Behind the Last Line of Defense

Surviving SoC Faults and Intrusions

Inês Pinto Gouveia, Marcus Völp and Paulo Esteves-Verissimo
University of Luxembourg
Interdisciplinary Center for Security, Reliability and Trust (SnT) - CritiX group
ines.gouveia@uni.lu, marcus.voelp@uni.lu, paulo.verissimo@uni.lu

***Abstract*—Today, leveraging the enormous modular power, diversity and flexibility of manycore systems-on-a-chip (SoCs) requires careful orchestration of complex resources, a task left to low-level software, e.g. hypervisors. In current architectures, this software forms a single point of failure and worthwhile target for attacks: once compromised, adversaries gain access to all information and full control over the platform and the environment it controls. This paper proposes *Midir*, 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 mechanisms, while securing low-overhead quorum-based consensus on all critical operations, in particular privilege management and, thus, management of containment domains. Allowing versatile redundancy management, *Midir* promotes resilience for all software levels, including at low level. We explain this architecture, its associated algorithms and hardware mechanisms and show, for the example of a Byzantine fault tolerant microhypervisor, that it outperforms the highly efficient MinBFT by one order of magnitude.**

***Index Terms*—fault and intrusion tolerance, hypervisor, processor architecture**

I. Introduction

Practically all activity of modern societies depends on information and communication technologies (ICT). Such dependency obviously hinges on the correctness of these systems, some of them critical, which may fail in a combination of multiple causes and ways [1]–[6]. Systems have been progressively pushed to extremes of efficiency through modularity in platform sharing, firstly through virtualization and lately by leveraging the enormous power growth, functional diversity and adaptation flexibility offered by multi- and manycore. This has taken platform sharing to new heights, into the realm of multi-processor systems-on-a-chip (MPSoCs).

The organization of these complex computing resources depends on low-level platform management hardware (e.g., memory-management units (MMUs)) and software (e.g., firmware, hypervisors, management engines). 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 (*SPoF*), for two main reasons. First, the way platform privilege-enforcement mechanisms (e.g. MMUs or hardware-enforced capabilities [7]) are designed allows faults in a core/tile to propagate through MPSoC components. Second, 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.

If these SPoFs are compromised by adversaries, the latter gain full authority over the platform's privilege-enforcement mechanisms and, through them, 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 or autonomous cars).

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 [8], Xen/Critix [9] or concerning Spectre [10] and Meltdown [11], have been repeatedly reminding us of how brittle the assumption of "tamperproof and unattackable low-level platform management assets" is. Even formally verified kernels (e.g., seL4 [12]) may fail due to model/reality discrepancies or hardware faults violating modeling assumptions [13].

Being the risk real, are there no solutions yet? The solution design space for contemporary hardware platforms dependability and security has been unfolding in two directions: (i) application-specific system-level replication (e.g., triple modular redundancy, mainly in cyber-physical systems (CPS), by means of multiple electronic control units (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.

At this time, we call the reader's attention to an interesting fact, which will become crucial to our solution. The current MPSoC architectures' complexity, modularity and networked interconnectivity, suggests attributes of distributed systems [14], albeit imperfect such systems (an example of which is the aforementioned SPoF syndrome). However, distributed systems have been used to mitigate SPoF syndromes and to implement fault and intrusion tolerance schemes [15], [16]. In consequence, the root of the MPSoC problems just presented may also be an avenue to their solution.

So, in this paper, we start by identifying the gaps from (MP)SoCs to distributed systems, and propose (MP)SoC mechanisms to bridge them, which essentially means achieving: fault independence and fault containment, despite low software-level compromise and while retaining the flexibility (MP)SoCs offer. Having a manycore that behaves as a (closely-coupled) distributed system, should allow us to design a set of

efficient and low-overhead distributed systems-inspired modular protection and redundancy management mechanisms, e.g., Byzantine fault tolerant state machine replication (BFT-SMR), for fault and intrusion tolerance (FIT). The remaining problem, how to implement and where to locate all the mechanisms above, is addressed by the *Midir*[1] achitecture presented in this paper, which leverages the computing critical mass and flexibility of contemporary tile-based manycore architectures.

