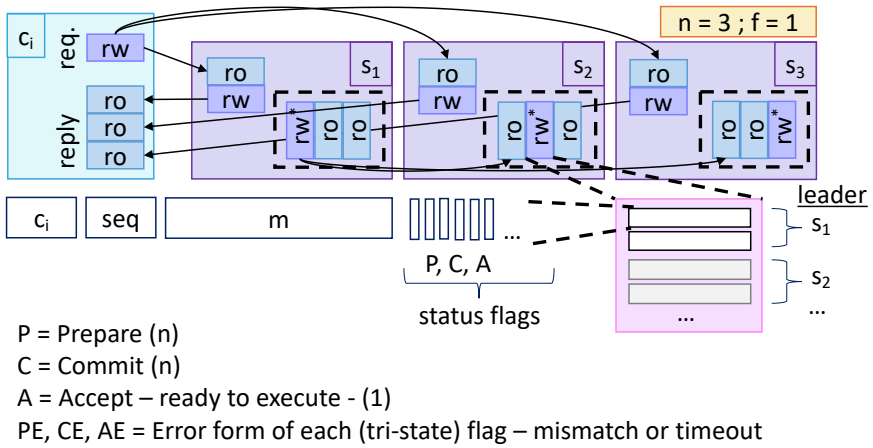


Figure 5.2: Setup and permissions of shared and memory buffers and internal structure of the protocol buffers in *wo* memory.

sitive to the bitfield of the leader and ensures that no further modification of the string are possible once a bit is set in the bitfield.

Because each buffer can be written by exactly one replica (or in case of request buffers by exactly one client), we have implicit writer authentication, but this authentication is not transferable, as explained in Section 3.1.2.2.

5.1.2 Execution Environment

There are several ways to establish the execution environment *iBFT* requires. For example, trusted-trustworthy kernels, at least the boot code of such kernels, may establish shared-memory mappings to grant read-only access to remote peers and restricted write access to memory. However, it is also possible to completely remove such a kernel (or boot code) from the RCB by hardwiring each replica's core to the read port (but not to the write port) of the trusted memory devices defined in Section 5.4. *iBFT* requires no kernel support or privilege changes once replicas are started, but will of course benefit from client-identifying signals or from a kernel for sharing the CPUs of replicas with other applications. Even the wake up of passive replicas can work without kernel support, provided the platform offers a sleep/wait mechanism, such as x86 `monitor`/`mwait`, at application level.

iBFT is concerned primarily with reaching consensus among replicas efficiently. However, with the help of a trusted copy operation, which we shall further describe in Section 5.7.7, *iBFT* can be converted into a resilience mechanism that is as powerful as *Midir* (though with a larger RCB).

5.2 Fault Model

Introspection is built for scenarios where a number of tightly-coupled nodes operate in consensus in a multitude of possible configurations with different implications on the RCB and hence on the trustworthiness of the replication scheme. We consider hosted as well as bare-metal implementations (e.g., with replicas executing in a single chip on the cores of a multi- or many-core system). In the remainder of this thesis, we will primarily refer to a bare-metal execution of *iBFT*. For bare-metal configurations, we consider tightly-coupled systems to be comprised of sufficiently many nodes to execute all n replicas $\mathbb{N} = \{s_0, \ldots, s_{n-1}\}$ concurrently, such that $n = 2f+1$. We follow a model with architectural hybridization, where trusted-trustworthy components and other parts of the RCB follow a distinct fault model.

We consider fault models at two ends of a spectrum. At the one end, we investigate contemporary systems that rely on a trusted hypervisor to configure and isolate replicas. In these systems, the software and hardware components required for establishing the replicas' fault containment (e.g., memory management units (MMUs)) remain part of the RCB. On this end of the spectrum, up to f replicas may exhibit arbitrary faults, but only at application-level. Hypervisor and hardware faults cannot be tolerated. At the other end of the spectrum are systems where cores may fail arbitrarily, even at hardware level, but not in a way where such a hardware failure brings down other cores. At this end, we tolerate up to f arbitrary faults at hard- or software-level, as long as the physical effects of faults remain confined to the core or the data it produces. These include bit-flips in local state, wrong computations, among others, but no power glitches that bring down neighboring cores and also no faults in the power distribution and clock networks, which are often shared and span large areas of the chip. Multi-chip solutions, such as triplicated ECUs with access to a shared memory, are one example of such a configuration. However, multi- and manycore systems on a chip have the potential to achieve the same fault containment, provided power and clock distribution are handled carefully. In particular chiplet-based solutions bear the promise of achieving the required level of fault containment.

The former side of the spectrum seems to contradict the whole purpose of this thesis, however, we remind the reader that (i) this represents only one end of the spectrum of configurations where *iBFT* could be used and (ii), unlike *Midir*, *iBFT*'a primary concern is an accelerated form of consensus, although it can also be used to achieve the same goals as *Midir*.

Implementations of the trusted-trustworthy component, write-once memory (*wo* for short), may follow distinct fault models of which we consider two flavors, orthogonal to the question of which parts of the hardware to trust:

- Write-once memory implementations that do not fail.

- Write-once memory implementations that can fail, but only by crashing and in a detectable manner.

For the former, we assume these memories to eventually complete *Introspection* and write operations and to report the last value written. Moreover, we assume they prevent overwriting values that have been marked as 'ready'. In this setting, no further progress guarantees can be conveyed once a write-once memory crashes. Our second trust model considers such crashes. We aim to continue guaranteeing progress unless more than a total of f replicas become faulty or their memories crash. We further assume these memories crash only in a detectable manner. As long as memory value errors build up slowly, the combination of ECCs, memory scrubbing and deliberate crashing[2] (once ECC detects more errors than can be corrected) ensures safety despite crashes. Also notice that we only bound the total number of faults, not distinguishing replicas with a crashed write-once memory from compromised replicas. This aspect will become important for the safety of our approach, since with c write-once memories crashed, we will assume that the remaining system has to cope only with up to $f - c$ Byzantine replicas. One added benefit of this fault model is that intrusion detection systems may deliberately crash a write-once memory to silence a suspected faulty replica.

When recovering, *iBFT* avoids extra replicas if service down times can be tolerated[3].

5.3 Introspection

Introspection grants replicas the ability to observe the internal state and progress of their peers without requiring the peers' active involvement or consent. Rather than having to request information and wait for the other replica to send it, *Introspection* allows information-providing replicas to remain passive (or even become faulty) without preventing other replicas from reading (but not writing) their peer's state. In this setting, the following challenges arise (see Section 3.1.2 for a deeper discussion):

1. From BFT-SMR protocols for tightly-coupled systems, one expects performance characteristics close to the speed of the replica-connecting communication media. In case of NoCs, this is several orders of magnitude faster than Ethernet TCP/IP or UDP;

2. Because of this higher communication speed, the relative costs of cryptographic means, traditionally used for ensuring transferable authentication, become prohibitively high.

[2] The main reason a write-once memory would crash itself is when its correction ability for memory faults is exhausted. Deliberate crashing is an additional mechanism, which requires consensus among replicas and is applied only after a replica revealed itself as Byzantine, which cannot be known initially.

[3] Service down-time is only required if one wishes to prevent having to use the additional $2k$ replicas (as described in [Sou+10]), where k equals the number of simultaneously rejuvenating replicas.

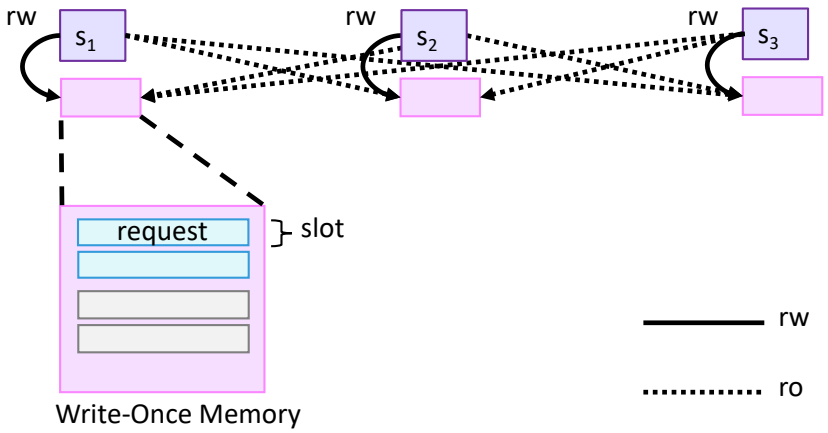


Figure 5.3: Relation among replicas and request blocks. The *rw* relation means the replica has read and write access to the block and *ro* means the block is read-only for that replica.

To address these challenges, each node leverages a trusted-trustworthy component, implementing the notion of write-once (*wo*) memory: replicas obtain write access to their dedicated *wo* memory block and read-only access to the blocks of all other replicas. Figure 5.3 depicts this relation among replicas and request blocks. Once the replica marks a value (a request) as written in their block, the trusted component prevents it from being overwritten and hence replicas from changing proposals.

With such a mechanism in place, we designed a hybrid[4] *Introspection* protocol, *iBFT*, which we shall describe in detail in Section 5.5. *iBFT* naturally leans towards the concept of optimistic protocols [Kap+12; DCK15], since passive replicas merely have to introspect the progress of active replicas to catch up. Likewise, introspection relieves replicas from assembling and transferring the progress they have made, which further simplifies exceptional situations like view change or the catch-up after replicas have been rejuvenated. For the latter, it is sufficient to reset to-be-rejuvenated replicas to a fresh, diverse instance, from which they catch up independently.

5.4 Write-Once Memory

Write-once (*wo*) memory is a trusted-trustworthy memory abstraction, which leaves reads unconstrained, but prevents successfully written values from being overwritten until the location holding this value is reset. Reset is a consensual operation, which requires agreement from $f + 1$ replicas. More specifically, it prevents replicas from

[4]Meaning, base on architectural hybridization.

modifying consensus information (the request string or progress flags) after they have been set and until consensually reset.

We shall use two types of data for this type of memory:

- *Write-once tri-state bitfields*, whose bits can be set, but not cleared. Bits are split into agreement and error bits, forming together the tri-state. Setting an agreement bit, prevents the corresponding error bit to be set and vice versa.

- *Fixed-size character strings*, for requests, which cannot be overwritten once the string is marked 'ready' (e.g., by setting a bit in a corresponding write-once bitfield).

In *iBFT*, a leader replica encodes client requests in a character string, stores it in its write-once memory buffer and marks it as 'ready'. Introspecting this buffer and observing this status, peers detect this proposal and know from its status that the proposing replica can no longer change what is suggested, which prevents equivocation. Therefore, because followers read the same location as the leader, the leader cannot lie inconsistently about the client or its request. Note, it is still possible for a leader to make up a request. Followers express their agreement/disagreement in a similar manner by setting the corresponding bits in a write-once bitfield, which prevents equivocation during this protocol step as well (i.e., a replica indicating agreement toward one of its peers and disagreement to others).

We consider and evaluate two implementations of write-once memory:

1. Using microcode-based atomic operations to conditionally set bits in bitfields or write parts of the string, provided the string is not marked ready - Section 5.4.1;

2. Using tagged memory hardware devices, configured as shown in Figure 5.4 - Section 5.4.2.

The first variant is a trivial microcode exercise by constraining the operations that can write the otherwise read-only memory pages used for *wo*-memory. In fact, aside from this enforcement, contemporary architectures, such a Intel x86, can already emulate *wo* memory in a performance-preserving manner. Write-once bitfields are written exclusively by bit set operations (e.g., a generalized `lock; bts` as in x86, i.e., atomic bit test and set, but for tri-state flags). Write-once character strings are written by atomic compare and swap, where compare checks for a specific value reserved to denote an empty buffer. Of course, full microcode access would also allow for cache-lock protected multi-address conditional writes, checking the bitfield and writing conditionally to the bits being clear. We have implemented the above emulation (cache-based) on x86 and a full implementation of the second variant on a Zync ZC702 FPGA. We describe both in the following two sections.

Naturally, microcode-based variants rely on the correctness of all cores, caches and of the cache coherence logic, leading to quite a large RCB in terms of necessarily trusted hardware. Our tagged-memory variant partially lifts these correctness assumptions, by removing the above components from the reliable computing base. However, it does so by requiring special hardware to implement tags and to interpose writes to *wo* memory for the purpose of checking tags. Instead of all cores and the whole memory subsystem, only the *tag-based* wo *memory devices* must be trusted.

This trust can come in two forms: *trusted to not fail at all*, and, as it is more common in other works leveraging architectural hybridization, *trusted to fail only by crashing*. In this work, we investigate both and discuss the complications that arise from the latter, even if it can be reliably detected whether a *wo* memory crashed.

5.4.1 Microcode-Based Write-Once Memory

By restricting which operations can be executed on write-once memory blocks (e.g., through a memory type or page permission flag), it is possible to utilize standard memory subsystems for implementing *wo* memory. The *wo* bitfields can be constrained to only allow atomic bit-set operations (again, a generalized `bit_test_and_set`), checking both error and agreement bits, and the *wo* strings can be realized by reserving one value (e.g., `exp = ~0UL`) to denote writable words, and by writing with atomic `compare_exchange(dest, exp, value)`.

Cache locks are one common way to implement atomic read-modify-write instructions in modern processor architectures. More fine-grain control over these locks opens further, more direct ways of implementing write-once semantics. For example, one could make writes to strings conditional to tags in the bitfield being clear.

The above mechanisms do not prevent caching write-once memory locations. Aside from requiring atomic operations to write these locations, microcode-based implementations therefore incur no extra overhead. However, the RCB of this variant necessarily includes all hardware components that are required to execute instructions atomically (i.e., all processors, caches and the used fragment of the memory subsystem). Our second variant further reduces the RCB.

5.4.2 Tagged-Memory Based Write-Once Memory

Tagged memory implementations significantly reduce the RCB, albeit at the cost of more expensive introspection operations. Rather than polling bitfields in local caches (possibly leveraging sleep-wake techniques, such as x86' `monitor` and `mwait`), replicas must now reach out to remote (yet on-chip) devices to observe changes. Values can no longer be cached without trusting caches and coherence logic.

Tagged memory [Feu71] stores values as unions of data and type while making it dependent on the type which operations can be executed on the data. For write-once

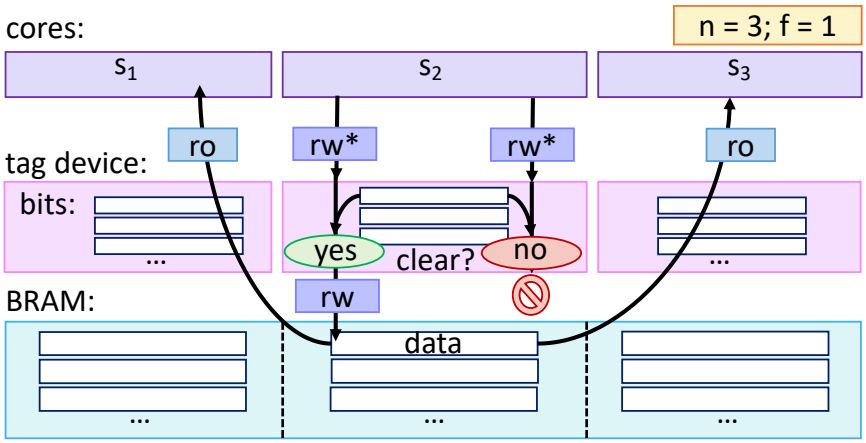


Figure 5.4: Implementation of *wo* memory as a combination of an AXI slave tag-mem device and a standard BRAM block.

memory, it suffices to implement a restricted form of tagged memory, using for each string buffer a field of sticky bits to define whether the data buffer remains writeable (all tag bits clear) or whether no further writes are allowed (any tag bit set).

To evaluate this variant, we have implemented *wo* memory as a combination of a standard per-replica block RAM (BRAM) area to hold *wo* strings and an AXI[5] slave device for implementing *wo* bitfields (one per buffer), as shown in Figure 5.4. The slave device interposes writes and prevents overwriting strings that are marked as ready by setting any one of the bits in the corresponding bitfield. Moreover, it prevents the replica from clearing bits by `and`-ing updates to the inverse of the bits that are already set (both error and agreement bits), prior to `or`-ing them to the stored value. We denote this in the figures as restricted read/write permissions ($rw*$). Peer replicas obtain direct read-only access to the bitfields and string buffers.

Since, in this variant, writes are mediated by the trusted AXI slave device and since the trusted BRAM guarantees that reads cannot modify data, cores no longer need to be trustworthy, provided proper fault containment. This reduces the RCB to the tagged memory device, its BRAM, and the reset devices, which we discuss in Section 5.4.4. BRAM can be further protected from accidental faults through error correcting codes.

[5]The Advanced eXtensible Interface (AXI), part of the ARM Advanced Microcontroller Bus Architecture 3 (AXI3) and 4 (AXI4) specifications, is a communication interface for on-chip communication. AXI interface IP blocks are common in block designs for Xilinx FPGAs, such as the one we use in our implementation (Zynq ZC702).

5.4.3 Implementation Details

The simplest implementation of *wo* memory would be a local memory and/or scratchpad SRAM (static random-access memory) with a device interposing the write enable signal of the core. More specifically, taking the example of the tagged-memory hardware implementation, write enabling is still done by the regular memory controller, however, the enable signal is `and`-ed (logic-wise, with no code) with an enable signal produced by the write-once memory upon receiving (from the memory controller) the address that is to be written. This hardware logic will evaluate the bitfields set for that address and determine whether the write is allowed. If it is not, the *wo* memory will decide a write enable signal of 0, which `and`-ed with the 1 from the memory controller will still prevent the write. The write-once memory is not merely a block of memory, but a simple hardware construction that contains memory space and an associated logic for checking flags for an incoming address that will determine the write enable output for writing on the memory block. The allocation of this memory can be hard-coded in hardware at design time (as in our proof of concept) or managed by a trusted hypervisor (in case of the least harsh fault model as described in the our fault model in Section 5.2).

5.4.4 Reset

Obviously, replicas consume *wo* memory space over time as they use it to handle requests. Therefore, once the available buffer space is used up, replicas have to reset *wo* memory to clear all tags before they can resume processing requests. We shall align this reset with the writing of a checkpoint and store the latter as well in *wo* memory. Double buffering alternates between checkpoint buffers and ensures that the latest checkpoint always remains intact.

Single handed or premature reset would allow replicas to equivocate, by resetting and overwriting a field after another replica has introspected it. We therefore make reset a consensual operation and require $f + 1$ replicas to agree before tags are cleared. The fact that a reset has just happened is recorded by setting reset flags RF, which are checked together with the remaining bits of the bitfield, but which can be cleared by the replica to continue writing to the device. We shall return to the necessity to synchronize checkpoints and resets in Section 5.5.4.

Several implementations of the above reset functionality are conceivable. For example, replicas could enter a trusted execution environment (TEE), e.g., enclaves, and implement reset by waiting for $f + 1$ replicas to enter their TEE before clearing *wo* bitfields and strings through normal writes or through a dedicated interface. Obviously, the permission to perform these operations must be restricted to the TEE.

Alternatively, reset could be implemented as a second device, similarly to write-once memories, collecting the intention to reset in a bitfield with one bit per replica. The device resets all *wo* memories (clearing bits and making strings writable again), as

described above, after $f + 1$ replicas agree by setting their reset bit. Naturally, reset must as well be part of the RCB.

We have implemented the latter for our tagged-based implementation, but recommend using a TEE-based implementation for microcode-based *wo* memory. The high costs of entering and leaving TEEs discourage implementing also *wo* memory writes in such an environment. We therefore limit the use of TEEs to the occasional resets required to clear the buffers.

It is of course possible to implement a reset device per core, capable of only resetting this core's write-once memory, this must, however, be done as a trusted operation.

5.5 iBFT Protocol

We refer back to Figure 5.2 to briefly explain status flags before diving into the protocol itself. P flags denote a resemblance to the prepare phase in PBFT, MinBFT and other BFT protocols and serve the purpose of making sure replicas compared the leader's proposal with the client request. C flags correspond to the commit phase and ensure at least $f + 1$ replicas prepared. A is set to mark the request ready to execute, i.e., after seeing $f + 1$ C flags. The error form of each flag (PE, CE and AE) denote a mismatch or timeout in each phase of the protocol and trigger error handling, being then also used to skip requests. AE is the final tri-state value of A and ensures the A flag is no longer modifiable in case of error, preventing faulty replicas from tricking others into executing requests.

In *iBFT* only the software replicas running the protocol (on different cores) and the write-once memories are replicated[6], as redundancy of other components is not mandated by the protocol. However, communication between the cores and all write-once memories and reset devices is needed, but, since NoCs are now a common means of having all-to-all communication between cores and certain peripherals like memories, this is not an issue.

Let us then describe the behaviour of all involved parties in each phase of the protocol.

5.5.1 Clients

Clients c_i store requests in their request buffer (Line 1 in Figure 5.5) and coordinate with the server replicas by setting the client sequence number[7] $c_i.req.seq$ to a value larger

[6] Replication of the reset device is also possible and, in fact, recommended.

[7] We shall use standard C notation for accessing arrays and structures, but allow whole structure copy and compare. For example, $buf_l[x].P[l]$ in Line 17 in Figure 5.6 refers to slot x in the buffer of replica l, accessing the P flag array in the message data structure at position l. That is, we set the P flag of replica s_l in this replica's buffer at the current request slot x.

```
1  client c_i:
2      c_i.req.m  := m
3      c_i.req.seq  := c_i.req.seq + 1
4      wait for f + 1 matching replies in c_i.reply[k]
5                  from different replicas s_k
```

Figure 5.5: Client Code

than the previously processed requests. After executing the request, the replicas s_i will reply with this sequence number to indicate that they have completed this request. In particular, this ensures that servers will not confuse requests that remain in the client's request buffer as new, since these requests will have a client sequence number $c_i.req.seq$ that is smaller than or equal to the client sequence number of requests that the server has already processed (Line 11 in Figure 5.6).