Midir constrains the connection of all tiles to the network-on-chip (NoC) through simple and self-contained hardware-based trusted components, which we call *T2H2*. Exploring the concept of architectural hybridization [17], 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.

The *T2H2* components implement the functionality achieving fault independence, containment, and tolerance mechanisms mentioned above. In consequence, tile-internal software or hardware faults are contained in the tile and the objects the tile can access. Furthermore, the baseline mechanisms for protection and redundancy management provided by *T2H2* 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).

Locating *T2H2* between the tile and the NoC interconnect not only provides a clear pathway for integration by chip manufacturers and integrators, it also allows drawing from many well-understood building blocks (e.g., region protection, capabilities [18], and other chip-level resource management mechanisms [19], capable of isolating tiles and the resources they can access). The novelty of *Midir* lies in their arrangement to avoid SPoFs, even while they are reconfigured.

In a nutshell, contributions of this paper are:

(1) An analysis of the gaps separating current MPSoC architectures from genuine distributed systems, and gap fixing through measures promoting fault independence and fault containment in tile-based architectures, enforced at the level of the tile-to-NoC interface.

(2) An architecture (*Midir*) leveraging the resulting distributed system-on-a-chip (DSoC) to achieve incremental levels of modular fault and intrusion tolerance, through a range of diverse redundancy management techniques implemented by simple hardware-based voting/consensus mechanisms.

(3) The design of a simple and ultimately **t**rusted-**t**rustworthy **h**ardware **hy**brid, *T2H2*— the core component of *Midir*, staged at the tile-to-NoC interface — providing just two generic baseline functions: access control (capability registers) and quorum-based consensus (voters). By configurations and combinations of these two basic functions, *T2H2* is capable of implementing all the techniques mentioned in (1) and (2).

(4) As a proof of concept, we give and evaluate an implementation featuring *Midir* and essential parts of a fault and intrusion tolerant microhypervisor built on top of it. Though the architecture serves several reliability strategies, we chose the most effective, active replication with error masking. Being the most complex and costlier, we believe to have shown the performance and practicality of our concept.

Next, we evaluate the challenges for bridging from SoCs to DSoCs (Sec.II), and present the system and threat model (Sec.III). Then, we introduce the *Midir* architecture (Sec.IV) and the *T2H2* component in Sec.V. At this point, we are able to show *Midir* in action, discussing the design of a fault and intrusion tolerant microhypervisor built on top of it (Sec.VI), as an example of critical low-level management software. Finally, we discuss some relevant implementation matters in Sec. VII, and in Sec.VIII, we evaluate *Midir* on a Zynq ZC702 board, showing how *Midir*'s hardware voters accelerate BFT-SMR protocols, voted execution of system calls and consensual reconfiguration of *T2H2*. An analysis of related work (Sec. IX) follows, and Sec.X concludes the paper, pointing to further research and innovation opportunities.

II. From MPSoCs to Distributed SoCs

Multi- and manycore systems consolidate in a single chip computing resources that used to reside on multiple chips. Tiles [20] 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.

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 [21], Cap [18], M3 [22] and others. Hive introduces MAGIC, a bus-level firewall to confine faults to the individual processors of the Stanford Flash multiprocessor system. M3 follows the same scheme with hardware enforced capabilities, originally introduced in Cap [18] to control resource accesses and, thereby, fault containment of heterogeneous processors. Configurable isolation [19] leverages dual-mode redundant MMUs to, like M3, confine faults in on-chip resources. Tiles favour functional and non-functional diversity since they can host cores from several makers. 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 with the same intent [23]–[25].

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.

A final and subtle gap concerning fault containment and independence affects all previous systems we know of, including those deploying hardware-enforced fault containment [18], [19], [21], [22]: potentially faulty or compromised low-level kernels still retain control over platform privilege configuration

[1]pronounced meedir

mechanisms. As we explain in Sec.IV, this is a harmful effect. Our main contribution is to neutralize this effect by imposing that critical platform management operations are performed through consensus of a majority of correct components.