5.5.2 Normal Phase

iBFT draws inspiration from Veronese et al. [VCBL09] and implements a rotating leader scheme, while recording proposals and agreement status in *wo* memory. We start by discussing the *iBFT* pseudocode for error-free cases (shown in Figure 5.6), before we consider error handling and the code in Figure 5.7. We have marked in both figures the introspection operations `poll`[8], `copy` and `compare` in green. Lines marked with '*' are required only to cope with crashing *wo* memories.

Replicas take turns as leaders for a configurable number of `slots_per_leader` (Line 9). As long as unused slots are available, leaders insert pending client requests[9] from $c_i.req$ in the next free slot x they control[10], copying the message m, the client sequence number seq and the client number c_i into their buffer $buf_l[x]$ (Lines 10–16) and marking it as complete by setting their P flag (Line 17), which in turn instructs *wo* memory to prevent further writes to this character string.

Followers maintain a timeout for pending client requests to avoid indefinite waiting for a faulty leader not proposing pending requests[11]. To find out when the leader has proposed, they poll the P flag of the leader s_l in the leader's buffer (i.e., $buf_l[x].P[l]$),

[8] The operation `poll` refers to repeated polling until the target is found.

[9] *iBFT* supports multiple clients. The leader, when searching for new client requests, polls different clients, for instance in a round-robin fashion.

[10] In Figure 5.6, *buffer_length* refers to the number of slots and not the size of the slot.

[11] In a bare metal implementation, both the leader and its followers have no other means than polling to learn about new requests, cycling through all clients in the process. Naturally, this can be quite inefficient as the number of local clients grows. For this reason, we recommend complementing sleep/wait techniques with some way of informing about the source, triggering the wake up. Hosted setups provide this source information with the replica-invoking inter-process communication.

possibly using sleep/wake techniques to limit contention and to reduce energy consumption (Line 22).

Finding $P[l]$ set, followers know that the proposed request can no longer be changed by the leader. They therefore copy the leader proposal to their buffer (Line 25) and compare it against the proposal made by the client (Line 27). Upon match, they indicate their agreement, by setting the leader's P flag $P[l]$ in their buffer (i.e., $buf_k[x].P[l]$) (Line 28), otherwise, in case of mismatch (or timeout), they set this flag as PE (remember flags are tri-state).

Lacking transferable authentication, replicas cannot distinguish whether (1) the leader is faulty and made up a request, (2) the client is faulty and tricked the leader into proposing a wrong message[12], or (3) both client and leader are faulty. Leaders therefore copy the request into their *wo* memory and followers copy the leader request into their *wo* memories to prepare for the case when the *wo* memory of the leader might crash. Followers s_i compare the leader proposal against the client request and confirm this by setting $P[i]$

After that, leader and followers alike wait for $f+1$ replicas s_j to set their P flag $P[j]$ (Lines 32–36), after which they set their C flag (Line 37) (or CE in case of timeout) and wait until $f+1$ replicas have done the same before they consider the request as ready to execute, by setting the A flag (Line 43). In particular, they confirm before setting P-flags that remote copies match their copy as received from the leader.

Waiting for $f+1$ C-flags set in $f+1$ replicas ensures for the case when $c \leq f$ *wo* memories crash that $f-1$ replicas confirmed the copies in the $f-c+1$ remaining *wo* memories of replicas that participated in this operation. This third round is not required when no further guarantees are given upon *wo* memory crash.

Ready requests are executed by the code (Lines 45–48) once previous slots are executed (or skipped as a result of error handling). Replicas reply by writing both the response and the client sequence number to the reply buffer, which is mapped read-only to the client (Lines 50–51). The consensual reply resets the client buffer[13].

First marking slots by comparing proposals and by setting P flags accordingly, but then delaying execution until all previous slots are executed or skipped, allows for some out-of-order processing without sacrificing linearizability.

We shall return in Section 5.5.4, to checkpoints and the reset operation required to clear the buffer when wrapping around and discuss now how *iBFT* handles errors.

[12]The word "wrong" here relates to equivocation, i.e., making other replicas believe the leader is in the wrong when, in fact, the client changed the request.

[13]Multiple buffers can be used for each client to amortize reset costs.

```
6     server replica s_k:
7       /* round 1 */
8       /* next free slot: x */
9       let l = x div slots_per_leader mod n
10      if (s_k = s_l) /* leader */
11        if x < buffer_length
12          search new client requests
13          on new request req from c_i:
14            buf_l[x].req.c := c_i
15            buf_l[x].req.seq := c_i.req.seq
16            buf_l[x].req.m := c_i.req.m
17            set buf_l[x].P[l]
18            x := x + 1
19      else /* follower */
20        on new client requests (e.g., req from c_j):
21          set timeout (c_j)
22        poll buf_l[x].P[l]
23        on buf_l[x].P[l] is set:
24          /* found proposal from leader */
25  *       copy buf_l[x] to buf_k[x]
26          let c_i = buf_k[x].req.c_i
27          if compare buf_l[x].req = c_i.req
28            set buf_k[x].P[l]
29            x := x + 1
30      /* round 2: both */
31      for each slot y < x not ready to execute:
32        poll buf_j[y].P[j] of other replicas s_j
33        on buf_j[y].P[j] is set:
34  *       if buf_k[y].P[l] and compare buf_j[y] = buf_k[y]
35            set buf_k[y].P[j]
36        on f + 1 P-flags are set:
37  *       set buf_k[y].C[k]
38  *     /* round 3 */
39  *       poll buf_j[y].C[j] of other replicas s_j
40  *       on C[j] and f + 1 P-flags set in buf_j[y]:
41  *         set buf_k[y].C[j]
42  *       on f + 1 C-flags are set in f + 1 replicas
43            buf_k[y].A[k] /*mark y as ready to execute*/
44      /* consensus reached */
45      for each slot y < x:
46        if all slots z < y are executed or skipped
47           and ready to execute(y)
48          result := execute buf_k[y].req.m
49          /* reply to client */
50          c_i.reply[k].m := result
51          c_i.reply[k].seq := buf_k[y].req.seq
52      /* wrap around */
53      if all slots y < x are executed
54         and x = buffer_length
55        compute checkpoint C
56        store C in write-once memory and set P flag
57        if f + 1 matching checkpoints are written
58          reset flags, buffers
59            and the previous checkpoint; x := 0
```


Figure 5.6: Normal Phase, Checkpoint and Buffer Reset

5.5.3 Error Handling

Once healthy replicas time out they no longer modify their acceptance flags. Instead, they set the error flags corresponding to all acceptance flags (AE flags) not yet set in all slots y that have been proposed, but not yet completed, including in all slots for which the current leader is responsible. We denote the latter by $[x]$. *wo* memory detects if the A-flag or its corresponding error flag AE is set in $f + 1$ replicas and will trigger the equivalent of the operation from Line 63 in all *wo*-memories to ensure replicas can no longer change flags after the majority timed out. Replicas will not engage into actually processing this timeout before either $f + 1 - c$ replicas have prepared the request or $f + 1 - c$ reached an error state where the tri-state nature of flags prevent them from preparing it later. Here, c is the number of *wo* memories that have crashed.

Similar to MinBFT, we define as necessary condition for a replicas to have prepared a request that it has set $f + 1$ of its P-flags, which resembles *iBFT*'s notion of having received $f + 1$ prepare messages. However, we consider a replica as prepared only if it either completed to executing the request (i.e., if it has $f + 1$ $C - flags$ and the A-flag set as well, respectively only the A-flag for the no-crash case), or if it has timed out and set all error flags for the agreement flags that remained unset and if in this state it has set at least $f + 1$ P-flags. If replicas set a P-flag, the trusted *wo* memory implementation prevents them to also set the E-flag. It is important to require replicas to have timed out before considering them to be prepared in a state less advanced than all flags set that are required for execution since replicas need to independently reach the same conclusion whether or not a request should be processed.

Replicas execute those requests for which they find that $f + 1 - c$ replicas having prepared this request (Lines 70–74). They skip executing this slot if $f + 1 - c$ replicas have reached an error state from which they cannot later prepare it (Line 75). Since the leader's *wo* memory might have crashed, this request may reside as a copy in another replica's buffer. Lines 71–72 identify this request.

5.5.4 Checkpoints and Reset

Once all slots are used up, replicas have to reset the buffer before they can proceed. Without such a reset, slots, which now have flags set, would not be writable due to *wo* memory preventing overwrites. However, there are three inherent race conditions when resetting buffers:

1. A faulty replica may prematurely agree to reset the *wo* memories before the checkpoint is stable;

2. A replica may vote to reset a buffer that has just been reset; and

```
60    /* replica s_k */
61    on timeout or error:
62      for each slot y ≤ [x]
63        set all error bits for unset agreement bits(*)
64        poll buf_j[y] of other replicas s_j
65  *     let c be the number of \emph{wo} memories
66  *       that have crashed
67        wait until either f+1−c replicas have pre-
68          pared the request or f+1−c have reached
69          an error state with ≥ f+1 E-flags set
70        in the former case
71  *       identify request m such that m matches
72  *         the request in the buffers of ≥ f+1−c
73  *         replicas that have prepared this request
74          execute request // (ln. 46-51)
75        otherwise skip the slot by setting buf_j[y].AE
```

Figure 5.7: Error handling

3. A lagging, but otherwise healthy, replica may resume in a slot after the other replicas have reset all *wo* memories. In this case a faulty replica may exploit the lagging replica to replay an old request that the lagging replica was about to handle.

We avoid the first by requiring healthy replicas to first agree on a checkpoint and wait for this checkpoint to stabilize before agreeing to reset the *wo* memories. Checkpoints are written to write-once memory as well, using double buffering to always have a valid checkpoint in place. Checkpoints include a version number to denote which of the buffers holds the most recent checkpoint. Like for requests, *wo* memory prevents modification of completed checkpoints by setting a corresponding P flag (Line 57). Once a healthy replica detects $f + 1$ matching checkpoints, it agrees to reset the buffers in all replicas, including the now old checkpoint.

The second race is in fact an instance of the first, since without further precautions, agreeing to reset after the reset already happened translates into prematurely agreeing to the reset in the next round. We shall use the same mechanism to prevent the second and third race condition: We use one additional flag RF in the bitfields to denote that a reset has just happened. RF is checked when writing *wo* memory or when setting flags to prevent any modification of the tag-based *wo* memory device due to ongoing operations. Instead, these operations will fail, leaving the device in the state after reset, which allows the replica to recover from this situation. Moreover, RF is checked when agreeing to reset *wo* memory. The agreement is ignored when RF is set.

In consequence of the above, after each *wo* memory write or set flag operation and after reset in Line 57, the replica checks whether the device has just undergone reset

and reacts to this by clearing all RF flags, loading the most recent checkpoints and resuming from this checkpoint and an empty buffer. We have omitted these checks from the pseudocode for better readability. RF flags are the only flags that can be reset by the writing replica, but only by this one. As indicated above, the most recent checkpoint is the one that received $f + 1$ agreement and that has the higher version number of the two checkpoint slots.

5.5.5 Optimism

iBFT naturally leans towards the concept of optimistic protocols [Kap+12; DCK15], since passive replicas merely have to introspect the progress of active replicas to catch up. Likewise, introspection relieves replicas from assembling and transferring the progress they have made, which further simplifies exceptional situations like view change or the catch-up after replicas have been rejuvenated. For the latter, it is sufficient to reset to-be-rejuvenated replicas to a fresh, diverse instance, from which they catch up independently.

5.6 Experimental Results

We have measured our a cache-based emulation on x86 of the microcode-based[14] implementation of *iBFT* on an AMD Ryzen 7 3700X 8-Core CPU (2 threads per code) running at 2.2GHz (with 64GB RAM, 256KB L1I, 256KB L1D, 4MB L2 and 32MB L3 caches); and a tag-based implementation on a Zync-7 ZC702 FPGA configured with 3 Microblaze cores (running at 50MHz) and AXI busses to connect to our tag-mem devices and the BRAM block.

Our measurements focus on two scenarios: agreement with all replicas participating and catch-up with one replica remaining unresponsive while the remaining replicas reach agreement to then catch up with the progress they made. Replicas do not write checkpoints or wrap around buffers in this scenario. We evaluate both *wo* failure by crashing and the case where no further guarantees are provided in case *wo*-memory fails.

5.6.1 Implementation

The emulation of cache-based *wo* memory modifies *wo* bitfields with atomic *or* instructions (`lock; orq`) and *wo* strings with atomic compare exchange instructions (`lock; cmpxchgq`), which check for $\sim 0UL$. The implementation always writes the complete string buffer for a single slot to prevent faulty replicas from appending to

[14] A full microcode implementation would require access to the full x86 Intel microcode, which we do not have.

shorter prefixes. Reads are through arbitrary instructions. The emulation described above exhibits correct performance characteristics, but does not prevent writes through other instructions or unaligned writes with the above instructions. This behaviour can be easily retrofitted through microcode instructions.

We evaluated performance on the cache-based x86 emulation, with up to $n = 2f + 1 = 13$, to tolerate up to $f = 6$ faults. On the other hand, on the hardware tag-based *wo* memory implementation, we evaluated exclusively the setting $f = 1$ due to FPGA resource constrains[15].

As baseline, we have implemented the seminal hybrid BFT protocol MinBFT [Ver+13] over the same communication medium that we use for *iBFT*, but without any restrictions. That is, we use cachable shared-memory buffers on x86 and a shared BRAM block for Microblaze[16]. We have inlined MinBFT's trusted-trustworthy component — the USIG — into the C++ code to not introduce further overheads. For the same reason, we omit client signature generation and validation. That is, the primary overhead of MinBFT origins from HMAC computation and validation, which *iBFT* does not require. The choice for comparison with MinBFT relates to its high efficiency and state-of-the-art reputation in hybrid BFT protocols.

5.6.2 Performance Cache-Based Implementation

All figures plot the mean latency of request handling in cycles as experienced by clients (i.e., the time between issuing a request and receiving $f + 1$ matching responses) (y-axis), for an increasing number of tolerated faults (x-axis).

Figure 5.8 shows the time to agreement (Scenario 1), i.e., normal case execution, whereas Figure 5.9 shows the two cases of Scenario 2, that is, normal-case operation of $n-1$ replicas and time for the late replica to catch up. The figures identify the graph bars corresponding to the *iBFT* cache-based version of *wo* memory on x86 in the situation where *wo* memories can crash, side-by-side with the corresponding tag-based hardware implementation (which we shall discuss next). These results are as well compared with a shared memory-based implementation of MinBFT.

Catch up in MinBFT is implemented as the lagging replica receiving the messages sent by the other replicas (by reading their message buffers) and processing the request as usual.

As can be seen, *iBFT* is roughly 10 times faster than MinBFT when reaching agreement (16, respectively if *wo* memories do not crash), which we attribute mostly to the

[15]Note that we refer specifically to Zynq ZC702 resource constraints, where we could only instantiate up to 4 MicroBlaze cores plus the corresponding tagged memory devices, block memories and AXI interfaces, bringing the maximum possible f to 1 ($n = 3$). Other, modern FPGA boards will allow for more replicas to coexist.

[16]To be precise, we use the very same architecture for both settings, but access no tag-mem devices in MinBFT and use the shared BRAM block in *iBFT* only for storing measurements.

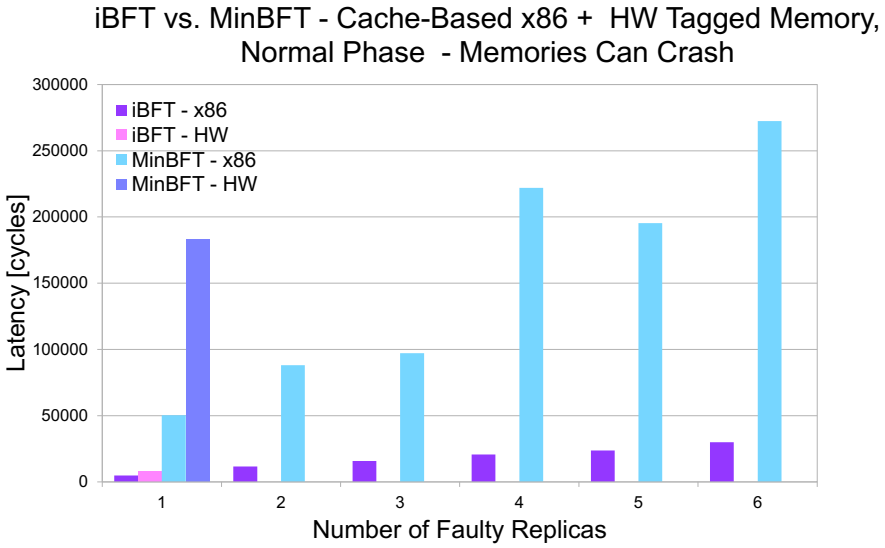


Figure 5.8: Latency of normal-case operation (in cycles), comparing cache-based and the tag-mem variant of *iBFT* against MinBFT on the same platform. *wo* memories can crash.

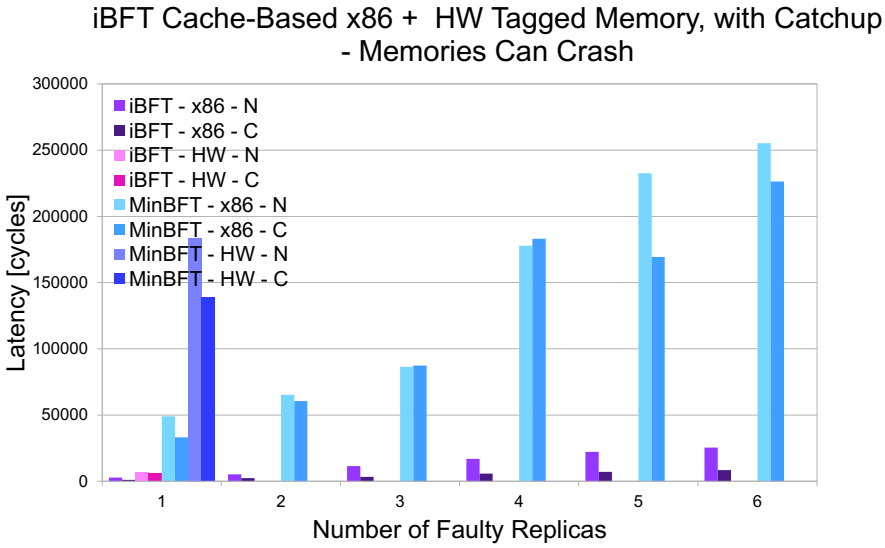


Figure 5.9: Latency of normal case operation (N) with one late replica and catch up (C) of this replica. *wo* memories can crash.

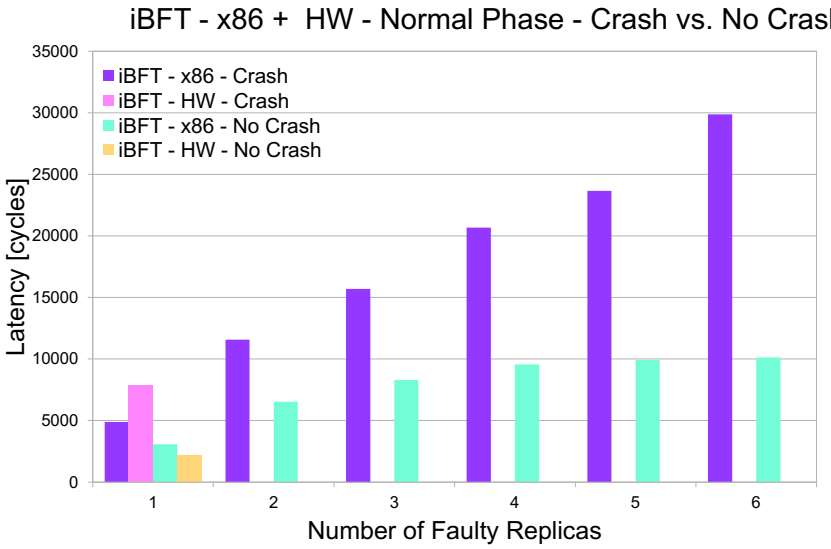


Figure 5.10: Comparing normal case *iBFT* when *wo* memories can crash vs. when they are assumed not to not crash.

costs of HMAC computation and validation, but in a significantly smaller part also to the larger message sizes that origin from having to transmit up to two HMACS (for commit). The optimization of *iBFT*, which allows lagging replicas to catch up to the progress of the leading ones, proved effective, by requiring only 1019 cycles on average for $f = 1$ (324 respectively for the no-crash version), with a linear increase for higher f.

We consider also a model where *wo* memories do not crash. Figures 5.10 and 5.11 represent a comparison of both environments: where memories can crash and where memories do not crash.

In an environment with no *wo* crashes, the reader may notice a stabilization of the latency with the increasing number of replicas participating. The ratio of reads (to introspect peers) versus writes (to update replicas' own state) increases. The cache coherence protocol executes these reads in parallel, which leads to the smoother slope in the graph. Cross hyper-thread[17] pre-fetching further improves performance.

It is also relevant to mention latency numbers can slightly vary depending on which replicas are late. Since replicas can proceed once they find $f + 1$ occurrences of the information sought after, and since introspected replicas start sequentially from the replica with the lowest ID to the one with the highest, if there is no late replica in the first $f + 1$,

[17]Cores can support hyper-threading, the implementation of which would be assumed part of the RCB in our cache-based variant. In our second variant, the core as a whole is considered the fault containment domain. That is, even though the core may have multiple hardware threats, there can only be one replica on this core.

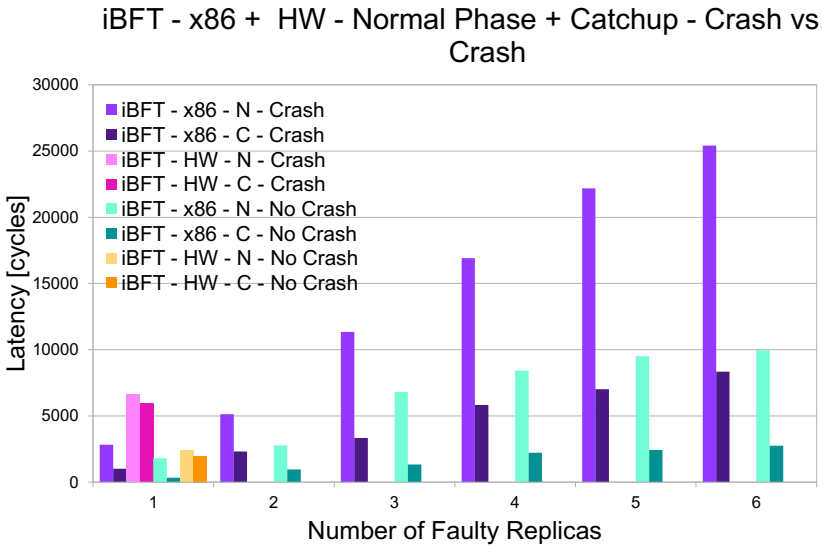


Figure 5.11: Comparing normal case *iBFT* plus catch up when *wo* memories can crash vs. when they are assumed not to not crash.

latency will not be affected by non-consecutive reads of late replicas' state. For the shown evaluation we let the late replica always be the one with highest ID, meaning it does not interfere with normal-case operation. Giving late replicas low IDs would slightly increase latency by a few cycles corresponding to introspecting the late replica.

5.6.3 Performance Tag-Based Implementation (FPGA)

For the following discussion, let us notice that writing a tag-mem device register / a word in the shared BRAM block requires 65 cycles. This corresponds roughly to the time required to reach the shared cache (L3) on x86. This translates into 99 cycles for setting flags and 106 cycles for reading.

Like above, Figs. 5.8 and 5.9 show the performance for the two scenarios (normal-case only and normal-case plus catch up) for *iBFT*, and Figs. 5.10 and 5.11 represent the no-crash case.

As can be seen, the previous results from the cache-based *wo* memory implementation are confirmed. With a factor of 23.24 (83.09 respectively if *wo* memories do not crash), *iBFT* is, on average, almost one order of magnitude faster than shared-memory MinBFT and almost two orders of magnitude faster in the no-crash version. However, the percentiles are much closer to the average times (see Figure 5.12), which on one side is due to the higher determinism of tightly-coupled memory accesses over coherent caches, with frequent bounces when polling shared data for state changes. However, it also indicates higher best case costs due to the inability to cache state. Catch up remains

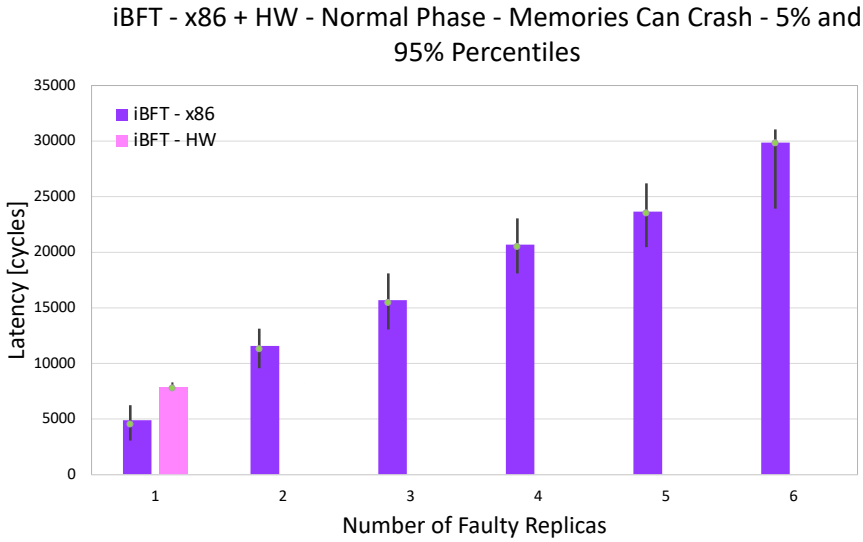


Figure 5.12: Mean values (bars) together with the 5% and 95% percentiles for both versions of the *wo* memory in *iBFT*.

relatively fast with a 5927 cycle latency on average (1972 respectively for the no-crash version).

5.6.4 Code Size and Hardware Utilization

Table 5.1 lists the C++ lines of code (LoC) for *iBFT*. Also shown are the VHDL source lines of code for the logic alone and the overall design (logic plus AXI interface) of the tagged memory device and the C++ lines of code for interfacing with this device. The reset device LoCs are minimal and, thus, not shown.

Table 5.2 shows the FPGA resources of the (post-synthesis) implementation of tagged memory and of the reset device. LUTs are units with no state, used to implement the combinatorial logic; while registers hold state, e.g, to keep buffer contents, but implement no logic. Each F7 Mux (wide multiplexer) combines the outputs of two LUTs together, while F8 Muxes combine the outputs of two F7 Muxes.

5.7 iBFT Discussions

We dedicate this Section to the discussion of *iBFT*-related details, clarifications and enhancements (such as the trusted copy operation discussed in Section 5.7.7).

iBFT	LoC Crash	No Crash
Common Definitions	49 C++	49 C++
Agreement (Normal Case)	404 C++	278 C++
Error Handling	154 C++	100 C++
Recover from Checkpoint	83 C++	83 C++
Total:	690 C++	510 C++
***wo* memory**		
Tag Mem SW Interface	156/207 C++	156/207 C++
Tag Mem HW	64 / 97 VHDL	64 / 97 VHDL

Table 5.1: Code size in lines of C++ and VHDL code. For the tagged memory (Tag Mem) interface, we separate the hardware-based (HW)/cache-based (SW) implementations; and for the tagged memory hardware implementation we separate logic/total (logic plus port declaration).

	Tagged Memory		Reset Device
	wo/ AXI	w/ AXI	w / AXI
Slice LUTs	2339	2372	58
Slice Registers	2144	2400	145
F7 Mutexes	803	803	
F8 Mutexes	286	286	

Table 5.2: Top table: FPGA resources required by the hardware-based tagged memory implementation (without / with AXI interface). Bottom table: FPGA resources required by the hardware-based reset device implementation. The few lines of code are implemented directly on $n = 3$ AXI interfaces, thus we present here the total numbers of AXI code plus custom reset code. The values presented denote resource utilization for each interface.

5.7.1 Performance

The latency for *iBFT* (in both implementations of *wo* memory) clearly demonstrates the benefit of constructing hybrid BFT-SMR protocols specifically for tightly coupled systems. The value of introspection is confirmed, in particular when replicas have to catch up to the progress of their peers.

Of course, we are naturally introducing overhead in comparison to non-replicated operation, but with added fault tolerance and resilience. Considering the costs of fully replicating the whole system (e.g., ECU), *iBFT* offers a safety advantage without greatly increasing replication costs. This is a performance/safety trade-off. As seen in the comparison with MinBFT, if one considers the added cost of the required cryptographic operations plus the rest of the protocol, in order to achieve the same safety result, *iBFT* greatly accelerates consensus.

As for the presented versions, although a direct performance comparison between cache and tag-based *wo* memory is not possible, we would like to emphasize the significant RCB reduction when opting for the second solution, even in otherwise cache-coherent systems, in particular when core isolation can be enforced without having to trust caches or their coherence logic.

Finally, as we expected, the hardware overhead, in terms of resources, of tag-based *wo* memory is dominated by the resources required for the tightly-coupled memory (i.e., the BRAM block itself). Costs for the tag-mem device, as shown in Table 5.2 are negligible. We therefore propose to augment on-chip memories, with tag-based *wo* memories to enable *iBFT* and similar algorithms that would benefit from write-once behavior.

5.7.2 Equivocation

In the setting discussed in [CJKR12] the impossibility result regarding non-equivocation and transferable authentication (discussed in Section 3.1.2) applies since faulty replicas may change the information stored in memory before, while or after it is read and, without synchronizing writes with reading operations, they may do so to deliver some information to one group of replicas, while conveying different information to others. It is, therefore, impossible to distinguish a scenario where the sender of a message falsely sends (i.e., writes) some information from one where the receiver (i.e., reader) modifies it. *iBFT* circumvents the above impossibility by not relying on fault diagnosis for reaching agreement. This problem is solved by relying on architectural hybridization and the introduction of trusted-trustworthy mechanisms as described by the fault model in Section 5.2. Assuming at most f replicas are faulty and given replicas only have write access to their own write-once memories, we can ensure replicas stick to their votes.

5.7.3 Write-Once Memory Pitfalls

Let us discuss some of the pitfalls *wo* memory and introspection provide for BFT-SMR protocols and how *iBFT*overcomes them.

The first aspect an attentive reader might wonder about is why *iBFT*is a hybrid protocol. In a homogeneous protocol without transferable authentication, faulty replicas may continue to lie inconsistently about error states, reporting for example to one replicas that the request got executed and to the other that the request is not yet prepared.

This dilemma remains even in the presence of *wo* memory in the following two aspects: First, without tri-state flags and *wo* memory enforcing that acceptance and error cannot be set simultaneously, a faulty replica could indicate acceptance to one healthy replica, making it believe that the request has to be executed, to then set also the error flag, revealing itself as faulty and making it impossible for another healthy replica to believe that the request should have been executed.

The second aspect occurs when a healthy replica keeps lagging behind while another healthy replica experiences a timeout. Because of the impossibility to distinguish late from faulty replicas, healthy replicas can only wait for up to $n - f$ replicas before it would need to proceed with error handling. Since in *iBFT*, we aim at reaching a conclusion whether to execute or skip a slot without coordinating with other replicas (e.g., in a view change), situations might arise where less than $f + 1$ of these replicas have prepared the request. In this case, the only conclusion a replica may take is to skip the slot. However, the lagging replica may later prepare the request and tip the global state over this threshold, which consequently indicates execution. We avoid this by *wo* memory triggering part of the timeout operation also in the other *wo* memories once they find $f+1$ replicas to have set their timeout or execution indicating A-flag (to either accept or the corresponding error state).

Crashing *wo* memories cause a further difficulty of this sort. Some replicas may find the memory not yet crashed while others will see it crashed at a later stage[18]. *iBFT*recovers from crashed memories exclusively during error handling. Our timeout mechanism ensures that before a replica starts processing timeouts (i.e., after $f + 1$ replicas have timed out as well or already executed the requesst) that all unset flags of all *wo* memories for this state are set to their error state. Therefore if one replicas sees the request prepared with up to c *wo*-memories crashed (i.e., $f + 1 - c$ replicas have prepared this request), then the same is true if another replica finds $c' > c$ *wo*-memories crashed (i.e., it sees the same replicas having prepared the request minus possibly the $c' - c$ replicas whose *wo* memories crashed as well).

[18]Thanks to an anonymous reviewer for spotting this pitfall.

5.7.4 Leader or Leaderless?

In *iBFT*, the leader provides total order for the client requests. A leaderless approach is possible if the assignments of client requests to slots is deterministic. Without a leader, replicas would still have to agree on an order or rely on some secondary ordering mechanism to do that for them. Having a rotating leader simplifies agreeing on the sequence of requests and their parameters.

5.7.5 Safety and Liveness

iBFT, as described in Figures 5.6 and 5.7, is safe and live in the sense that only those requests are executed for which a quorum of $f + 1$ replicas has reached consensus (in the agreed upon order and with the additional notion that if one healthy replica executes a request then all do so) and that the protocol makes progress in phases where the environment exhibits sufficient synchrony. Let us provide a formal argument for these facts.

Safety: It is easy to see why the base *iBFT*protocol is safe under a fault model where no guarantees are made once a *wo* memory crashes. Once proposed, the leader can no longer change its proposal. So for the same slot, either all healthy replicas will confirm the leader proposal (by setting their P-flag in Line 29) or set their E-flags when they time out. Therefore, because the *wo*-memory timeout mechanisms sets unset E-flags after $f + 1$ replicas have either executed or timed out, timeout handling replicas either find $f + 1$ replicas to have prepared the request (i.e., $f + 1$ P-flags set) or $f + 1$ replicas that have not (and due to the timeout mechanism) will not in the future execute the request. Before timing out, healthy replicas will not execute a request unless it is prepared. Replicas cannot change their flags once set. Therefore, a request will be executed by all healthy replicas if and only if it is executed by one healthy replica. The deterministic order follows from the way replicas proceed through slots and from the fact that reset prevents replicas from proceeding without first clearing the RF-flags.

To ensure safety in case *wo*-memory crashes, we have to see why replicas will only find valid requests in the matching $f + 1 - c$ non-crashed *wo* memories of replicas that have prepared the request. If $c = f$, then only healthy replicas remain with non-crashed *wo* memories, who trivially fulfill this condition. If $c < f$, then $f - c$ faulty replicas may have forged a request when copying from the leader or from the client. But then for the request to be prepared, a healthy replica would need to have $f + 1$ P-flags set, which it will only do if it can confirm that the faulty replica has copied the proposal correctly. Like above, the fact that a healthy replica executes a request if and only if all healthy replicas do so follows from the insight that during normal phase operation, no healthy replica will execute a request unless it finds $f + 1$ replicas having verified the copies in f other replicas.

Liveness: Liveness during "good" phases of bounded computation and communi-

cation, follows from the fact that at no time, healthy replicas wait for more than $n - f$ replicas. During error handling, where replicas wait in Lines 68-69 for either $f + 1 - c$ replicas to have prepared the request or for $f + 1 - c$ to have reached an error state where they have all unset E-flags set. The *wo* memory timeout mechanism fulfills this condition, latest after $f + 1$ replicas have their A-flag or the corresponding E-flag set. Therefore, eventually the system will reach a state where replicas experience synchrony long enough to process the last timeout their experienced before this situation, caught up to the same slot and executed a slot for which a healthy leader proposed a request by a correct client, skipping faulty ones.

5.7.6 Why is Homogeneous Consensus Unfeasible?

As discussed in Section 3.1.2, without synchronizing all introspection operations, a third replica can no longer distinguish whether the originator has produced a wrong message at the time it was introspected by a replica which then correctly accuses the originator as being faulty, or whether the introspecting replica has wrongly accused the originator. In particular, this third replica cannot check the originator buffer for this information, because by the time it is introspecting this buffer, the originator may already have replaced the message with a correct one.

Synchronizing introspection would allow all introspecting replicas to see the same message. However, the hardware overhead and complications entailed with synchronizing access to the buffers outweigh any benefits obtained from *Introspection* itself; not to mention the possibility of faulty replicas stalling its peers if locks are used to enforce simultaneous and exclusive reads.

From the above, we further see that there is no full replacement for commonly applied message authentication techniques such as signatures or MAC vectors. Unfortunately, proofs of agreement, by chaining certificates of the above kinds from a quorum of different replicas, cease to work in introspection-only protocols (i.e., without architectural hybridization and trusted-trustworthy mechanisms), as do certificates for checkpoints.

5.7.7 Trusted Copy Operation

iBFT does not, however, solve our safe platform reconfiguration problem by itself. In order to safely reconfigure privileges and update critical data, a trusted copy operation is required to copy the agree-upon request from the replicas' local write-once memory to the designated memory zone. Essentially, in addition to consensus, *Midir*'s T2-H2 voters offer in-place updates by writing the agreed upon value to the intended location. *iBFT* reaches consensus out-of-place in the replicas' write-once memory buffers. To apply these consensual decisions, we therefore have to introduce a trusted operation to

```
1  TC tc_i:
2    for slot l = 0 to max_slots
3      if l marked ready to execute on f+1 replicas
4        dest.[req.m.addr] := buf_l[x].req.m.data
5      else
6        l+1
```

Figure 5.13: Trusted Copy Operation.

allow replicas to copy the agreed upon value to the agreed upon address, unchanged and exactly once.

Naturally, such a copy operation would need to be implemented by some trusted-trustworthy mechanism, either in hardware (as tagged memory) or in microcode (cached-based version). Introspecting replicas' *wo* memory, this mechanism, upon finding $f+1$ matches on ready-to-execute marked requests, would then copy the agreed-upon result to the destination address, much like T2-H2 forwards it through the network to the final location. Figure 5.13 lists the simple pseudo code for this procedure. An atomic operation must check the ready-to-execute flag on $f+1$ replicas for a given slot l (since a faulty one may just deliberately mark some request as agreed upon without actually achieving consensus) and write the data associated with that slot in the appropriate destination denoted by $req.m.addr$, a piece of information embedded within request m itself. The copy must be made in the order found in the slots, as such, if a request for a certain slot did not reach agreement, it is skipped and the next slot is checked. Figure 5.14 illustrates the trusted copy mechanism, copying an agreed-upon request to the designated destination address.

As with *wo* memory and T2-H2, the implementation of the trusted copy is simple enough to be trusted not to fail and to be easier to verify, however, it can be made redundant for continued operation in the case of a crash and the memory itself (plus the memory controller) may as well have some form of redundancy depending on the desired fault model. Recall that the failure of a *wo* memory or a T2-H2 simply means the associated replica is now considered faulty. The trusted copy is not implemented for each replica, but instead an instance that collects results. Therefore, its level of redundancy is not dependent on the value of n.

5.7.8 Remote Direct-Memory Accesses

Although this work focuses exclusively on manycore systems, our results apply equally well to coarser grain replica consolidation schemes, for example, by placing replicas on multiprocessor sockets and connecting them through buses, or by leveraging server nodes connected through a network supporting remote direct-memory accesses (RDMA). As replicas move closer together, new mechanisms become available through

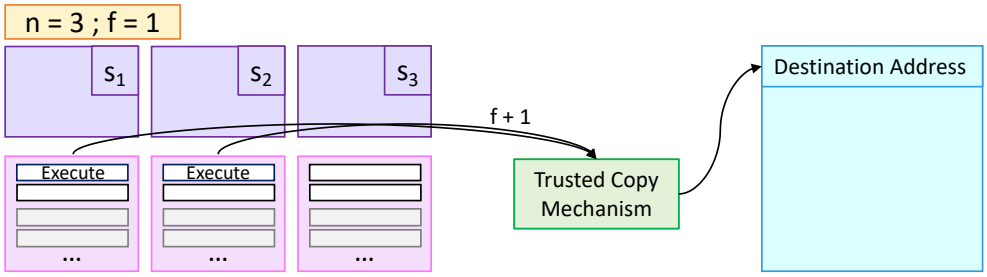


Figure 5.14: Representation of the trusted copy mechanism, copying an agreed-upon request to the designated destination address.

which replicas can act over their peers.

Chapter 6

Solutions Discussion

This Chapter will be dedicated to the comparison of our two solution, *Midir* and *iBFT*, and to a conclusive dive into persistent consensus and the successful elimination of SPoFs. We shall compare the two solutions as a whole as well as the trusted devices used by each.

6.1 iBFT vs. Midir

The choice of *Midir* over *iBFT* or vice versa pertains to a couple of trade-offs regarding RCB size, simplicity, overhead, cacheability, memory restrictions and MPSoC area. The study of both solutions envisioned the evaluation of these performance/reliability points.

RCB: As previosuly discussed in Chapter 5, *iBFT* variants themselves present RCB trade-offs. While the tagged memory implementation trades cacheability for a reduced RCB (at the cost of more expensive *Introspection* operations), trusting the tagged memory device, its BRAM, and a reset device (and the trusted copy operation, if used); the microcode-based variant must rely on the correctness of all cores, caches and the cache coherence logic, leading to quite a large RCB in terms of necessarily trusted hardware. *Midir* on the other hand, requires solely the T2-H2 device to be trusted.

Simplicity: *iBFT* relies on simpler trusted-trustworthy mechanisms than *Midir* does, which in turn leads to greater performance. In other words, *iBFT* further accelerates consensus and is an optimization over *Midir* on that parameter. The *wo* memory is more concise and much simpler logic-wise than T2-H2. An easy way to make such a comparison, is to analyze the inner workings of the tagged memory version of *iBFT* vs. T2-H2 as they are both hardware-based. While tagged memory merely checks if a bit (ready flag) is set to decide whether or not to let a write through, T2-H2 must perform access control via the capability registers, collect votes, check their sequence number, wait for $f + 1$ matches, forward the agreed upon request and check for reset votes.

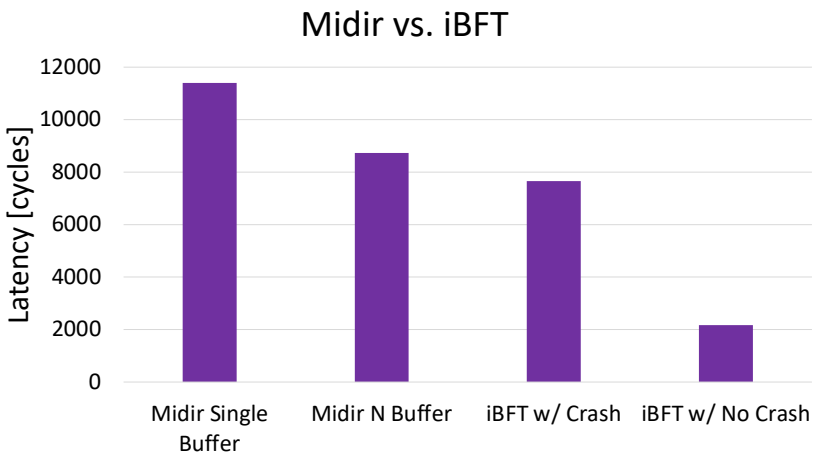


Figure 6.1: Comparison of the normal case phase hardware FPGA implementation's mean cycle count of both variants of the *Midir* voters (single buffer and N buffer) and both fault model cases of *iBFT* (the case where *wo* memories can crash and the one where they are assumed not to crash).