In conclusion, with the enhancements described in this paper, tiles fail like nodes in a distributed system, faults affect only the tile itself and the components (e.g., replicas) executing on it, but they do not propagate to the entire manycore, in particular other components related to the same application or subsystem. This interplay between protection and consensus to achieve fault containment, in particular during platform reconfiguration, including of the fault containment domains themselves, allows hypervisor replicas to retain the flexibility of the MPSoC, even after a minority of hypervisor tiles failed accidentally or have been compromised by an adversary.

III. System and Threat Model

We now describe the system and threat model educating the development of our distributed system-on-a-chip (DSoC).

A. System Model

We assume a fully connected system, where on-chip network components offer the abstraction of a correct network, interconnecting all tiles to one another. Tiles communicate by messages, and messages sent are eventually delivered, unchanged, to the destination. Network coding [26], multi-tenant [27] and adaptive routing techniques [28] substantiate the coverage of this assumption.

We rely on a partially-synchronous model. At first sight, manycores might seem the perfect example of a (closely-coupled) synchronous (distributed) system. However, reality is a bit different, several possibilities for instability in the time domain (speed of tiles throttling for thermal control, cache exceptions, NoC-level bursts, etc.) would prove the strict synchronous model brittle.

However, being a closely-coupled environment, short-term liveness is normally guaranteed, barring delay variations. This has two implications on the design of *Midir*, for robustness: (i) we absorb possible inter-tile delays, notably by buffering messages (e.g., votes) in *Midir*'s *T2H2*; (ii) the structure of the protocols is time-free and, as such, they remain safe in the presence of delay oscillations, provided that the fault assumptions hold.

B. Threat Model

Our threat model considers software-level compromise at all levels, including hypervisors, firmware and, more generally, in any critical software component. This assumption is consistent with our aim of tolerating an incremental level of threat on tiled manycore systems, up to sophisticated and persistent attacks possibly 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).

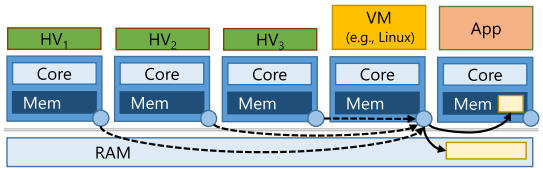

Fig. 1. Example systems showing the *Midir* architecture: software in tiles need a capability to authorize access to resources in other tiles (solid lines); capability modifications in a tile (in fact any critical operation) are subject to consensus of a majority of other correct tiles (dashed lines), here the three tiles hosting hypervisor replicas of which one may be faulty.

We consider 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 in case the user/supervisor mode isolation failed). Once that happens, we no longer make any assumptions about the correctness of any software in that tile. However, we also consider (and enforce it with the strategy described in Sec.II) that tiles themselves are fault containment domains, such that faults inside a tile do not propagate across the manycore.

We enforce the assumption above through architectural hybridization [17], [29], [30]. Despite the general system fault model enunciated for tiles, *T2H2* (*Midir*'s trusted-trustworthy component) follows a more restricted fault model, enforced by construction and, through its simplicity, amenable to verification, failing only by crashing, much like USIG [29] or CASH [30]. Thus, *T2H2*, residing at the tile-to-NoC interface, reliably implements its functions despite faulty tiles.

IV. The *Midir* Architecture

As discussed earlier, *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 (*T2H2*). In fact, *T2H2* provides just two generic baseline functions staged *in hardware* at the tile-to-NoC interface: access control (capability registers) and quorum-based consensus (voters).

Fig.1 depicts one possible layout, of a stereotypical hypervisor-based system, where the hypervisor is replicated for fault/intrusion tolerance, serving operating system and applications: hypervisor replicas are distributed across tiles, so that each replica executes on a different tile, separate from applications; tiles and software therein interface with each other through the NoC; and *T2H2* are the "blue dots" performing that interconnection. *T2H2* interposes such accesses, validating that the invoking tile has sufficient privileges, through the capability registers, which include the logic for privilege enforcement.