Overhead: Figure 6.1 compares the normal case phase hardware FPGA implementations (mean cycle count) of both the single buffer and N buffer variants of *Midir*, in the per-replica capability space context studied in Section 4.6, and both fault model cases of *iBFT*, the case where *wo* memories can crash and the one where they are assumed not to crash. The fairest way to compare *iBFT*and *Midir* in terms of overhead is to compare the results of the fastest *Midir* version, N buffer, with the version of *iBFT* with the same fault model, i.e., when memories are allowed to crash, given the T2-H2s in *Midir* are as well allowed to fail by crashing. In the case just described, *iBFT* beats *Midir*'s performance by approximately 1068 cycles. This is observable mostly due to simplicity and cacheability. However, if we compare *Midir*'s N buffer variant with the case where *wo* memories crash, then *iBFT* is around 6555 cyles faster. Recall that *iBFT* is safe whether or not the *wo* memories crash, the protocol just gains higher complexity if they are allowed to.

Additional Votes: *iBFT* gives us the system call log directly when voting, as requests are stored in slots up until all slots are used, which is dependent on memory size, and a checkpoint must be taken. It, however, requires a trusted copy to transfer agreed upon votes elsewhere. *Midir*'s T2-H2s, on the other hand, require voting on inserting entries in the system call log, which is an independent data structure in consensual memory, outside of the voting space.

Cacheability: Voters' contents cannot be cached, leading to poorer performance in comparison to *wo* memory microcode-based implementation, thus behaving similarly,

in that regard, to the tagged memory version.

Memory Restrictions: *Midir*'s voters present a memory restriction. As they are designed in hardware, buffer size and quantity is determined at design time, making the available space for voting limited. *iBFT*, on the other hand, resolves voting in memory, with the trusted device merely mediating the success of write operations.

6.2 T2-H2 vs. Write-Once Memory

The most obvious way in which *wo* differs from T2-H2 has to do with the consensus buffers' location within the MPSoC. While the former opts for an in-memory approach, with a trusted-trustworthy device mediating write accesses to memory; the latter instead places this buffers within the T2-H2 voters themselves.

Another difference lies in the functionalities offered by each: while T2-H2 yields in itself trusted-trustworthy access control and voting, *wo* merely protects the voting space by means of trusted-trustworthy tri-state bitfields.

Additionally, T2-H2 is constructed in a way that the reconfiguration voter's output is directly connected to its own access control/privilege reconfiguration interface, ensuring updates to this information are done exclusively through agreement of a majority of replicas. Also, the access control capabilities therein provide fault isolation for that tile. On the other hand, *wo* by itself does not provide fault isolation or privilege reconfiguration and, instead, needs to be aided by the trusted copy operation discussed in Section 5.7.7. Nevertheless, *wo* incurs less overhead than T2-H2, given its greater simplicity.

6.3 Persistent Consensus

One of the requirements established for the successful construction of a SPoF-free D-MPSoC was the guarantee of persistent consensus, i.e., the ability to allow replicas to catch up if late/lagging, always ensuring enough replicas know about the requests agreed for execution. Although the ways to tackle this requirement were addressed for both solutions throughout the description of their protocols, we summarize here the main points for each.

iBFT tackles this both through *Introspection*'s nature and with the way checkpoints and resets are handled (see Section 5.5.4). *Introspection* makes it possible for a late replica to introspect other at any time and catch up with their progress and checkpointing requiring healthy replicas to first agree on it and wait for it to stabilize before agreeing to reset the *wo* memories. In addition, the RF flag stops a lagging replica from resuming in a slot after the other replicas have reset all *wo* memories. Hence, the availability and consistency of consensus information does not waver.

Midir ensures this same property via the *syscall* and *error* logs, and through careful handling of sequence numbers in subordinate votes. The *syscall log* records agreed upon system calls and the *error log* error information to protect it from getting lost if the voter is reset prematurely. Thus, even if a replica misses one or more votes and, thus, cannot read the consensus data directly from the voter itself, it can always know precisely what happened during the agreement phases it missed.

6.4 How is the SPoF Eliminated?

As discussed earlier, *Midir* and *iBFT*equipped with a trusted copy operation both manage to resolve all D-MPSoC requirements to address the main problem we set to solve in this thesis: eliminating all software low-level SPoFs. In essence, this is accomplished by combining techniques from the state of the art, such as on-chip tile isolation, capability-based access control and voting; with ways to safely reconfigure privileges and, thus, resource management, in a fast enough manner, without relying on a trusted underlying software layer, tasked to implement the fault containment and communication that resilience protocols require.

Midir's T2-H2s, being the trusted-trustworthy devices that represents the core of the architecture and means through which critical operations are handled, are trusted not to fail. Our threat model in fact assumes T2-H2 to fail only by crash (due to its simplicity), in which case we consider the replica failed. Similarly, *iBFT* is already prepared to consider both cases where the write-once memory is (i) trusted to not fail at all or (ii) trusted to fail only by crashing (as supported by its simplicity) up to the threshold f. Additionally, T2-H2's and write-once memory's low complexity makes them easy to formally verify, specially in comparison to micro-kernels and micro-hypervisors.

Chapter 7

Resilience

In this Chapter, long-run resilience will be discussed in the context of *Midir*. Although the rejuvenation techniques mentioned below can and should be applied to *iBFT* as well, for the sake of simplicity, and also due to the more robust nature of the former, we shall focus solely on *Midir*'s rejuvenation and tile relocation.

As long as no more than f replicas become compromised, voting on critical operations prevents harm, while mandatory consensual privilege change puts a stop to faulty replicas breaking out of their fault containment domains: the tiles on which they execute. However, over time, any healthy majority may get depleted, asking for rejuvenation. In our case, rejuvenation must be done over bare metal, that is, without having to rely on the correctness of software in an underlying infrastructure. Rejuvenation is the process of returning suspected or proven faulty replicas into a healthy state that is sufficiently diverse from the states adversaries have already analyzed. Without any underlying infrastructure, rejuvenation translates into performing a sequence of critical (and therefore voted upon) operations that conclude in booting the rejuvenated replicas into a fresh image. From there, replicas access the shared syscall log to catch up with the others. The voted operations are:

1. Stop all execution in the tile to prevent compromised code from manipulating the new binary while booting;

2. Copy the boot trampoline code to the tile[1];

3. Equip the tile with capabilities to read the new binary;

4. Reboot the tile (to roll forward into the new state).

[1]This step is required if the booted core in a tile cannot fetch instructions through capabilities, but only from tile-local scratchpad memory.

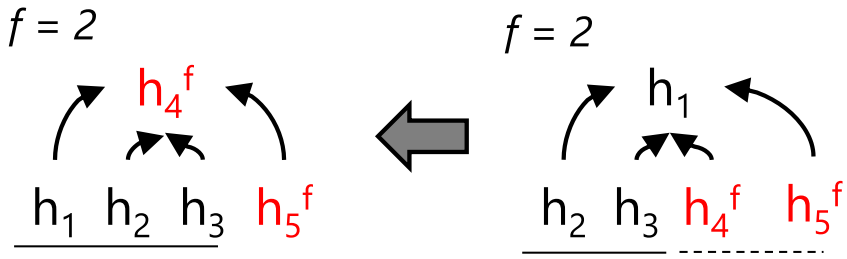


Figure 7.1: Possible configurations when rejuvenating proven faulty vs. suspected faulty replicas (shown for $f = 2$).

7.1 Restoring Synchrony

Rejuvenation needs to outpace adversaries in compromising more than f replicas. Clearly, this is not possible in an asynchronous model [FLP85]. To obtain synchrony, replicas enter a reduced functionality mode (akin to "exception handling") where they stop executing client requests, disable interrupts to avoid interrupt bursts, and only focus on rejuvenation. Reducing their functionality, limits their threat surfaces and how adversaries may interfere, but also the performance replicas are able to achieve. Therefore, latest after some cool-down period, tiles will start to exhibit known bounded execution times for rejuvenation, however, at costs that likely will not be sustainable for normal operation. Prioritizing rejuvenation traffic at the NoC and switching to deterministic arbitration leads to similar performance/predictability trade-offs and known bounded communication delays. We set timeouts larger than these bounds to identify non-responsive replicas as faulty.

7.2 Rejuvenating Proven vs. Suspected Faulty Replicas

Figure 7.1 illustrates the problem we have to solve when rejuvenating up to $k \geq f$ replicas at a time with less than $2f + 1 + 2k$ replicas, as suggested in [SNV06; Sou+10] as long as only proven faulty replicas are rejuvenated (left case), enough healthy replicas remain as rejuvenators to outvote the remaining faulty ones.

Rejuvenating k proven faulty replicas, $n - k = 2f + 1 - k$ rejuvenators remain of which at most $f - k$ are faulty; we have $f + 1$ healthy rejuvenators. Unfortunately, no such proof will be available when rejuvenating replicas proactively. In this case, a situation similar to the one on the right may occur, where, out of the $n - k$ rejuvenators, f replicas are faulty. To resolve it, we have arranged for the rejuvenation steps to use double buffering to not be destructive until we reach the final Step 4 in which we reboot to the new image. Therefore, we can always resume executing the suspected (but not

proven faulty) replica in a state where it left off. Now, the only way for a replica to not get rejuvenated is if there is disagreement in any of the votes for one of the four steps. But then, together with the vote of the suspected replicas, we can diagnose the cause of this disagreement to proof that one of the disagreeing replicas is faulty. Once diagnosed, we first reactively rejuvenate this proven faulty replica before returning to proactively rejuvenating the suspected. Notice, that we have to repeat all four steps and that the image is used up if the first rejuvenation failed between Step 3 and 4. Switching the system to synchronous mode for rejuvenation ensures that the votes for Step 1–4 will not repeatedly timeout and that therefore the above diagnosis will become possible. What remains to see is why all replicas will be rejuvenated, even if faulty replicas may conceal themselves by exposing other replicas when pushing error state. First, proactive rejuvenation will be triggered periodically for all replicas in groups of size k and healthy replicas will persist rejuvenating these replicas until all are rejuvenated (or relocated). Therefore, the only way to prevent this rejuvenation is by causing subordinate votes for the above steps to fail. But then, we obtain proof of a faulty replica, which we reactively rejuvenate to escape the worst-case situation of f healthy vs. f faulty rejuvenators. Once there, we can roll forward the proactive rejuvenation of the k replicas. The remaining faulty replicas may still cause voters to get suspended, but they cannot prevent the votes from succeeding (latest after $f - 1$ retries).

7.3 Diversity and Replica Pool

We have already seen that it is necessary to present adversaries a moving target. To achieve this, we exploit privilege reversion and decouple replica generation from use, by managing a pool of unused replicas (see Figure 7.2). A possibly replicated local application, to which none of the kernel replicas has read or write access, fills this pool either by decrypting images from an external source or by creating them with the help of a local diversifier. The kernel grants this application write access to the next free buffer without ever giving kernel replicas read or write access. It revokes this write access once the image is prepared. Only in Step 3 of the rejuvenation procedure, when equipping the to-be rejuvenated replica with capabilities to a new binary, will the other replicas engage in a vote to grant this replica and only this replica read access to the next unused image in the replica pool. This image is then used up in the process of rejuvenating the replica. It may happen that the votes up to Step 3 succeed, but a faulty replica reveals itself in Step 4 before rebooting the to-be rejuvenated replica. Since in this case the replica rolls back to its previous state, the image must be dropped and a fresh image chosen when returning to the rejuvenation of this replica. Otherwise, if $f > 1$, both the replica that failed the reboot and the to-be rejuvenated replica could be faulty, which would give the adversary the possibility to analyze the image and thereby reduce the time to compromise this replica.

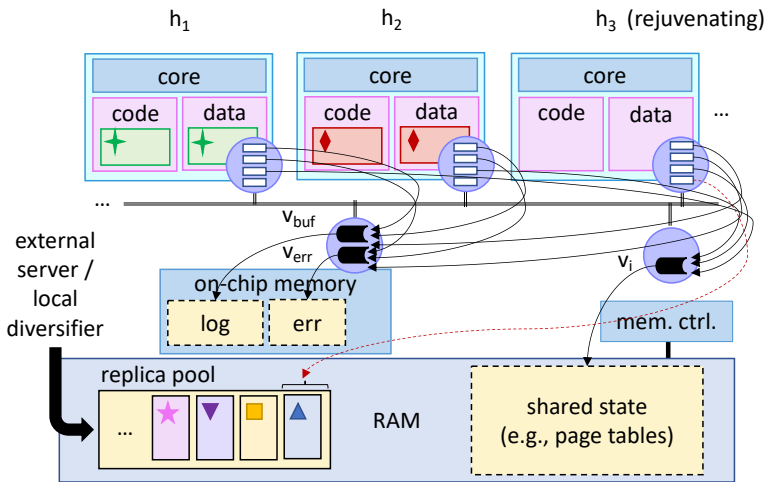


Figure 7.2: Rejuvenation of replica h_3. To create a new, sufficiently diverse replica, h_1 and h_2 provide h_3 with a capability to the next fresh image in the replica pool. The remaining images remain inaccessible until they are required.

7.4 Relocation

Relocation becomes necessary if a tile fails persistently due to permanent hardware errors. We evade this tile and allocate a new kernel replica at another tile. The steps for relocation are the same as for rejuvenation, except that the old tile is stopped and the new tile rebooted. However, special care must be taken to not violate property 1 and allow the old tile to impersonate the new one in votes. We resolve this by implementing a trusted wormhole protocol [Ver06] between the involved T2-H2s, which replaces Step 4 in the rejuvenation sequence. The kernel replicas agree to silence the persistently failing tile (which is an operation at this tile's T2-H2 to clear all capabilities) and to initiate the reboot of the specified destination tile through that tile's T2-H2. This way, the two tiles are never active simultaneously. Since we assume Mon crashes are detectable, replicas can always reboot the new tile after detecting such a crash happening during the wormhole protocol. Tiles with a crashed Mon cannot invoke capabilities and in turn execute operations on external objects.

Chapter 8

Application-Level Use Case

Aside from the protection of low-level management software and their elimination as a SPoF, the target of this thesis, the solutions presented so far can as well have application-level uses. In the MPSoC scenario already studied here, different (replicated) applications may share critical data. However, applications are not necessarily trusted even if replicated, as emphasized by the discussion in Chapter 2. In this Chapter, we provide an example of how solutions such as *iBFT* or *Midir* can be extended for use at application level and for the protection of critical data shared by replicated applications that may represent a subsystem within the chip.

8.1 Data Structures for Critical Data Protection

We have addressed the consensual execution of critical operations in the context of dynamic tightly-coupled, manycore environments. We now turn our attention to the data kept and shared by application-level subsystems (interchangeably, applications or replica groups) and, assuming for our study the increasingly complex functionality of critical systems is realized by multiple subsystems working together on the same multi- or many-core system-on-a-chip (MPSoC), we investigate how such subsystems can share data and how we can apply the strategies devised to further protect critical data structures in general, building up on a specific example provided in Section 4.6, where a data structure was used to keep consensually-updated capabilities in memory. This issue is relevant both if a single replicated subsystem is accessing the data or if multiple communicating subsystems share this data and, possibly, manipulate it.

As we shall demonstrate in Section 8.1.1, classical multi-threaded applications need necessarily be correct, as one malicious thread is enough to corrupt a data structure kept in shared memory, meaning also that, as we have discussed previously, replication is not enough. The solution presented in this Chapter makes it possible to tolerate a minority of compromised application replicas. Only the application, as a whole, needs to be

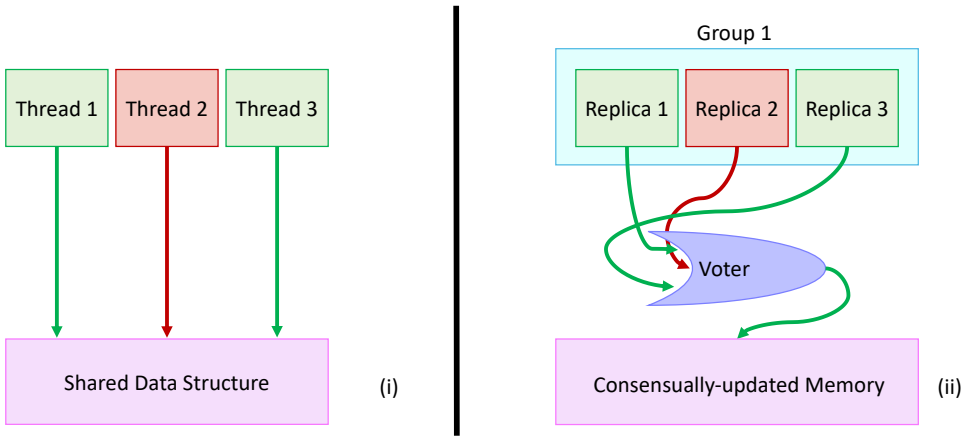


Figure 8.1: Shared data structure updated by multiple threads vs. consensual memory updated by a replicated application.

correct, as it shall be secured by a majority of correct replicas. Figure 8.1 illustrates this point, where in (i) threads manipulate the same shared data structure and in (ii) the memory where is resides is updated consensually.

Solutions have been proposed to carefully control when replicas can write individual elements of critical data structures [Gol84; CVP99]. However, the overheads of these solutions are significant and they necessarily involve OS functionality which, in consequence, has to be part of each replica's RCB.

As such, it becomes necessary to construct a complexity-reducing solution that not only tackles overheads imposed by previous solutions, but also enhances safety by significantly reducing the RCB. We must also establish the same level of autonomy over data structures in replicated subsystems as seen in regular multi-threaded applications, where threads query the structure on their own and often update it in a similar fashion, constrained only by the requirement to acquire one or more locks to ensure certain operations are executed in a mutually exclusive way. This parallels with the distributed systems' reality, where, following their replication and interaction pattern, operations on data structures can be provided as a replicated service, invoked by clients to query and update them.

There are, however, two major ways data structures can be shared. We shall describe and discuss the setting where i) a single read\write subsystem is used to manipulate local per-replica data structures, we call this managing subsystem the "owner group"; and ii) consensually-updated shared data structures. We extend the use of these data structures to multiple subsystems, where several applications interact with the same

data simultaneously (with provided synchrony).

With *Midir* we explored the consensual execution of critical operations on a BFT-like, fault-isolated manycore architecture by means of trusted-trustworthy hardware components; and with *iBFT* we looked at means of accelerating consensus, taking advantage of the tight-coupling on the manycore environment and using a write-once memory abstraction to prevent replicas from equivocating others (both by means of hardware components called tagged memory or a with microcode-based implementation). Concepts from both can be used to achieve our generic consensual data structures goal, however, we shall borrow from the mechanisms from the latter, as it has improved performance.

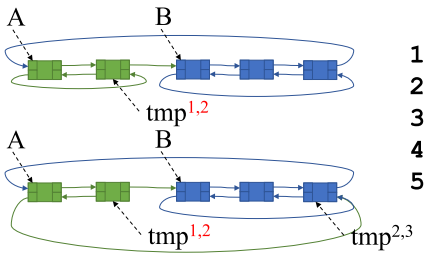
As demonstrated with *Midir*, there are performance and memory usage trade-offs when using (i) a read-shared per-replica memory region or (ii) a single, read-write shared and consensually-updated one with which all replicas interact with (see Section 4.6). We measured there the performance of capability installation in the setting of a private data structure in each replica (i), where updating the data structure meant voting to install the capability, to reply to the client (application requesting the hypervisor to perform the operation) and to mark the system call as finished; and in the setting of a read-shared, consensually-updated data structure (ii), trading off speed for a smaller memory footprint by introducing additional votes for track keeping. The choice is then application dependent.

In the construction of consensual data structures, it is tempting to search for solutions where consensus about individual updates is reached within the elements forming the data structure. For example, one could equip the next pointer of a linked list with bits set for all those replicas that have agreed to its value. However, such constructions require complex ownership management (and, in turn, costly privilege management) operations to prevent the value-proposing replica from modifying other parts of the data structure and followers of this leader from tampering with the agreement bits of other replicas or resetting their own after agreement has been reached. Instead, we choose to utilize *iBFT* mechanisms.

8.1.1 Motivating Example

Unlike what may be apparent at first, safely updating a data structure does not come down to voting and reaching consensus on write operations pertaining to said object. To further illustrate this statement, let us briefly discuss the pitfalls of consensual data structures on the example of appending cyclic doubly-linked lists, which we shall use as running example throughout the remainder of this chapter.

The `append` operation merges two lists A and B by putting together a list containing all elements from A and B, while maintaining their order. In case of cyclic doubly-linked lists (without head element), this implies dereferencing the `prev` pointers of the element at the head of A and B to obtain the respective tail element, and connecting the




```
1    A->prev->next :=c B
2    B->prev->next :=c A
3    tmp := A->prev
4    A->prev :=c B->prev
5    B->prev :=c tmp
```

Figure 8.2: Naive consensual append of two cyclic double-linked lists.

head of B to the tail of A and vice versa. The pseudocode in Figure 8.2 illustrates this operation, if we replace the consensual assignment $:=^c$ with normal assignment $:=$ and execute this code in a single thread.