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 *T2H2* 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*.

This is like the internal behavior of nodes in a distributed system, where nodes are the unit of fault containment. 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 IO) and *T2H2* validates that each such access has been authorized by a capability the tile possesses. Consequently, hardware faults inside a tile or accidential or malicious faults in any part of the software it executes, are limited in propagation to the objects authorized by these capabilities.

Further to capability checking, *Midir* is capable of subjecting these accesses to voting by distributed components 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 hypervisor replicas from bypassing the aforementioned fault containment when reconfiguring the resources a tile can access.

Midir's concept of controlling the tiles' lowest-level privilege enforcement mechanism is agnostic of 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. Hence our choice for capabilities.

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 operation 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 components to perform these tasks. More precisely, voters suspend voting on disagreement, freeze the proposals made by the components 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. Sec.VI shows how we utilize this error handling support and Sec.VII details our voter implementations.

V. *T2H2*– Midir's Trusted-Trustworthy Component

In this section, we provide further details about *T2H2*.

A. 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, *T2H2* 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

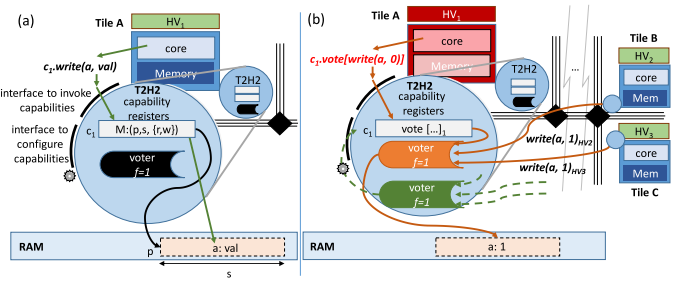

Fig. 2. (a) Non-voted memory access by tile A through capability invocation. (b) Voted memory access by tiles A-C (tile A faulty) through capability invocation then voting (orange); reconfiguration of a platform capability register in tile A through voting (green).

interact with non-critical devices). The scenario in Fig.2(a) illustrates a non-voted (write) memory access by Tile A, performed by invoking a capability in this tile's *T2H2*. Since *T2H2*'s capability registers hold a read-write capability to the memory region $[p, p+s]$, the operation to write value val in variable a is authorized.

However, *T2H2* also supports voting, particularly useful when e.g., platform management software or hypervisor replicas, further have to 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 *f-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. Fig.2(b) represents a similar operation as in Fig.2(a), but in voted access form. Tiles B and C vote to write value 1, while Tile A, being faulty, votes to write value 0. In order to perform these votes, all tiles invoke a capability on their local *T2H2* to access the designated voter (in this case, residing on Tile A's *T2H2*) . Given that a majority of tiles voted to write 1, value 1 will be written to variable a.

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.

B. Consensual privilege change

One particularly relevant scenario for voted access is consensual reconfiguration of the *T2H2* instances themselves. *T2H2*'s reconfiguration interface 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 [31]) or firmware, as in Intel SGX [32]), it becomes a single point of failure for the platform.

Midir provides a further 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 Sec. V-A, 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 Fig.2(b), in green colour, we represent such a flow of reconfiguration of a platform capability register in tile A's *T2H2*. Exemplifying with *f-out-of-n* error masking in a replicated low-level kernel, several replicas make the reconfiguration request, which is voted (green voter). The result from the voter is wired through a special *T2H2* capability configuration interface to the concerned capability register, masking the presence of up to f faulty replicas.

VI. Towards Fault and Intrusion Tolerant Microhypervisors

We now turn our attention to the construction of *Midir*-aware FIT microhypervisors, such as suggested in Fig. 1. 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.

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[2] by subjecting all critical operations to voting. We focus on this facet. The replicated microhypervisor 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).