Replicated execution is, however, prone to race conditions, such as the following, which we illustrate in Figure 8.2 by marking with superscripts what the individual replicas read and store in their local variable `tmp`. Assume replica r_2 is malicious and replica r_3 is correct but slightly delayed in executing the code presented in the figure. Replica r_1 and r_2 advance to Line 4, voting to consensually assign `A->prev->next`, `B->prev->next`, and `A->prev` to their respective values. In particular, they read `A->prev` as the last element of list A and store this result in `tmp`. The update $:=^c$ denotes such a consensual update, i.e., a state modification that reached agreement by a safe quorum in the system. Before executing Line 5, replica r_2 allows r_3 to catch up. However, because r_1 and r_2 already updated `A->prev` in Line 4, replica r_3 reads `tmp = B->prev`, the last element of list B. Consequently, without further precautions, r_3 will propose B's last element when reaching agreement in Line 5 on what to store in `B->prev`. Anticipating this confusion, the compromised replica r_2 may even agree with this proposal to consensually update `B->prev` to the situation shown in the lower part of Figure 8.2.

Notice that each individual vote was correct and that there is also no confusion about sequence numbers. Instead, the root cause of the race condition at hand is the write in Line 4, which modifies a value that still needs to be read by other replicas if they are late. We shall call these writes *destructive* for the data structure-manipulating subsystem.

8.1.2 Setting

Even though so far we have considered a single MPSoC system with internally replicated low-level software, for the purpose of this section, we shall extend our setting to the resilient interaction among multiple subsystems (applications) sharing data structures. We consider both bare-metal and hosted settings. For the latter, we assume that the operating-system kernel gives the abstraction of being correct and impenetrable by

adversaries through the usage of a solution like *iBFT* or *Midir*.

Let $\mathbb{G} = \{G_0, \ldots, G_m\}$ be the set of subsystems (replica groups) that share a consensual data structure. Each group $G_i \in \mathbb{G}$ represents an application τ_i in the original (non-resilient) system. Additionally, groups may have different replication degrees. Let $\mathbb{N} = \{n_0, \ldots, n_{max}\}$. Each group may have a number of replicas n_0 to n_{max}, depending on their criticality. We strive to tolerate that up to $f_i < n_i$ of the replicas in each group can be Byzantine (i.e., be faulty in an arbitrarily malicious manner). We denote the replicas of a group G_i by $s_i^1, \ldots s_i^{n_i}$ and write $G_i = \{s_i^1, \ldots s_i^{n_i}\}$. We assume groups have disjoint replica sets (i.e., $G_i \cap G_j = \emptyset$ for $i \neq j$).

Depending on the choice of a per-replica or a common, consensually-updated shared memory space, each replica has write access to a region of memory that can either only be modified by this replica or consensually modified by all groups. In case of the former, we assume this region to hold the per-replica copies of the data structure or its elements. For the latter, read/write shared consensual data structures, atomic operations in the form of compare and swap (CAS) are required to ensure updates are performed atomically when multiple groups simultaneously request an update. For this end, trusted-trustworthy memory controllers, such as *Azura* for *Midir* or an enhanced trusted copy operation for *iBFT*, will ensure concurrent updates happen atomically relative to each other and relative to concurrent reads, at the granularity of the vote. We shall further describe *Azura* and the enhanced copy in Section 8.1.6.

8.1.2.1 Trust Model

Replicas of one group trust the consensual decisions of all other groups. That is, if group g_i performs an operation consensually with agreement from a fault threshold f_i exceeding quorum of replicas, then the effects of this operation are assumed to be correct by all healthy replicas of all other groups.

The rationale for this system and trust model is as follows. Typically, applications operate on shared data structures with code of different complexity and criticality. For example, one pattern commonly found in real-time applications includes complex producers that are monitored by much simpler observers which, in turn, are responsible for verifying the timely creation of data items (e.g., the next control command) and for initiating countermeasures if no such command was received in time. Whereas in such a setting, more exploitable vulnerabilities are to be expected in complex subsystems, motivating higher fault thresholds f_i and replication degrees n_i for this group g_i, criticality may as well indicate higher replication degrees due to the consequences of failure on the safety of the overall system. By being able to operate with different such values for different groups, we retain the flexibility to optimize each group individually.

Also shown in the above example is the importance of different functionalities (e.g., complex control and safety monitoring) providing liveness and quality of service while retaining safety. Whereas individual replicas in the functionality-providing groups may

fail, the functionality as a whole must be provided for an operational and safe system, even if, depending on the application, more distinguished trust models are possible (e.g., failure of the complex control as long as shared data structures remain consistent to not propagate this failure to other subsystems).

Notice also that we explicitly include the possibility of applications that are comprised of several components, which, despite forming single-points of failure in the original, non-resilient instance, are sufficiently isolated to cope with situations of other components failing.

8.1.2.2 Threat Model

For this setting, we tolerate up to f_i arbitrary (i.e., Byzantine) faults in each replica group and choose n_i large enough for this fault threshold (e.g., $n_i = 2f_i + 1$ with quorums Q_i of size $|Q_i| = f_i + 1$). Adversaries compromise up to f_i replicas of a group no faster than T_i^a. The overall amount of faults is, therefore, $f = \sum_i f_i$ and $T^a = min\{T_i^a\}$. A rejuvenation period T_i^r of rejuvenating all n_i replicas of g_i faster than T_i^a prevents exhaustion failures.

8.1.3 Single Replicated Subsystem

The simplest case where consensual data structures are applied is the case of a single, replicated subsystem interacting with the data structure. Such a scenario is greatly equivalent to the capability space management example provided in *Midir*. There, we measured the performance of agreeing on and executing client-invoked system calls for granting and priming capabilities kept in a data structure. This measurement was performed in two different implementations of capability spaces: (i) as a private data structure in each replica, requiring, e.g., in the case of prime, a vote to install capabilities (add an element to a data structure) and two further to reply to the client and mark the system call as finished; and (ii) as a read-shared, consensually-updated data structure, trading off speed for a smaller memory footprint by introducing additional votes for track keeping. Figure 8.3 exemplifies the interaction between the replicas of the subsystem and the (i) local or (ii) the consensually-updated memory. In both cases a voting trusted-trustworthy mechanism is used to agree on the order of the updates to perform. In the figure, this mechanism is represented as black-box voting entity for simplicity, hiding the details of *iBFT*. However, for a local memory approach, upon reaching agreement within the data-structure-implementing group, each replica individually applies the operation on its own copy of the data structure, residing on each replica's local memory block. On the other hand, when using consensually-updated memory, the voting results are applied to a single block of memory, shared by all replicas directly by the trusted operation of the voting mechanism, preventing replicas from directly mod-

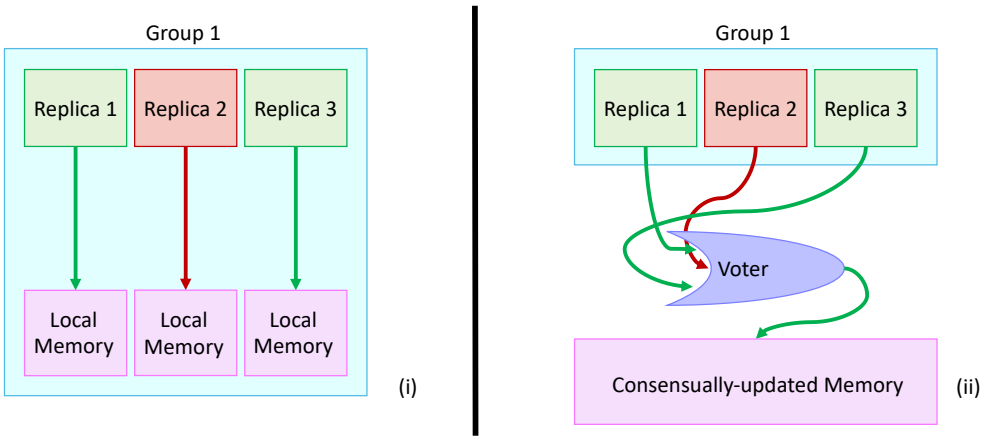


Figure 8.3: Subsystem interacting with (i) local or (ii) the consensually-updated memory.

ifying the memory region where the shared structure resides and avoiding the need to have a structure-owning group.

Even a single group interacting with a data structure requires attention to some details, depending on the approach chosen for the memory implementation. In such a scenario, issues only arise when using approach (ii):

- ***1)*** As exemplified in Section 8.1.1, agreeing on old values is crucial to safely performing destructive updates. This issue is, however, only relevant for approach (ii), as in (i) each replica will read from its own local memory, to which faulty replicas shall have no access, an assumption substantiated by the presented architectures. Thus, in (ii), replicas of the subsystem must vote not only on the update to perform, but also keep the values that will be destroyed, e.g., by storing them in consensually-updated group local storage before any writes can be made, i.e., applied by the voting mechanism;

- ***2)*** Group local storage, a per-group consensual memory, then stores old values needed temporarily for the safety of an update. In a sense, group local storage has the same purpose as the system call log in *Midir*, storing information that would otherwise not be know to enough replicas;

- ***3)*** Ensuring a deterministic order for subordinate votes ensures correct replicas apply the same updates to the same state. In *Midir*, we called subordinate votes the sequence of voted operations that resulted from agreeing on a system call, such

as performing critical operations and responding to the client. Our solution does not require lock-step execution and makes no assumptions on the order in which replicas update their local state. Nevertheless, to simplify tracing the progress of the update (and in turn the code that late or rebooted replicas have to execute to catch up), it is important that this requirement is met;

- ***4)*** Replicas read the structures according to the implementation limitations and taking into account read/write synchrony. In the case of local per-replica structures, the fault threshold accounts for one limitation, as a majority of copies must be read to ensure faulty replicas did not meddle with the data. We shall go into such details in Section 8.1.5;

The positive trade-off for using approach (ii) comes from the smaller memory footprint and the lack of overhead from reading a majority of copies.

8.1.4 Replica Groups

We now generalize our approach from a single subsystem to multiple interacting subsystems, reading and updating common data structures. In addition to the concurrency (and possibly malicious intent of faulty replicas) in a single group, a second dimension is added for concurrency and malicious intent, by allowing different groups and the malicious replicas therein to perform different operations concurrently on the same data structure. Specifically, faulty replicas of one group now find as potential victims the healthy replicas of other groups and as potential accomplices the faulty replicas of these groups. In other words, there are now two dimensions of concurrency: (i) replicas of the same group accessing a shared data structure and (ii) synchronization across replica groups. We therefore have to ensure in addition the following principles:

- ***P1***: Replicas s_i^j of a group G_i can only participate in consensual operations of that group;
- ***P2***: Replicas s_i^j cannot impersonate replicas of other groups;
- ***P3***: Faulty replicas in G_i cannot create inconsistencies in the data structure by performing concurrent operations (consensually, on their own or in collusion with other faulty replicas).

We achieve ***P1*** and ***P2*** by constraining replicas to per replica group-dedicated voting. That is, at any point in time, $v_k \in V$ is either exclusively accessible by group G_i or multiplexed among groups in a trustworthy manner. To address ***P3***, we introduce the notion of consensual locks, to coordinate which group obtains access to specific parts of the data structure.

```
1   reader:
2     pre := read seqn
3     read structure
4     post = read seqn
5     if pre != post
6         repeat
7
8   writer:
9     seqn++
10    write structure
11    seqn++
```

Figure 8.4: Sequence lock code

Before moving to the details regarding the different ways consensual data structures can be implemented, let us further illustrate the remaining minutiae on achieving safe reading and updates. We shall enumerate here these important technicalities, explaining and referring to them later in Section 8.1.5 as we describe the implementations.

8.1.4.1 Multiple Non-Replicated Readers

The simplest case of reading interaction with multiple subsystems pertains to multiple non-replicated readers interacting with a single data-structure-implementing group. This would be the case if readers consisted of non-replicated subsystems. Different subsystems may have distinct replication degrees, according, e.g., to their criticality. As such, a subsystem with no replicas is possible to co-exist with other subsystems.

Synchronization: For a subsystem to have a chance to read a consistent snapshot of the data structure, it must know whether a write operation is ongoing or whether such a write interleaved with the read, a knowledge sequence locks can secure. Figure 8.4 demonstrates their use. When a reader initiates a read, it saves the value of the current sequence lock, held by structure-implementing group. It then proceeds to read the data structure and, once the operation is terminated, compare the saved value with the sequence lock value after the read. If they differ, it means a write was performed during the read, deeming a repetition of the read necessary.

Writers, on the other hand, increase the sequence lock before writing and then again after writing. When the value is odd, it means a write is ongoing, when it is even it means the write concluded.

Faulty replica behaviour: Faulty replicas may not adhere to data structure invariants. This is an issue affecting only the local per-replica approach, as the shared consensual option has the trusted copy operation performing the write to a single memory region. In the former approach, each replica updates their local copy individually, meaning that faulty ones may introduce null pointers, pointers to arbitrary memory, pointers

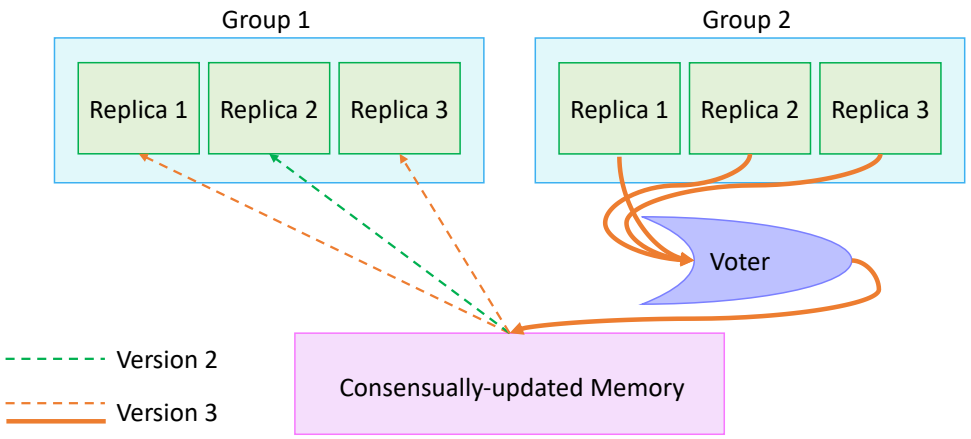


Figure 8.5: Replicated reader group reading different versions of the same correct data structure in the case where the values read are not agreed by at least $f + 1$.

to addresses that will result in a fault (if the memory region is not pre-mapped) and create wrong structures, e.g., cycles in supposedly acyclic lists and wrong $prev$ and $next$ pointers.

8.1.4.2 Multiple Replicated Readers

When the interaction involves multiple replicated readers, a new issue arises: the possibility of inconsistent reads when reading from the same shared data structure. It may so happen that replicas within a subsystem read different versions of the same correct structure. As discussed before, even within a manycore's NoC, synchronicity cannot always be guaranteed, as such, latency and interleaved writes may lead two correct replicas to read different values. Figure 8.5 exemplifies such a scenario. Replicas 1 and 3 read version 3 of the structure, while replica 2 reads version 2. Thus, consensus can only be reached if $f + 1$ replicas reach agreement on the same value and same version number.

8.1.5 Implementations

Several implementations are possible depending on how one wishes to share the data and synchronize operations. In this section we show some options and discuss their respective advantages and pitfalls.

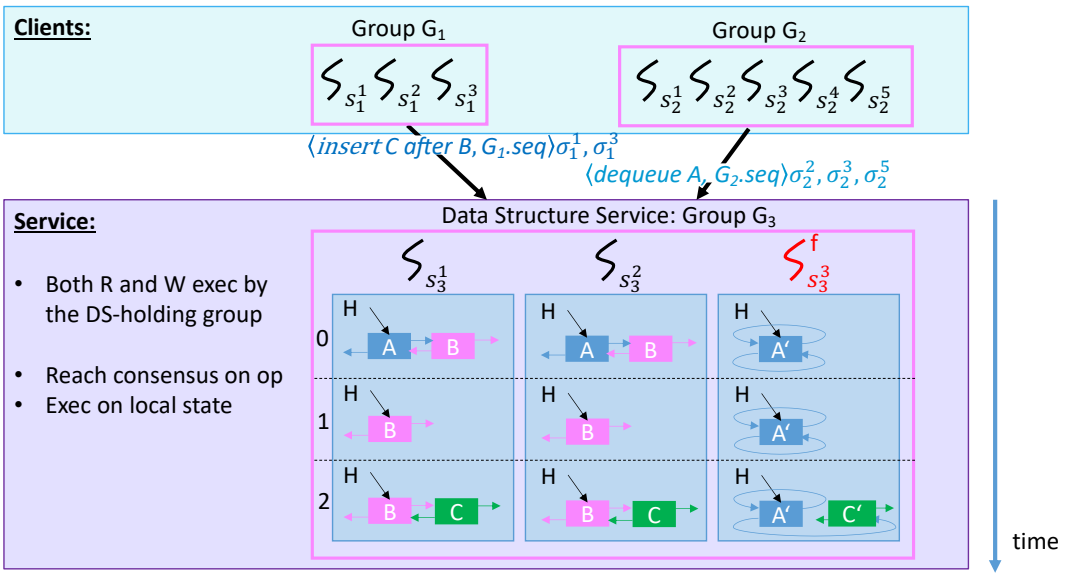


Figure 8.6: Data structure service.

8.1.5.1 Data Structure Service

Implementing the data structure as a service (client-server model) in essence means one of the subsystems will be the implementing data structure-managing group. Figure 8.6 exemplifies such an implementation and a demo interaction. In it, two groups G_1 and G_2 act as clients of the service, wishing to update the shared data structure that is held by group G_3. Each group has a different request, with G_1 asking to insert element C after B and G_2 asking to dequeue A. Here, a local per-replica memory model is provided and each replica is responsible for updating their own copy of the data. Informally, each group votes on the operation to perform on their own and, when consensus has been reached, the operation is proposed for execution in the implementing group, which, in turn, orders the requests received from all the groups and has each replica update their local data structure independently. Both read and write operations are executed by the data structure-holding group, meaning replicas from other groups cannot ever directly access the data.

Such an approach comes, of course, with its own pitfalls. Namely, locally updating state means faulty replicas are free to modify their state as they please, ignoring the requests, applying it more than once, updating the data in a completely different way than intended or deleting it. This incurs some overhead in operations such as reads, which have to be performed on a majority of replicas to ensure the results reflects a correct majority.

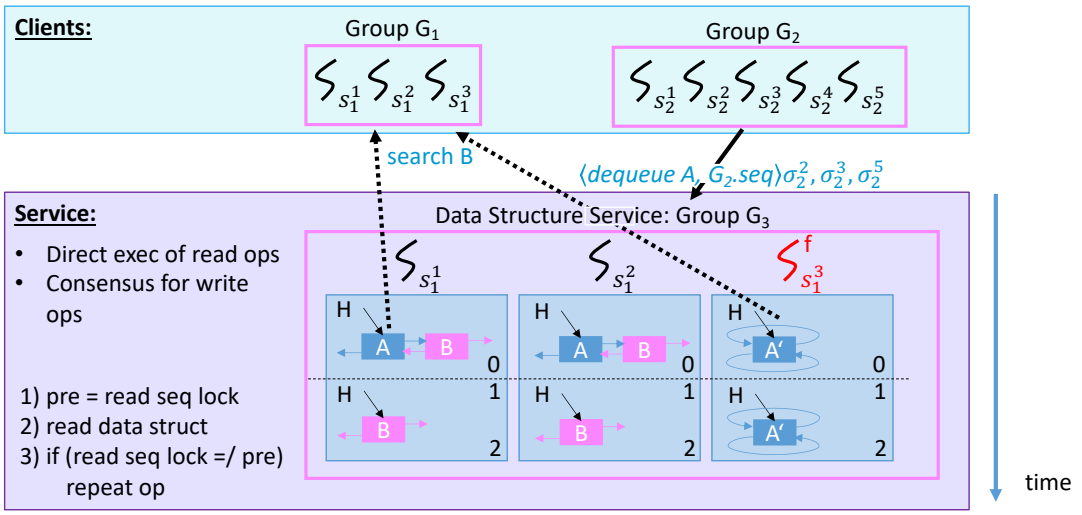


Figure 8.7: Read-shared data structure service.

8.1.5.2 Read-Shared Per-Replica Data Structures

Data structure services can implement read-shared data structures in a similar manner by granting subsystems direct read access to the memory regions of the service replicas. Write requests (such as dequeueing an element) are handled as described above, however, subsystems are expected to perform read requests on their own as depicted by the dashed lines in Figure 8.7.

As shown in the Figure and described in Section 8.1.4.1, several challenges must be addressed to compensate for the service no longer ordering read requests and to mask potentially faulty service replicas. For the former, we shall at first only discuss how replicas can detect when exactly they are reading (relative to the writes of other groups) to then return in Section 8.1.6.4 to more elaborate synchronization schemes. For a group (here G_1) to have a chance to read a consistent snapshot of the data structure, it must know whether a write operation is ongoing or whether such a write interleaved with the read. Sequence locks[1] meet this requirement. That is, healthy replicas mark the data

[1] Sequence locks (seqlock) are a reader–writer mechanism addressing the issue of writer starvation. A seqlock stores both a lock and a sequence number. Naturally, the lock enables synchronization between two writers, while the counter secures consistency for readers. When updating the shared data, the writer increments the sequence number, both after acquiring the lock and before releasing it. Readers read the sequence number before and after reading the shared data. If the sequence number is odd when read, then a writer took the lock while the data was being read and it may have changed. If the sequence numbers are different, a writer has changed the data while it was being read. Reader keep on polling until they manage to read the same even sequence number before and after reading the data.

structure as being written by incrementing a sequence number from an even to an odd value, advancing it again after the write completes. Readers check this sequence number before and after the read, waiting for the sequence number to become even if the initial check revealed an ongoing write (odd sequence number) and repeating the operation if sequence numbers do not match.

Notice that it may well happen that different replicas (e.g., s_1^1 and s_1^2) read the structure at different points in time (i.e., with different sequence numbers), as first mentioned in Section 8.1.4.2. It then depends on the operation implemented by this group whether such behavior can be tolerated. In Section 8.1.6.4, we will discuss consensual locks to allow replicas to read the data structure in the same state.