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 microhypervisor design is that the replicas behind critical operations are placed in different tiles, such that they communicate by messages, are subject to *T2H2* 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 (*T2H2* voting relies on identifying the individual replicas through their capability).
- **P.2** ***Bypass prevention*** Correct replicas must deny any operation attempting to grant direct write access to a consensual-update-only object (Sec.V-B).

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) is a critical operation, 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 application has invoked this call. A further role of the service loop is therefore to reach consensus on system call execution order and parameters. From our evaluation (Sec. VIII) we found that *Midir*'s support for consensually executing critical operations also provides for accelerating the BFT protocol that the kernel replicas must execute to reach this agreement.

A. Consensual System Calls

Fig.3 provides a more detailed picture of how *T2H2*'s voters and capability registers contribute to a FIT hypervisor's service loop reaching consensus on the system call to execute.

The service loop utilizes two data structures: a consensually updated ringbuffer — the *syscall log* — records agreed upon system calls and its 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 there, but refuse to execute the system call later on, too few correct

[2]Modulo *Midir*'s *T2H2*, which, justified through its simplicity, we assume will not fail.

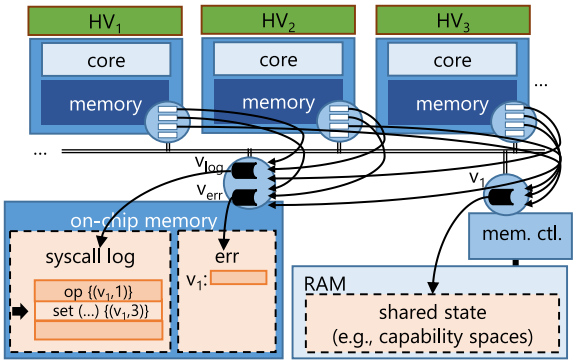

Fig. 3. Read-shared, consensually updated data structures used by the kernel: system calls are recorded in the syscall log, the error log keeps voting error information, a capability space holds an application's capabilities (Sec.VIII).

```
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
```

Fig. 4. Generic voting pattern used in the service loop and when executing system calls.

replicas would have obtained this knowledge to complete the system call. Similarly, the service loop utilizes an *error log* to protect error information from getting lost in premature resets of the voter. 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 kernel replicas poll[3] for new requests. Consensual privilege change allows creating such buffers by granting write access to a single client, but to no kernel replica. The leading kernel replica proposes one such system call by initiating a vote with v_{log}, which followers introspect and agree or deny. Once written to the syscall log, replicas proceed by executing the system call and the votes for its critical operations. 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 [33]. In our setting, cryptographic operations would come at overproportionally high costs relative to the speed of the transport medium (the NoC). 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. Additionally, clients maintain write access to their request buffers. Thus, 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 error diagnosis for the system-call vote to regain it when we need it: in the subordinate votes for reaching agreement on critical operations.

The following details the protocols the hypervisor replicas execute to reach consensus on and execute system calls. Leveraging the generic voting pattern in Fig.4, replicas first reach agreement on the system call (Fig.5) to then consensually perform critical updates during its execution (Fig.6).

B. Generic Voting Pattern

Fig.4 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 identifier $v_i.seq \bmod n$ 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 Sec.VII)).

C. 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. Fig.5 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 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:

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

```
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
```

Fig. 5. Service loop - Phase 1: agree on next system call to execute

- **P.3** ***Coordinated subordinate votes:*** correct replicas vote only on subordinate voters ($v_i \in V$) to execute the current system call.
- **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 (c.f., P.1 in Sec.VI). 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 Sec.VI-A, 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.

```
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)
```

Fig. 6. System call execution - Phase 2: subordinate votes and error handling

D. Subordinate Votes

The code for executing subordinate votes in Fig.6 has to solve two problems: (i) preserve determinism despite errors and (ii) 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. 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: (i) 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. (ii) 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 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.

VII. Implementation

The implementation of capability invocation is standard (c.f. [18]): *T2H2* intercepts external operations, 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 [34], which *T2H2* inserts as additional parameter into the operation.