The second challenge — masking the behavior of faulty replicas during reads — requires further precautions. For one, faulty replicas are not guaranteed to follow the sequence number scheme depicted above. This means they may present a consistent view while modifying it. Moreover, no guarantee is provided whether data is structured in the expected way. For example, in 8.7, replica s_3^3 presents cycles to the reader in a supposedly acyclic list. Traversal (e.g., when searching for element B) can, therefore, not rely on the validity of pointers or on the termination condition one expects from a correct instance. Before dereferencing a pointer, reading replicas must therefore check its validity (i.e., whether they point into the service replica's memory region). Moreover, as stated, readers cannot advance through the replica copies individually, but must consider multiple copies at the same time to detect faults in structural mismatches.

8.1.5.3 Element-Granular Read-Shared Consensual Data Structures

Read-shared consensual data structures still require a dedicated replica group to perform all updates on the data structure. This limits concurrency, in particular if groups commonly operate on their own elements without changing the structure (e.g., if they update the data kept in the list elements they added without re-enqueueing them to a different position). Element-granular read-shared consensual data structures provide for this extra concurrency. Rather than holding a copy of the entire data structure, the replica group which inserts an element keeps a copy of this element in each of its replica's memory region. This way, updates to this element can be performed directly without having to first consult with another replica group. To enqueue this element into the structure, the help of other groups is required to update in their respective elements pointers that must refer to this new element.

Figure 8.8 illustrates this approach. To insert B, group G_2 allocates a copy of it in their replica's memory regions and then requests group G_1, owner of element A to update A's next pointer to refer to B.

Several implementation variants are imaginable. However, approaches capable of reusing the same pointer for all copies suggest themselves. Assuming the memory region of each replica is known, then allocating elements at the same offset in this region

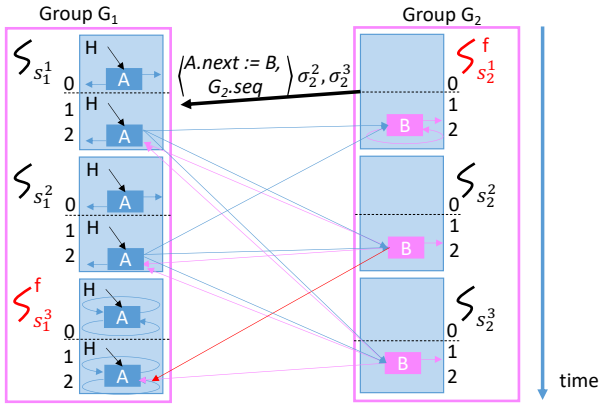


Figure 8.8: Element-granular read-shared data structure service.

allows a single pointer to be encoded as identifying the group and this offset.

8.1.5.4 Read/Write-Shared Consensual Data Structures

The previously presented approaches all use per-replica local memory. Instead this final solution allows for a single read/write-shared memory location for all replicas, which is consensually updated and used by all replica groups needing access to the data structure therein.

Naturally, there are two dimensions of concurrency: replicas of the same group accessing a shared data structure and synchronization across replica groups. All individual groups must reach consensus on the updates to perform as desired by the corresponding application. Thus, the challenge becomes coordinating each group's access to this shared state. When considering a single replica group, one does not need to worry about multiple entities modifying the same data structure simultaneously. The debate arises only from concurrent replica groups.

As explained previous sections, the agreed upon operations, if any, are copied (using the trusted copy operation) to memory, representing only one update, i.e., the update is not executed by each replica. However, this does not restrain each group from trying to update the same element of a data structure simultaneously. We need some way to prevent these concurrent writes to the same memory location, giving the opportunity to solely one group to write at a given time. Otherwise, consistency is compromised. In other words, this problem can be described as lock-free vs. lock-based data structures, which we shall explain in Section 8.1.6.

Figure 8.9 depicts the interaction of two replica groups with a read/write-shared data structure.

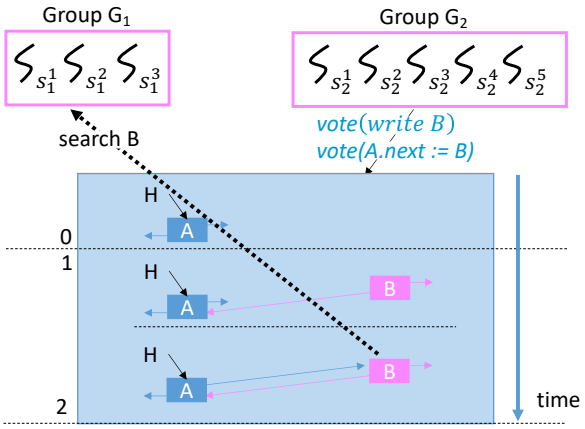


Figure 8.9: Read/write-shared data structure service.

8.1.6 Concurrent Access by Multiple Subsystems

In order to discuss concurrent access by multiple subsystems to the same shared data structure, let us first introduce access synchronization to then discuss it in the context of this Chapter.

8.1.6.1 Synchronization

Generally, when processes or threads must access a shared data structure, they do so either in a blocking, non-blocking or lock-free manner. In the first case, an unexpected delay by one process or thread can prevent others from making progress. In the worst case, a thread holding the lock may be put to sleep and thus block every other thread that is waiting on that lock, preventing them from making any progress. In essence, some operation is considered blocked if it is unable to progress in its execution until some other thread releases a resource. Non-blocking data structures [Her93; HLM03] are those on which all operations are non-blocking. For instance, all lock-free data structures are inherently non-blocking. Spin-locks are an example of non-blocking synchronization, meaning that, if one thread has a lock then waiting threads are not suspended, but must instead loop until the thread that holds the lock releases it. However, spin locks and other algorithms with busy-wait loops are not lock-free, since, if the thread holding the lock is suspended, then no thread can make progress. In order for operations to qualify as lock-free, they must allow a thread to complete its task regardless of the state of other threads.

A lock-free data structure is one that does not use any locks. The implication is

that multiple threads can access the data structure concurrently without race conditions or data corruption. Nevertheless, this does not mean that there are no access restrictions. A lock-free linked list might allow one thread to add values to the back while another removes them from the front, allowing this structure to be modified concurrently without corruption. In contrast, multiple threads adding new values concurrently would potentially make it corrupt. The data structure description must then identify which combinations of operations can safely be called concurrently. Therefore, if any thread performing an operation on the data structure is suspended at any point during that operation then the other threads accessing the data structure must still be able to complete their tasks.

8.1.6.2 Lock-Free vs. Consensual Locks

When two subsystems update the same shared data structure, then it must be either lock-free, meaning parallel updates must be considered, or one subsystem must prevent the other from interfering in the middle of a sequence of updates. Locks achieve the latter. For example, in the case of a doubly-linked list, two writes are required to delete a node (`prev->next = next; next->prev = prev`). Using locks would then be required to prevent another subsystem from inserting an element after the first, but before the second operation. Naturally, the modification of a data structure by one subsystem does not necessarily lock it in its entirety. As long as the updates of different subsystem target distinct elements, i.e., distinct memory pieces, parallel updates can be considered. This is, however, a design decision on where to place locks and at what granularity.

In our case, however, locks must be acquired and released by a group in a consensual manner as well, otherwise, a faulty replica could get hold of the lock preventing others from making progress by either never releasing it or constantly attempting to acquire it.

The question now becomes "how can we update distinct elements if they are correlated?". For instance, group A wishes to update element x with the value of y, while group B wants y to take the value of x. Even with a compare and swap operation, these two cannot be allowed to execute in parallel as it would be risky and possibly lead to inconsistencies.

8.1.6.3 Lock-Free: Azura

Compare and swap (CAS) is the most used primitive in lock-free algorithms [HLM03; DHM13], consisting on an atomic instruction for synchronization. CAS compares the contents of a memory location with an expected value and, if they match, modifies the contents of that memory location with the selected value.

A lock-free solution using CAS, for a specific location in memory, ensuring atomicity, comes in the form of a simple device, Azura. Azura gathers the updates originating

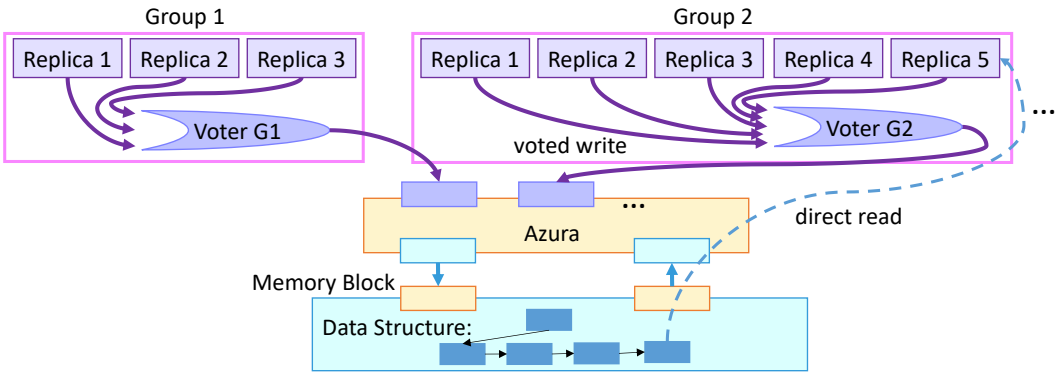


Figure 8.10: System overview, depicting the interaction of replica groups and the trusted-trustworthy components — voters and Azura — involved in implementing consensual memory. Replicas vote to update data structures in shared memory, while reading directly from this memory block. Azura arbitrates concurrent votes and ensures that votes are atomic.

```
1  if non-conflicting addresses then
2    if enough memory ports
3      forward all in parallel
4  else
5    check last forwarded request
6    set priority to last + 1
7    forward last + 1
```

Figure 8.11: Azura's voter to memory forward pattern.

from all voters and allows only one to go through at a time, unless they involve completely independent elements, in which case more than one update can be forwarded at a time if we assume a memory block with more than one port.

An overview of the interaction among replica groups, their voters, Azura and consensual memory is represented in Figure 8.10 and the pseudocode for Azura's logic can be found in Figure 8.11.

By taking into account the memory addresses accessed by each voters' operations, Azura is automatically locking other groups when necessary. Consider two cases:

Conflicting Updates: Imagine all voters, each representing a subsystem, tell Azura they want to update the same element. Assuming a memory controller does not apply some sort of coherence mechanism for conflicting updates to the same element performed simultaneously through different ports, all voters requests cannot be forwarded

to the memory in parallel. With this in mind, Azura applies a round robin-like arbitration scheme, nonetheless, without waiting for any update. Consider the following example. It is group g_0's turn to update memory, according to Azura's "scheduling". However, g_0 has no update to be performed. Instead of waiting for g_0 to propose an update, Azura simply checks if it received any update from g_1. If g_1 has an update, this update is forwarded to memory, otherwise, Azura checks for g_2. In case g_1 has sent nothing, g_2 is still next in line and g_0 is not checked again until Azura wraps around back to the beginning of the list. If g_0 for some reason flooded Azura with requests conflicting with other groups, it would be given the same chance to access the data structure as all the other groups.

Independent Updates: Imagine now groups are proposing independent updates. Azura checks in each cycle, which updates there are from each of its group ports. It compares the addresses requested by each and, if, independent forwards as much requests as there are ports to the memory block.

8.1.6.4 Consensual Locks

Alternatively, and to avoid the lock-free complexity, one could implement locks to protect the data structures from concurrent access in a more classical way, although by acquiring and releasing locks consensually to prevent corruption by faulty replicas. Also, for more complex algorithms requiring more fine grained mutual exclusivity, an approach in the form of consensual locks is demanded. Consensual lock (and unlock) operations come in the form of a spin lock, involving checking if a lock is free and collecting the sequence numbers for subordinate votes, i.e., acquiring and releasing the lock.

Consensual locks raise the same concerns that we have already seen in previous sections, namely if operations include destructive updates:

- By participating in the consensual acquisition, faulty replicas may prevent healthy replicas from learning that the lock is held;
- By participating in the consensual release, faulty replicas may create inconsistencies by colluding with replicas that have not learned about the release and possibly the re-acquisition of the lock; and
- Similar race conditions occur when quorum-size subsets of a subsystem consensually release a lock they hold, since late replicas in the complement of this quorum may continue to propose votes even after the lock has been released.

Unfortunately, the principles presented in Section 8.1.4 are not sufficient to prevent these inconsistencies, due to the inherent race conditions with other replica groups. The following illustrates these points.

```
1 vote("CAS (lock, free) => groupID, seq", seq)
2 vote("write_lock = free", seq + ops + 1)
```

Figure 8.12: Consensual lock voting.

Let us start by assuming an abstract lock, investigating which properties extend to consensual locks, before we provide concrete implementations for the latter. In classical lock implementations (e.g., `bit-test-and-set`), only the lock acquiring thread learns about the acquisition (by finding the bit clear). If we would generalize this behavior to consensual locks, only the replicas of the lock acquiring quorum learn about the acquisition. But if faulty replicas in this quorum refuse to participate in the subsequent vote to record in group-local storage the fact that the lock is held by the group, too few healthy replicas remain for forming a quorum that know about the acquisition.

A consensual lock operation must, thus, indicate both lock holder (allowing lagging replicas to know) and sequence number pertaining to when the lock was acquired, preventing any further operations after its release. Figure 8.12 presents the pseudo code.

8.2 Discussion

We have presented in this Chapter one of the many possible uses of *Midir* and *iBFT* on the context of interacting replicated applications sharing critical data structures. Despite *Midir*'s or *iBFT*'s applicability to any level of the software stack, specific considerations need to be taken when devising solutions for specific problems, such as consensual data structures. We have discussed ways in which subsystems may interact and how that interaction can be implemented. We have also analyzed how concurrent access can be handled when multiple subsystems must update the same data.

8.2.1 Azura Fault Model

Much like any other trusted component introduced thus far, Azura is simple enough to be trusted not to fail and easy to verify. It, however, still consists on a SPoF for crash faults. As such, similarly to the mechanism implementing the trusted copy, some form of replication can be applied, independent of the number of subsystems and their replication degree, both for Azura and the memory, depending on the chosen fault model.

Chapter 9

Conclusions and Future Work

9.1 Conclusion

This thesis presents solutions for the elimination of single points of failure in low-level management software, thus protecting the ever rising-in-popularity and -criticality MP-SoCs. We devised ways to provide fault and intrusion tolerance by reaping benefit of the tight coupling of replicas in such an environment and by applying concepts from classical distributed systems.

In particular, we constructed two distinct solutions with different approaches and benefits: *Midir* and *iBFT*. We showed how to circumvent a well-known impossibility for BFT-SMR protocols that cannot rely on transferable authentication, which is the case for tightly-coupled BFT-SMR protocols if they want to remain close to the performance of the replica-connecting communication medium: the on-chip networks of multi- and manycore systems and the shared memories they connect. We introduced trusted-trustworthy components to establish means for replicated low-level software to vote on critical operations, such as privilege reconfiguration and resource access.

D-MPSoCs achieve a quantum step towards off-the-shelf chip resilience, since the mechanisms presented are generic enough to support, in-chip and with high reliability, a large variety of the protection and redundancy management techniques normally implemented in software at higher layers in 'macro' systems. To convincingly prove our point, we exemplified and evaluated an implementation, over *Midir*, of the most complex version of our solution set: a Byzantine fault tolerant micro-hypervisor. We have shown the practicality of our concept, having quite satisfying performance. We observed that *iBFT* is able to outperform a shared memory implementation of the state-of-the-art hybrid BFT protocol MinBFT by one/two orders of magnitude (*wo* can crash/cannot crash); and *Midir* by one order of magnitude.

The low overhead of our approach shows as well large promise for future safe, full hardware solutions. Furthermore, our solutions were intentionally designed as a non-

intrusive extensions to current chip architectures, being anchored on simple and self-contained hardware extensions or microcode instructions (for the case of microcode-based *iBFT*). Taken up by a hardware manufacturer or integrator, they allow a backwards-compatible, non-fracturing evolution. We hope that our findings may be key to enhance general MPSoC architectures towards D-MPSoCs and, among other avenues, lead to next-generation COTS resilient chips.

9.2 Limitations and Future Work

Although we have achieved the goal we set for in Chapter 1, some challenges remain to be addressed in future work. Also, several questions remain to be answered, namely on kernel design details, diversification for sustainability, application-level uses, real-time applicability, coverage for network attacks, dynamic reconfiguration of deployed parameters and so forth.

Formal verification: In order to achieve maximum certainty of the zero fault probability of the trusted-trustworthy components presented (T2-H2, write-once memory and Azura), their formal verification should be carried out.

Scalability: We have not yet tested the work present here for an n in the order of the hundreds. As the available cores in manycore architectures continue to grow and given the increasing systems' complexity and security threats, higher replication degrees should be evaluated.

Memory Safety: Our work for now assumes the memory bus and memory subsystem to work correctly, hardware wise. As we have mentioned in regards to the trusted copy and Azura, some form of redundancy for the memory blocks and memory controller could be applied to ensure continued operation in the case of a crash. Further research on this topic should be carried out.

Bottlenecks: The L2 cache represents a bottleneck for agreeing on requests when introspecting other replicas in core to core communication. Even though this has not proven troublesome when evaluating *iBFT*, it can potentially hinder performance for higher values of n.

Safe Micro-Kernel Construction: Several aspects of micro-kernel construction remain untapped that *Midir* or *iBFT* could further simplify. In Section 4.6 we provided an example of granting and priming capabilities, however, more use cases can be explored.

Diversification: Although we consider diversification of chip components and n-version programming to be part of *Midir* and *iBFT*'s implementation, further research can be done on the subject, namely taking into account the state of the art. This pairs well with investigating network attacks, since at times the usage of components from different manufacturers (namely in regards to the NoC, which is not covered by fault isolation in our model) may not be a positive decision if the IP can potentially be corrupt. For instance, a compromised NoC IP would be able to manipulate data, degrade

performance or steal sensitive information [CM21].

Application-Level Uses: The motivation for our work was the SPoF syndrome in low-level management software, however, application-level software could as well benefit from the technologies we presented. *Midir* and *iBFT* could be used and/or adapted for application level use, providing for different scenarios of usability and criticality. As previously mentioned, our solutions can be applied at any level of the software stack.

Real-Time: We can adjust the developed mechanisms and protocols for use in safety-critical real-time systems and develop real-time kernel-level recovery mechanisms. Although we have high confidence both *Midir* and *iBFT* fit in real-time systems, as they require a finite number of operations and, in the worst case scenario, rotate leaders $f + 1$ times before completing agreement on an operation; further checking should be carried out.

Network Attacks: In our system model we assume the abstraction of a correct network. We did not, however, study the impact network attacks can have on the NoC and, in fact, did not perform any form of fault injection in the system as this was off the scope of the work performed in this thesis. Nonetheless, it is, of course, an interesting and relevant line of research. NoC security is, in fact, quite crucial since (i) it transports all system data and (ii) spans across the entire MPSoC. As pointed out in [CM21], there are three major forms of NoC vulnerabilities: malicious implants, backdoor/side-channels using test/debug interfaces, and unintentional vulnerabilities [FHM20]. Although countermeasures to these types of NoC attacks are already being developed, e.g., [SAJB14; RSBS16; CLM19; IHRS19], evaluating how they can be implemented on a *Midir*- or *iBFT*-aware platform is one of the predominant directions for future research.

Dynamic Reconfiguration: Hardware designs are, for now, not dynamically reconfigurable at run-time. Either it is in ASIC[1] (application-specific integrated circuit) form and, thus, an immutable integrated circuit; or programmed in an FPGA, in which case the "reconfigurable logic" term only applies to programming the hardware at design time, but not after the system is deployed. For instance, once the number of replicas is defined in the hardware, that circuitry serves no other purpose for the lifetime of the system. If we define n to be 5 and, thus, create 5 voter buffers, these will forever be there usable only for the purpose of voting. Of course we can later defined n in software to be 3 and leave 2 buffers unused, but we cannot repurpose that logic for something different. The idea is, then, that n, as well as other parameters, become reconfigurable while the system is running. This is possible to some extent by dynamically reprogramming the FPGA, however, this requires system downtime.

[1] An integrated circuit customized for particular use, commonly yielding better performance than reconfigurable hardware such as FPGAs

Bibliography

[ABWER21] Abdullah Al-Boghdady, Khaled Wassif, and Mohammad El-Ramly. "The Presence, Trends, and Causes of Security Vulnerabilities in Operating Systems of IoT's Low-End Devices". In: *Sensors* 21.7 (2021), p. 2329.

[AC+77] Algirdas Avizienis, Liming Chen, et al. "On the implementation of N-version programming for software fault-tolerance during program execution". In: (1977).

[ACG15] Will Arthur, David Challener, and Kenneth Goldman. *A practical guide to TPM 2.0: Using the new trusted platform module in the new age of security*. Springer Nature, 2015.

[ACKM06] Ittai Abraham, Gregory Chockler, Idit Keidar, and Dahlia Malkhi. "Byzantine disk paxos: optimal resilience with byzantine shared memory". In: *Distributed Computing* 18.5 (2006), pp. 387–408.

[ACR14] Dean Michael Ancajas, Koushik Chakraborty, and Sanghamitra Roy. "Fort-nocs: Mitigating the threat of a compromised noc". In: *Proceedings of the 51st Annual Design Automation Conference*. 2014, pp. 1–6.

[Agu+19] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra Marathe, and Igor Zablotchi. "The Impact of RDMA on Agreement". In: *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. PODC '19. Toronto ON, Canada: Association for Computing Machinery, 2019, pp. 409–418. DOI: 10.1145/3293611.3331601. URL: https://doi.org/10.1145/3293611.3331601.

[Agu+20] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J. Marathe, Athanasios Xygkis, and Igor Zablotchi. "Microsecond Consensus for Microsecond Applications". In: *14th USENIX Symposium on Operating Systems Design and Implementation*. Nov. 2020.

[Alo+05] Noga Alon, Michael Merritt, Omer Reingold, Gadi Taubenfeld, and Rebecca N Wright. “Tight bounds for shared memory systems accessed by Byzantine processes”. In: *Distributed Computing* 18.2 (2005), pp. 99–109.

[ALRL04] Algirdas Avizienis, J-C Laprie, Brian Randell, and Carl Landwehr. “Basic concepts and taxonomy of dependable and secure computing”. In: *IEEE transactions on dependable and secure computing* 1.1 (2004), pp. 11–33.

[AMT93] Yehuda Afek, Michael Merritt, and Gadi Taubenfeld. “Benign failure models for shared memory”. In: *International Workshop on Distributed Algorithms*. Springer. 1993, pp. 69–83.

[ARJS07] Nidhi Aggarwal, Parthasarathy Ranganathan, Norman P. Jouppi, and James E. Smith. “Configurable isolation: building high availability systems with commodity multi-core processors”. In: *International Symposium on Computer Architecture (ISCA)*. 2007, pp. 470–481.

[Asm+16] Nils Asmussen, Marcus Völp, Benedikt Nöthen, Hermann Härtig, and Gerhard Fettweis. “M3: A Hardware/Operating-System Co-Design to Tame Heterogeneous Manycores”. In: *Architectural Support for Programming Languages and Operating Systems*. ACM. Atlanta, GA, USA, Apr. 2016.

[Att02] Paul Attie. “Wait-free Byzantine consensus”. In: *Information Processing Letters* 83.4 (2002), pp. 221–227.

[Bau+09] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. “The Multikernel: A New OS Architecture for Scalable Multicore Systems”. In: *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*. SOSP ’09. Big Sky, Montana, USA: ACM, 2009, pp. 29–44. DOI: 10.1145/1629575.1629579. URL: http://doi.acm.org/10.1145/1629575.1629579.

[BCM13] Cristiana Bolchini, Matteo Carminati, and Antonio Miele. “Self-Adaptive Fault Tolerance in Multi-/Many-Core Systems”. In: *J. Electron. Test.* 29.2 (Apr. 2013), pp. 159–175. DOI: 10.1007/s10836-013-5367-y. URL: http://dx.doi.org/10.1007/s10836-013-5367-y.

[BCP12] Tyson T Brooks, Carlos Caicedo, and Joon S Park. “Security vulnerability analysis in virtualized computing environments”. In: *International Journal of Intelligent Computing Research* 3.1/2 (2012), pp. 277–291.

[BCSFL09] Alysson Neves Bessani, Miguel Correia, Joni da Silva Fraga, and Lau Cheuk Lung. “Sharing memory between Byzantine processes using policy-enforced tuple spaces”. In: *IEEE Transactions on Parallel and Distributed Systems* 20.3 (2009), pp. 419–432.

[BDK16] Travis Boraten, Dominic DiTomaso, and Avinash Karanth Kodi. “Secure model checkers for Network-on-Chip (NoC) architectures”. In: *2016 International Great Lakes Symposium on VLSI (GLSVLSI)*. IEEE. 2016, pp. 45–50.

[BDM02] Luca Benini and Giovanni De Micheli. “Networks on chips: A new SoC paradigm”. In: *computer* 35.1 (2002), pp. 70–78.

[BDM09] Geoffrey Blake, Ronald G Dreslinski, and Trevor Mudge. “A survey of multicore processors”. In: *IEEE Signal Processing Magazine* 26.6 (2009), pp. 26–37.

[Bha+16] Koustubha Bhat, Dirk Vogt, Erik van der Kouwe, Ben Gras, Lionel Sambuc, Andrew S. Tanenbaum, Herbert Bos, and Cristiano Giuffrida. “OSIRIS: Efficient and Consistent Recovery of Compartmentalized Operating Systems”. In: *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. June 2016, pp. 25–36. DOI: 10.1109/DSN.2016.12.

[BLH18a] Simon Biggs, Damon Lee, and Gernot Heiser. “The Jury Is In: Monolithic OS Design Is Flawed”. In: *Asia-Pacific Workshop on Systems (APSys)*. Korea: ACM SIGOPS, Aug. 2018. DOI: 10.1145/3265723.3265733.

[BLH18b] Simon Biggs, Damon Lee, and Gernot Heiser. “The Jury Is In: Monolithic OS Design Is Flawed”. In: *Asia-Pacific Workshop on Systems (APSys)*. Korea: ACM SIGOPS, Aug. 2018. DOI: 10.1145/3265723.3265733.

[BS95] Thomas C. Bressoud and Fred B. Schneider. “Hypervisor-based fault tolerance”. In: *15th ACM Symposium on Operating Systems Principles (SOSP)*. Copper Mountain, Colorado, USA, 1995, pp. 1–11.

[Cas+14] António Casimiro, José Rufino, Ricardo C Pinto, Eric Vial, Elad M Schiller, Oscar Morales-Ponce, and Thomas Petig. “A kernel-based architecture for safe cooperative vehicular functions”. In: *Proceedings of the 9th IEEE International Symposium on Industrial Embedded Systems (SIES 2014)*. IEEE. 2014, pp. 228–237.

[CAS13] Xin Chen, Erika Ábrahám, and Sriram Sankaranarayanan. “Flow*: An analyzer for non-linear hybrid systems”. In: *International Conference on Computer Aided Verification*. Springer. 2013, pp. 258–263.

[CD16] Victor Costan and Srinivas Devadas. *Intel SGX Explained*. Tech. rep. https://eprint.iacr.org/2016/086.pdf (Accessed: 2016-07-22). Massachusetts Institute of Technology, 2016.

[CDA14] John Criswell, Nathan Dautenhahn, and Vikram Adve. "KCoFI: Complete control-flow integrity for commodity operating system kernels". In: *2014 IEEE Symposium on Security and Privacy*. IEEE. 2014, pp. 292–307.

[CGR17] António Casimiro, Inês Gouveia, and José Rufino. "Enforcing timeliness and safety in mission-critical systems". In: *Ada-Europe International Conference on Reliable Software Technologies*. Springer. 2017, pp. 53–69.

[Cha+95] John Chapin, Mendel Rosenblum, Scott Devine, Tirthankar Lahiri, Dan Teodosiu, and Anoop Gupta. "Hive: Fault Containment for Shared-memory Multiprocessors". In: *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*. SOSP '95. Copper Mountain, Colorado, USA: ACM, 1995, pp. 12–25. DOI: 10.1145/224056.224059. URL: http://doi.acm.org/10.1145/224056.224059.

[CJKR12] Allen Clement, Flavio Junqueira, Aniket Kate, and Rodrigo Rodrigues. "On the (limited) power of non-equivocation". In: *Proceedings of the 2012 ACM symposium on Principles of distributed computing*. 2012, pp. 301–308.

[CL99] Miguel Castro and Barbara Liskov. "Practical Byzantine Fault Tolerance". In: *3rd Symposium on Operating Systems Design and Implementation*. ACM. New Orleans, USA, Feb. 1999.

[CLM19] Subodha Charles, Yangdi Lyu, and Prabhat Mishra. "Real-time detection and localization of DoS attacks in NoC based SoCs". In: *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2019, pp. 1160–1165.

[CM20a] Subodha Charles and Prabhat Mishra. "Lightweight and trust-aware routing in NoC-based SoCs". In: *2020 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE. 2020, pp. 160–167.

[CM20b] Subodha Charles and Prabhat Mishra. "Securing network-on-chip using incremental cryptography". In: *2020 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE. 2020, pp. 168–175.

[CM21] Subodha Charles and Prabhat Mishra. "A Survey of Network-on-Chip Security Attacks and Countermeasures". In: *ACM Computing Surveys (CSUR)* 54.5 (2021), pp. 1–36.

[CMDTM] Carlos Colman-Meixner, Chris Develder, Massimo Tornatore, and Biswanath Mukherjee. “A Survey on Resiliency Techniques in Cloud Computing Infrastructures and Applications”. In: *IEEE Communications Surveys Tutorials* 18.3 (), pp. 2244–2281. DOI: 10.1109/COMST.2016.2531104.

[CMS08] Byung-Gon Chun, Petros Maniatis, and Scott Shenker. “Diverse Replication for Single-machine Byzantine-fault Tolerance”. In: *USENIX 2008 Annual Technical Conference*. ATC’08. Boston, Massachusetts: USENIX Association, 2008, pp. 287–292. URL: http://dl.acm.org/citation.cfm?id=1404014.1404038.

[CNV04] Miguel Correia, Nuno Ferreira Neves, and Paulo Verissimo. “How to tolerate half less one Byzantine nodes in practical distributed systems”. In: *Reliable Distributed Systems, 2004. Proceedings of the 23rd IEEE International Symposium on*. IEEE. 2004, pp. 174–183.

[CNV12] Miguel Correia, Nuno Ferreira Neves, and Paulo Verissimo. “BFT-TO: Intrusion tolerance with less replicas”. In: *The Computer Journal* 56.6 (2012), pp. 693–715.

[CSK07] B. Chun, P. Maniatis andS. Shenker, and J. Kubiatowicz. “Attested append-only memory: Making adversaries stick to their word”. In: *21st ACM Symposium on Operating Systems Principles (SOSP)*. Stevenson, Washington, USA, Oct. 2007, pp. 189–204.

[CVP99] Tzi-cker Chiueh, Ganesh Venkitachalam, and Prashant Pradhan. “Integrating segmentation and paging protection for safe, efficient and transparent software extensions”. In: *Proceedings of the seventeenth ACM symposium on Operating systems principles*. 1999, pp. 140–153.

[Das19] Debak Das. *An Indian nuclear power plant suffered a cyberattack. Here’s what you need to know.* https://www.washingtonpost.com/politics/2019/11/04/an-indian-nuclear-power-plant-suffered-cyberattack-heres-what-you-need-know/. Accessed: 2017-03-12. 2019.

[Dav16] Alex Davies. *Tesla’s Autopilot Has Had Its First Deadly Crash*. https://www.wired.com/2016/06/teslas-autopilot-first-deadly-crash/. Accessed: 2017-03-12. 2016.

[DCCC08] Francis M. David, Ellick M. Chan, Jeffrey C. Carlyle, and Roy H. Campbell. “CuriOS: Improving Reliability Through Operating System Structure”. In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. OSDI’08. San Diego, California:

USENIX Association, 2008, pp. 59–72. URL: http://dl.acm.org/citation.cfm?id=1855741.1855746.

[DCK15] Tobias Distler, Christian Cachin, and Rüdiger Kapitza. "Resource-efficient Byzantine fault tolerance". In: *IEEE transactions on computers* 65.9 (2015), pp. 2807–2819.

[DGY14] Tudor David, Rachid Guerraoui, and Maysam Yabandeh. "Consensus Inside". In: *Proceedings of the 15th International Middleware Conference*. Middleware '14. Bordeaux, France: ACM, 2014, pp. 145–156. DOI: 10.1145/2663165.2663321. URL: http://doi.acm.org/10.1145/2663165.2663321.

[DHM13] David Dice, Danny Hendler, and Ilya Mirsky. "Lightweight contention management for efficient compare-and-swap operations". In: *European Conference on Parallel Processing*. Springer. 2013, pp. 595–606.

[Dis+11] Tobias Distler, Ivan Popov, Wolfgang Schröder-Preikschat, Hans P Reiser, and Rüdiger Kapitza. "SPARE: Replicas on Hold." In: *NDSS*. 2011.

[DLS88] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. "Consensus in the presence of partial synchrony". In: *Journal of the ACM (JACM)* 35.2 (1988), pp. 288–323.

[DS10] Alex Depoutovitch and Michael Stumm. "Otherworld: Giving Applications a Chance to Survive OS Kernel Crashes". In: *Proceedings of the 5th European Conference on Computer Systems*. EuroSys '10. Paris, France: ACM, 2010, pp. 181–194. DOI: 10.1145/1755913.1755933. URL: http://doi.acm.org/10.1145/1755913.1755933.

[DT01] William J Dally and Brian Towles. "Route packets, not wires: on-chip inteconnection networks". In: *Proceedings of the 38th annual design automation conference*. 2001, pp. 684–689.

[Dö14] Björn Döbel. "Operating System Support for Redundant Multithreading". PhD thesis. Dresden, Germany: Technische Universität Dresden, Nov. 2014.

[ECP18] Emanuele Giuseppe Esposito, Paulo Coelho, and Fernando Pedone. "Kernel Paxos". In: *37th Symposium on Reliable Distributed Systems (SRDS)*. IEEE. 2018.

[ED12] Michael Engel and Björn Döbel. "The reliable computing base: A paradigm for software-based reliability". In: *Workshop on SOBRES*. 2012.

[EG17] Mark Ermolov and Maxim Goryachy. “How to Hack a Turned-Off Computer —- or Running Unsigned Code in Intel Management Engine”. In: *Black hat Europe*. avail at https://www.blackhat.com/docs/eu-17/materials/eu-17-Goryachy-How-To-Hack-A-Turned-Off-Computer-Or-Running-Unsigned-Code-In-Intel-Management-Engine.pdf, accessed 15.04.2018. London, UK, Sept. 2017.

[ES13] Kevin Elphinstone and Yanyan Shen. “Increasing the trustworthiness of commodity hardware through software”. In: *43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2013.

[Fen+18] Yixiong Feng, Bingtao Hu, He Hao, Yicong Gao, Zhiwu Li, and Jianrong Tan. “Design of distributed cyber–physical systems for connected and automated vehicles with implementing methodologies”. In: *IEEE Transactions on Industrial Informatics* 14.9 (2018), pp. 4200–4211.

[Fer+17] Andrew Ferraiuolo, Yao Wang, Rui Xu, Danfeng Zhang, Andrew Myers, and Edward Suh. “Full-processor timing channel protection with applications to secure hardware compartments”. In: (2017).

[Feu71] Edward A Feustel. “The Rice research computer: a tagged architecture”. In: *Proceedings of the May 16-18, 1972, spring joint computer conference*. 1971, pp. 369–377.

[FHM20] Farimah Farahmandi, Yuanwen Huang, and Prabhat Mishra. *System-on-Chip Security*. Springer, 2020.

[FLHZ13] Dabin Fang, Huikai Li, Jun Han, and Xiaoyang Zeng. “Robustness analysis of mesh-based network-on-chip architecture under flooding-based denial of service attacks”. In: *2013 IEEE Eighth International Conference on Networking, Architecture and Storage*. IEEE. 2013, pp. 178–186.

[FLP85] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. “Impossibility of distributed consensus with one faulty process”. In: *Journal of the ACM (JACM)* 32.2 (1985), pp. 374–382.

[FPS08] Leandro Fiorin, Gianluca Palermo, and Cristina Silvano. “A security monitoring service for NoCs”. In: *Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis*. 2008, pp. 197–202.

[Fre+11] Goran Frehse, Colas Le Guernic, Alexandre Donzé, Scott Cotton, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang, and Oded Maler. "SpaceEx: Scalable verification of hybrid systems". In: *International Conference on Computer Aided Verification*. Springer. 2011, pp. 379–395.

[FS12] Matthias Függer and Ulrich Schmid. "Reconciling fault-tolerant distributed computing and systems-on-chip". In: *Distributed Computing* 24.6 (2012), pp. 323–355.

[Fur+12] Steve B Furber, David R Lester, Luis A Plana, Jim D Garside, Eustace Painkras, Steve Temple, and Andrew D Brown. "Overview of the spinnaker system architecture". In: *IEEE Transactions on Computers* 62.12 (2012), pp. 2454–2467.

[FZWK17] Pedro Fonseca, Kaiyuan Zhang, Xi Wang, and Arvind Krishnamurthy. "An empirical study on the correctness of formally verified distributed systems". In: *Proceedings of the Twelfth European Conference on Computer Systems*. 2017, pp. 328–343.

[Gan+21] Neeraj Gandhi, Edo Roth, Brian Sandler, Andreas Haeberlen, and Linh Thi Xuan Phan. "REBOUND: defending distributed systems against attacks with bounded-time recovery". In: *Proceedings of the Sixteenth European Conference on Computer Systems*. 2021, pp. 523–539.

[Gar+11] Miguel Garcia, Alysson Bessani, Ilir Gashi, Nuno Neves, and Rafael Obelheiro. "OS diversity for intrusion tolerance: Myth or reality?" In: *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*. IEEE. 2011, pp. 383–394.

[Gar+14] Miguel Garcia, Alysson Bessani, Ilir Gashi, Nuno Neves, and Rafael Obelheiro. "Analysis of operating system diversity for intrusion tolerance". In: *Software: Practice and Experience* 44.6 (2014), pp. 735–770.

[GBN19] Miguel Garcia, Alysson Bessani, and Nuno Neves. "Lazarus: Automatic Management of Diversity in BFT Systems". In: *Proceedings of the 20th International Middleware Conference*. ACM. 2019, pp. 241–254.

[Gen18] David Gens. "OS-Level Attacks and Defenses: From Software to Hardware-Based Exploits". PhD thesis. Technische Universität Darmstadt, Dec. 2018.

[Gol84] Jack Goldberg. *Development and analysis of the software implemented fault-tolerance (SIFT) computer*. SRI International, 1984.

[Gre15] Andy Greenberg. *Hackers Remotely Kill a Jeep on the Highway*. http://www.wired.com/remotely-kill-jeep-highway/. 2015.

[GTHR99] Kinshuk Govil, Dan Teodosiu, Yongqiang Huang, and Mendel Rosenblum. “Cellular Disco: Resource Management Using Virtual Clusters on Shared-memory Multiprocessors”. In: *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*. SOSP ’99. Charleston, South Carolina, USA: ACM, 1999, pp. 154–169. DOI: 10.1145/319151.319162. URL: http://doi.acm.org/10.1145/319151.319162.

[Guo+19] Shize Guo, Jian Wang, Zhe Chen, Zhonghai Lu, Jinhong Guo, and Lian Yang. “Security-aware task mapping reducing thermal side channel leakage in CMPs”. In: *IEEE Transactions on Industrial Informatics* 15.10 (2019), pp. 5435–5443.

[Har85] Norman Hardy. “KeyKOS Architecture”. In: *SIGOPS Oper. Syst. Rev.* 19.4 (Oct. 1985), pp. 8–25. DOI: 10.1145/858336.858337.

[HDL13] Martin Hoffmann, Christian Dietrich, and Daniel Lohmann. “Failure by Design: Influence of the RTOS Interface on Memory Fault Resilience”. In: *Proceedings of the 2nd GI Workshop on Software-Based Methods for Robust Embedded Systems (SOBRES ’13)*. Ed. by German Society of Informatics. Koblenz, Germany, 2013. URL: http://www4.cs.fau.de/Publications/2013/hoffmann_13_sobres.pdf.

[Her+06] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. “Construction of a Highly Dependable Operating System”. In: *Proceedings of the Sixth European Dependable Computing Conference*. EDCC ’06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 3–12. DOI: 10.1109/EDCC.2006.7. URL: https://doi.org/10.1109/EDCC.2006.7.

[Her93] Maurice Herlihy. “A methodology for implementing highly concurrent data objects”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 15.5 (1993), pp. 745–770.

[HHWT97] Thomas A Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. “HyTech: A model checker for hybrid systems”. In: *International Journal on Software Tools for Technology Transfer* 1.1-2 (1997), pp. 110–122.

[HLM03] Maurice Herlihy, Victor Luchangco, and Mark Moir. “Obstruction-free synchronization: Double-ended queues as an example”. In: *23rd International Conference on Distributed Computing Systems, 2003. Proceedings.* IEEE. 2003, pp. 522–529.

[HMGP18] Mubashir Hussain, Amin Malekpour, Hui Guo, and Sri Parameswaran. “EETD: An energy efficient design for runtime hardware trojan detection in untrusted network-on-chip”. In: *2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE. 2018, pp. 345–350.

[Hof+13] Owen S. Hofmann, Sangman Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel. “InkTag: Secure Applications on an Untrusted Operating System”. In: *SIGPLAN Not.* 48.4 (Mar. 2013), pp. 265–278. DOI: 10.1145/2499368.2451146. URL: http://doi.acm.org/10.1145/2499368.2451146.

[IHRS19] Leandro Soares Indrusiak, James Harbin, Cezar Reinbrecht, and Johanna Sepúlveda. “Side-channel protected MPSoC through secure real-time networks-on-chip”. In: *Microprocessors and Microsystems* 68 (2019), pp. 34–46.

[Iqb+16] Salman Iqbal, Miss Laiha Mat Kiah, Babak Dhaghighi, Muzammil Hussain, Suleman Khan, Muhammad Khurram Khan, and Kim-Kwang Raymond Choo. “On cloud security attacks: A taxonomy and intrusion detection and prevention as a service”. In: *Journal of Network and Computer Applications* 74 (2016), pp. 98–120.

[JA88] Mark K. Joseph and Algirdas Avizienis. “A fault tolerance approach to computer viruses”. In: *IEEE Symposium on Security and Privacy*. Oakland, CA, USA. 1988, pp. 52–58.

[JACR15] Rajesh JS, Dean Michael Ancajas, Koushik Chakraborty, and Sanghamitra Roy. “Runtime detection of a bandwidth denial attack from a rogue network-on-chip”. In: *Proceedings of the 9th International Symposium on Networks-on-Chip*. 2015, pp. 1–8.

[Jvn] https://jvndb.jvn.jp/en/.

[Kap+12] Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. “CheapBFT: Resource-efficient Byzantine Fault Tolerance”. In: *Proceedings of the 7th ACM European Conference on Computer Systems*. EuroSys ’12. Bern, Switzerland: ACM, 2012, pp. 295–308. DOI: 10.1145/2168836.2168866. URL: http://doi.acm.org/10.1145/2168836.2168866.

[KB03] Hermann Kopetz and Günther Bauer. “The time-triggered architecture”. In: *Proceedings of the IEEE* 91.1 (2003), pp. 112–126.

[KJJR11] Paul Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. “Introduction to differential power analysis”. In: *Journal of Cryptographic Engineering* 1.1 (2011), pp. 5–27.

[KL86] John C. Knight and Nancy G. Leveson. “An experimental evaluation of the assumption of independence in multiversion programming”. In: *IEEE Transactions on software engineering* 1 (1986), pp. 96–109.

[Kle+09a] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. “seL4: Formal Verification of an OS Kernel”. In: ed. by Jeanna Neefe Matthews and Thomas E. Anderson. ACM, 2009, pp. 207–220. DOI: 10.1145/1629575.1629596. URL: http://doi.acm.org/10.1145/1629575.1629596.

[Kle+09b] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. “seL4: Formal Verification of an OS Kernel”. In: ed. by Jeanna Neefe Matthews and Thomas E. Anderson. ACM, 2009, pp. 207–220. DOI: 10.1145/1629575.1629596. URL: http://doi.acm.org/10.1145/1629575.1629596.

[Koc+18] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haar, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. *Spectre Attacks: Exploiting Speculative Execution*. Tech. rep. (see also: CVE-2017-5715, -5753, CVE-2018-3693, -3640, -3639, -3665, -3615, -3620, -3646, -9056). ArXiv e-prints 1801.01203, Jan. 2018.

[KRAT13] Hemangee K Kapoor, G Bhoopal Rao, Sharique Arshi, and Gaurav Trivedi. “A security framework for noc using authenticated encryption and session keys”. In: *Circuits, Systems, and Signal Processing* 32.6 (2013), pp. 2605–2622.

[KS19] Sandeep K. Shukla. *Editorial: Reflections on the History of Cyber-Physical versus Embedded Systems*. ACM Digital Library: https://dl.acm.org/doi/fullHtml/10.1145/3325115. 2019.

[KSRL10] Eric Keller, Jakub Szefer, Jennifer Rexford, and Ruby B. Lee. “Nohype: Virtualized cloud infrastructure without the virtualization”. In: *37th International Symposium on Computer Architecture (ISCA’10)*. Saint-Malo, 2010.

[Kuv+16] Dmitrii Kuvaiskii, Rasha Faqueh, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. “HAFT: Hardware-assisted Fault Tolerance”. In: *11th European Conference on Computer Systems (EuroSys)*. London, UK, Apr. 2016, pp. 1–17.

[LAC16] Robert M. Lee, Michael J. Assante, and Tim Conway. *Analysis of the Cyber Attack on the Ukrainian Power Grid*. E-ISAC: https://ics.sans.org/media/E-ISAC_SANS_Ukraine_DUC_5.pdf. Mar. 2016.

[LAK09] Andrew Lenharth, Vikram S. Adve, and Samuel T. King. "Recovery Domains: An Organizing Principle for Recoverable Operating Systems". In: *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XIV. Washington, DC, USA: ACM, 2009, pp. 49–60. DOI: 10.1145/1508244.1508251. URL: http://doi.acm.org/10.1145/1508244.1508251.

[Lam74] Butler W Lampson. "Protection". In: *ACM SIGOPS Operating Systems Review* 8.1 (1974), pp. 18–24.

[Lam98] Leslie Lamport. "The part-time parliament". In: *Transactions on Computer Systems* 16.2 (1998), pp. 133–169.

[LC10] Slobodan Lukovic and Nikolaos Christianos. "Hierarchical multi-agent protection system for NoC based MPSoCs". In: *Proceedings of the International Workshop on Security and Dependability for Resource Constrained Embedded Systems*. 2010, pp. 1–7.

[LDLM09] Dave Levin, John R Douceur, Jacob R Lorch, and Thomas Moscibroda. "TrInc: Small Trusted Hardware for Large Distributed Systems." In: *NSDI*. Vol. 9. Boston, Massachusetts, USA, 2009, pp. 1–14.

[Lee18] Dave Lee. *MyFitnessPal breach affects millions of Under Armour users*. bbc.com. Mar. 2018.

[LHBF14] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. "SoK: Automated Software Diversity". In: *Proceedings of the 2014 IEEE Symposium on Security and Privacy*. SP '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 276–291. DOI: 10.1109/SP.2014.25.

[Lie95] Jochen Liedtke. "On micro-Kernel Construction". In: ed. by Michael B. Jones. ACM, 1995, pp. 237–250. DOI: 10.1145/224056.224075. URL: http://doi.acm.org/10.1145/224056.224075.

[Lip+18] Moritz Lipp, Michael Schwart, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. *Meltdown (CVE-2017-5754)*. Tech. rep. ArXiv e-prints 1801.01207, Jan. 2018.

[LLOR14] Aldelir Fernando Luiz, Lau Cheuk Lung, and Luciana de Oliveira Rech. "On the practicality to implement byzantine fault tolerant services based on tuple space". In: *2014 IEEE 28th International Conference on Advanced Information Networking and Applications (AINA)*. IEEE. 2014, pp. 1041–1048.

[LSP82] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. "The Byzantine Generals Problem". In: *ACM Trans. Program. Lang. Syst.* 4.3 (1982), pp. 382–401. DOI: 10.1145/357172.357176. URL: http://doi.acm.org/10.1145/357172.357176.

[LWHH18] Adam Lackorzynski, Alexander Warg, Michael Hohmuth, and Hermann Härtig. *L4Re*. https://l4re.org/doc/index.html. 2018.

[MA09] Jeanna Neefe Matthews and Thomas E. Anderson, eds. ACM, 2009.

[Mah+19] Abdulrahman Mahmoud, Radha Venkatagiri, Khalique Ahmed, Sasa Misailovic, Darko Marinov, Christopher W Fletcher, and Sarita V Adve. "Minotaur: Adapting Software Testing Techniques for Hardware Errors". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM. 2019, pp. 1087–1103.

[Man86] Luigi Mancini. "Modular redundancy in a message passing system". In: *IEEE Transactions on Software Engineering* 1 (1986), pp. 79–86.

[Mat+14] Rivalino Matias, Marcela Prince, Lúcio Borges, Claudio Sousa, and Luan Henrique. "An Empirical Exploratory Study on Operating System Reliability". In: *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. SAC '14. Gyeongju, Republic of Korea: ACM, 2014, pp. 1523–1528. DOI: 10.1145/2554850.2555021. URL: http://doi.acm.org/10.1145/2554850.2555021.

[Mes07] Jeanne Meserve. *Mouse click could plunge city into darkness, experts say*. http://edition.cnn.com/2007/US/09/27/power.at.risk/index.html. Accessed: 2017-03-12. 2007.

[MMRT03] Dahlia Malkhi, Michael Merritt, Michael K Reiter, and Gadi Taubenfeld. "Objects shared by Byzantine processes". In: *Distributed Computing* 16.1 (2003), pp. 37–48.

[Moo+65] Gordon E Moore et al. *Cramming more components onto integrated circuits*. 1965.

[MR98] Dahlia Malkhi and Michael Reiter. "Byzantine quorum systems". In: *Distributed computing* 11.4 (1998), pp. 203–213.

[NB13] Ruslan Nikolaev and Godmar Back. “VirtuOS: An Operating System with Kernel Virtualization”. In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP ’13. Farminton, Pennsylvania: ACM, 2013, pp. 116–132. DOI: 10.1145/2517349.2522719. URL: http://doi.acm.org/10.1145/2517349.2522719.

[NW74] Roger M. Needham and Maurice V. Wilkes. “Domains of Protection and the Management of Processes”. In: *The Computer Journal* 17.2 (1974).

[OAHY08] Simon Ogg, Bashir Al-Hashimi, and Alex Yakovlev. “Asynchronous Transient Resilient Links for NoC”. In: *Proceedings of the 6th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. CODES+ISSS ’08. Atlanta, GA, USA: ACM, 2008, pp. 209–214. DOI: 10.1145/1450135.1450182. URL: http://doi.acm.org/10.1145/1450135.1450182.

[OW02] Thomas J. Ostrand and Elaine J. Weyuker. “The Distribution of Faults in a Large Industrial Software System”. In: *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA ’02. Roma, Italy: ACM, 2002, pp. 55–64. DOI: 10.1145/566172.566181. URL: http://doi.acm.org/10.1145/566172.566181.

[OWB04] Thomas J. Ostrand, Elaine J. Weyuker, and Robert M. Bell. “Where the Bugs Are”. In: *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA ’04. Boston, Massachusetts, USA: ACM, 2004, pp. 86–96. DOI: 10.1145/1007512.1007524. URL: http://doi.acm.org/10.1145/1007512.1007524.

[Pal+14] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Gilles Muller, and Julia Lawall. “Faults in Linux 2.6”. In: *ACM Trans. Comput. Syst.* 32.2 (June 2014), 4:1–4:40. DOI: 10.1145/2619090. URL: http://doi.acm.org/10.1145/2619090.

[PE12] BP Prabahar and BE Edwin. “Survey on virtual machine security”. In: *International Journal of Advanced Research in Computer Engineering Technology (IJARCET)* 1.8 (2012), pp. 115–121.

[PG05] David Patterson and Archana Ganapathi. “Crash Data Collection: A Windows Case Study”. In: *3D Digital Imaging and Modeling, International Conference on* (2005), pp. 280–285. DOI: 10.1109/DSN.2005.32.

[PKCC17] N Prasad, Rajit Karmakar, Santanu Chattopadhyay, and Indrajit Chakrabarti. "Runtime mitigation of illegal packet request attacks in Networks-on-Chip". In: *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE. 2017, pp. 1–4.

[Pri19] Rob Price. *Facebook says it 'unintentionally uploaded' 1.5 million people's email contacts without their consent*. Businessinsider.com. Apr. 2019.

[Ram11] Carl Ramey. "Tile-gx100 manycore processor: Acceleration interfaces and architecture". In: *2011 IEEE Hot Chips 23 Symposium (HCS)*. IEEE. 2011, pp. 1–21.

[Rei+16] Cezar Reinbrecht, Altamiro Susin, Lilian Bossuet, Georg Sigl, and Johanna Sepúlveda. "Side channel attack on NoC-based MPSoCs are practical: NoC Prime+ Probe attack". In: *2016 29th Symposium on Integrated Circuits and Systems Design (SBCCI)*. IEEE. 2016, pp. 1–6.

[RP19] Venkata Yaswanth Raparti and Sudeep Pasricha. "Lightweight mitigation of hardware Trojan attacks in NoC-based manycore computing". In: *2019 56th ACM/IEEE Design Automation Conference (DAC)*. IEEE. 2019, pp. 1–6.

[RS10] Tom Roeder and Fred B Schneider. "Proactive obfuscation". In: *ACM Transactions on Computer Systems (TOCS)* 28.2 (July 2010), pp. 1–54.

[RSBS16] Cezar Reinbrecht, Altamiro Susin, Lilian Bossuet, and Johanna Sepúlveda. "Gossip noc–avoiding timing side-channel attacks through traffic management". In: *2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE. 2016, pp. 601–606.

[SABL06] Michael M. Swift, Muthukaruppan Annamalai, Brian N. Bershad, and Henry M. Levy. "Recovering Device Drivers". In: *ACM Trans. Comput. Syst.* 24.4 (Nov. 2006), pp. 333–360. DOI: 10.1145/1189256.1189257. URL: http://doi.acm.org/10.1145/1189256.1189257.

[SAJB14] Ahmed Saeed, Ali Ahmadinia, Mike Just, and Christophe Bobda. "An ID and address protection unit for NoC based communication architectures". In: *Proceedings of the 7th International Conference on Security of Information and Networks*. 2014, pp. 288–294.

[Sca11] KA Scarfone. *Guide to security for full virtualization technologies*. Vol. 800. 125. DIANE Publishing, 2011.

[Sch+14] Nicolas Schiper, Vincent Rahli, Robbert Van Renesse, Marck Bickford, and Robert L Constable. "Developing correctly replicated databases using formal tools". In: *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE. 2014, pp. 395–406.

[Sep+18] Johanna Sepúlveda, Damian Aboul-Hassan, Georg Sigl, Bernd Becker, and Matthias Sauer. "Towards the formal verification of security properties of a Network-on-Chip router". In: *2018 IEEE 23rd European Test Symposium (ETS)*. IEEE. 2018, pp. 1–6.

[SK11] K Sajeesh and Hemangee K Kapoor. "An authenticated encryption based security framework for NoC architectures". In: *2011 International Symposium on Electronic System Design*. IEEE. 2011, pp. 134–139.

[SKLR11] Jakub Szefer, Eric Keller, Ruby B. Lee, and Jennifer Rexford. "Eliminating the Hypervisor Attack Surface for a More Secure Cloud". In: *Proceedings of the 18th ACM Conference on Computer and Communications Security*. CCS '11. Chicago, Illinois, USA: ACM, 2011, pp. 401–412. DOI: 10.1145/2046707.2046754. URL: http://doi.acm.org/10.1145/2046707.2046754.

[SLQP07] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. "SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes". In: *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*. SOSP '07. Stevenson, Washington, USA: ACM, 2007, pp. 335–350. DOI: 10.1145/1294261.1294294. URL: http://doi.acm.org/10.1145/1294261.1294294.

[Smi19] Richard E Smith. *Elementary information security*. Jones & Bartlett Learning, 2019.

[SNV06] Paulo Sousa, Nuno Ferreira Neves, and Paulo Verissimo. "Proactive resilience through architectural hybridization". In: *Proceedings of the 2006 ACM symposium on Applied computing*. ACM. 2006, pp. 686–690.

[Sou+10] Paulo Sousa, Alysson Neves Bessani, Miguel Correia, Nuno Ferreira Neves, and Paulo Verissimo. "Highly available intrusion-tolerant services with proactive-reactive recovery". In: *IEEE Transactions on Parallel and Distributed Systems* 21.4 (2010), pp. 452–465.

[SPWS13] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. "Sok: Eternal war in memory". In: *2013 IEEE Symposium on Security and Privacy*. IEEE. 2013, pp. 48–62.

[SS10] Ulrich Schmid and Andreas Steininger. *Decentralised fault-tolerant clock pulse generation in VLSI chips*. Patent: US7791394B2. TU Wien. 2010.

[Sun+10] Swaminathan Sundararaman, Sriram Subramanian, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Michael M. Swift. "Membrane: Operating System Support for Restartable File Systems". In: *Trans. Storage* 6.3 (Sept. 2010), 11:1–11:30. DOI: 10.1145/1837915.1837919. URL: http://doi.acm.org/10.1145/1837915.1837919.

[SYT16] Jiangyong Shi, Yuexiang Yang, and Chuan Tang. "Hardware assisted hypervisor introspection". In: *SpringerPlus* 5.1 (2016), pp. 1–23.

[SZFS17] Johanna Sepúlveda, Andreas Zankl, Daniel Flórez, and Georg Sigl. "Towards protected MPSoC communication for information protection against a malicious NoC". In: *Procedia computer science* 108 (2017), pp. 1103–1112.

[Tec19] Infineon Technologies. *AURIX System Architecture*. https://www.infineon.com/dgdl/Infineon-AURIX_System_Architecture-Training-v01_00-EN.pdf?fileId=5546d46269bda8df0169ca92d6362599. 2019.

[TLS04] Pascal Traverse, Isabelle Lacaze, and Jean Souyris. "Airbus fly-by-wire: A total approach to dependability". In: *Building the Information Society*. Springer, 2004, pp. 191–212.

[TN16] Ammarit Thongthua and Sudsanguan Ngamsuriyaroj. "Assessment of hypervisor vulnerabilities". In: *2016 International conference on cloud computing research and innovations (ICCCRI)*. IEEE. 2016, pp. 71–77.

[TS13] Louis Turnbull and Jordan Shropshire. "Breakpoints: An analysis of potential hypervisor attack vectors". In: *2013 Proceedings of IEEE Southeastcon*. IEEE. 2013, pp. 1–6.

[Tsi18] Joseph Tsidulko. *The 10 Biggest Cloud Outages Of 2018*. https://www.crn.com/slide-shows/cloud/the-10-biggest-cloud-outages-of-2018. Dec. 2018.

[VB+17] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. "Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution". In: *26th {USENIX} Security Symposium ({USENIX} Security 17)*. 2017, pp. 1041–1056.

[VCBL09] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. “Spin One’s Wheels? Byzantine Fault Tolerance with a Spinning Primary”. In: *Proceedings of the 2009 28th IEEE International Symposium on Reliable Distributed Systems*. SRDS ’09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 135–144. DOI: 10.1109/SRDS.2009.36. URL: https://doi.org/10.1109/SRDS.2009.36.

[Ver+13] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Verissimo. “Efficient Byzantine Fault-Tolerance”. In: *IEEE Transactions on Computers* 62.1 (Jan. 2013), pp. 16–30. DOI: 10.1109/TC.2011.221. URL: http://dx.doi.org/10.1109/TC.2011.221.

[Ver06] Paulo E Verissimo. “Travelling through wormholes: a new look at distributed systems models”. In: *ACM SIGACT News* 37.1 (2006), pp. 66–81.

[Wai+97] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. “Baring it all to Software: Raw Machines”. In: *IEEE Computer* (Sept. 1997), pp. 86–93.

[Was+13] Hassan MG Wassel, Ying Gao, Jason K Oberg, Ted Huffmire, Ryan Kastner, Frederic T Chong, and Timothy Sherwood. “SurfNoC: A low latency and provably non-interfering approach to secure networks-on-chip”. In: *ACM SIGARCH Computer Architecture News* 41.3 (2013), pp. 583–594.

[WJM08] Wayne Wolf, Ahmed Amine Jerraya, and Grant Martin. “Multiprocessor system-on-chip (MPSoC) technology”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27.10 (2008), pp. 1701–1713.

[Woo+14] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. “The CHERI Capability Model: Revisiting RISC in an Age of Risk”. In: *Proceeding of the 41st Annual International Symposium on Computer Architecuture*. ISCA ’14. Minneapolis, Minnesota, USA: IEEE Press, 2014, pp. 457–468.

[WS12] Yao Wang and G Edward Suh. “Efficient timing channel protection for on-chip networks”. In: *2012 IEEE/ACM Sixth International Symposium on Networks-on-Chip*. IEEE. 2012, pp. 142–151.

[XCP15] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. “Controlled-channel attacks: Deterministic side channels for untrusted operating systems”. In: *2015 IEEE Symposium on Security and Privacy*. IEEE. 2015, pp. 640–656.

[Xen] *Recently reported Xen/Critix Hypervisor vulnerabilities, documented in CVE-2019-18420, CVE-2019-18421, CVE-2019-18424, CVE-2019-18425.*

[Yam+21] Toshihiro Yamauchi, Yohei Akao, Ryota Yoshitani, Yuichi Nakamura, and Masaki Hashimoto. “Additional kernel observer: privilege escalation attack prevention mechanism focusing on system call privilege changes”. In: *International Journal of Information Security* 20.4 (2021), pp. 461–473.

[Yan+16] Pengfei Yang, Quan Wang, Wei Li, Zhibin Yu, and Hongwei Ye. “A Fault Tolerance NoC Topology and Adaptive Routing Algorithm”. In: *2016 13th International Conference on Embedded Software and Systems (ICESS)*. Aug. 2016, pp. 42–47. DOI: 10.1109/ICESS.2016.20.

[Yeh98] Ying C. Yeh. “Triple-triple redundant 777 primary flight computer”. In: *1996 IEEE Aerospace Applications Conference. Proceedings*. Vol. 1. IEEE. 1998, pp. 293–307.

[YF13] Qiaoyan Yu and Jonathan Frey. “Exploiting error control approaches for hardware trojans on network-on-chip links”. In: *2013 IEEE international symposium on defect and fault tolerance in VLSI and nanotechnology systems (DFTS)*. IEEE. 2013, pp. 266–271.

[Yin+03] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. “Separating agreement from execution for Byzantine fault tolerant services”. In: *Proceedings of the nineteenth ACM symposium on Operating systems principles*. 2003, pp. 253–267.

[Yin+19] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. “HotStuff: BFT consensus with linearity and responsiveness”. In: *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. 2019, pp. 347–356.

[Yus19] Najeer Yusof. *Personal data of 808,000 blood donors compromised for nine weeks; HSA lodges police report*. TODAYonline. Mar. 2019.

[ZD20] Dina Zoughbi and Nitul Dutta. “Hypervisor Vulnerabilities and Some Defense Mechanisms, in Cloud Computing Environment”. In: *International Journal of Innovative Technology and Exploring Engineering* 10 (Dec. 2020), pp. 42–48. DOI: 10.35940/ijitee.B8262.1210220.

[Zha16] Guangda Zhang. *Fault Tolerant Techniques for Asynchronous Networks on Chip*. The University of Manchester (United Kingdom), 2016.

[Zho+06] Feng Zhou, Jeremy Condit, Zachary Anderson, Ilya Bagrak, Rob Ennals, Matthew Harren, George Necula, and Eric Brewer. "SafeDrive: Safe and Recoverable Extensions Using Language-based Techniques". In: *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*. OSDI '06. Seattle, WA: USENIX Association, 2006, pp. 4–4. URL: http://dl.acm.org/citation.cfm?id=1267308.1267312.

[ZKDK08] Nickolai Zeldovich, Hari Kannan, Michael Dalton, and Christos Kozyrakis. "Hardware Enforcement of Application Security Policies Using Tagged Memory." In: *OSDI*. Vol. 8. 2008, pp. 225–240.

[McC+10] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. "TrustVisor: Efficient TCB Reduction and Attestation". In: *2010 IEEE Symposium on Security and Privacy*. May 2010, pp. 143–158. DOI: 10.1109/SP.2010.17.