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

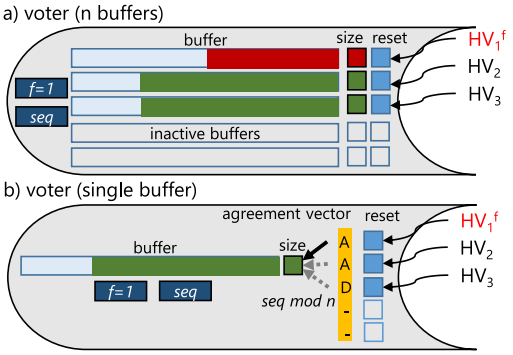

Fig. 7. Internal structure of a voter. One, resp. n buffers hold the message of replicas to vote upon and $size$ its length. f defines the fault threshold, seq is a voter maintained sequence number. The agreement and reset vector are described below.

A. 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[4]. Buffering votes is ideal in a NoC architecture, since votes are transmitted as normal messages. Tiles can continue executing once the message is sent. We therefore implement voters to contain buffers for storing proposals from the different replicas for the current vote executed with this voter.

B. 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 not have to repeat the vote.

We implemented two variants of *T2H2* voters to evaluate the resource/performance trade-off at the two extremes of this spectrum. Our n-buffer variant (Fig.7 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. Our single-buffer variant (Fig.7 b) trades agreement time for a more resource-efficient implementation: there is only one buffer; and only the current leader is granted write access to this buffer. The single-buffer

[4] To simplify monitoring of the progress of a system call, we shall later require 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.

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. This allows follower replicas to introspect 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.

C. 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 by a 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 introspect 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. P.5 and the protocol in Sec.VI-D handles this corner case.

D. 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 n_{max} buffers, assuming n-buffer voters). 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 *T2H2* can be implemented and shown correct, as well as stay functional even when the tile it is associated with fails.

VIII. Evaluation

As an early validation of our proposal, we have implemented *T2H2* in 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 *T2H2*, connecting the tiles through *T2H2* with an AXI interconnect (serving as the NoC). We measured the performance of the service loop (Fig. 5) to agree on and execute client-invoked system calls for granting and priming capabilities. Grant (`L4.map` [35]) copies capabilities between capability spaces and prepares for later revocation. Prime consensually copies a capability from the client's capability space into a *T2H2* capability register, where it is ready for invocation. We have measured the performance of grant and prime in two different implementations of capability spaces[5]: (i) as a private data structure in each replica, 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, 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 kernel (horizontal line), executing the same system calls on its private state, 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.

1) Per-Replica Capability Space: Figure 8 shows the average performance of the **grant** and **prime** system calls in a per-replica capability space implementation relative to the two baselines: **null** and a singleton kernel instance performing these system calls in a non-consensual manner. Shown are the system calls broken down into individual votes and the Q5 / Q95 percentiles of the overall measurements.

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, which is the baseline of the singleton kernel. System calls for the single buffer version have a factor 8.9 – 9.6 increase, which can be explained due to the voter not benefiting from caching. Whereas the singleton kernel merely has to copy one request from the memory where the client core places it, missing in all caches in the process,

[5]Container object for an application's capabilities.

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

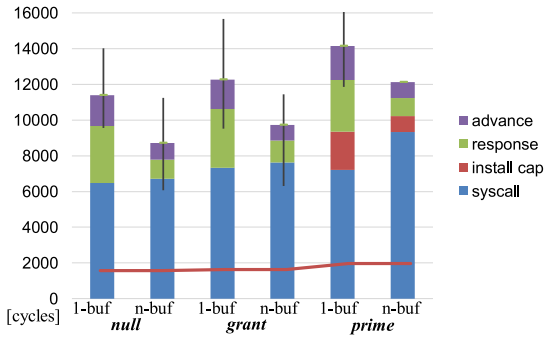

Fig. 8. Average execution times of the three consensual system calls — **null**, **grant** and **prime** — when executed on a per-replica capability space implementation. System calls are broken down into the individual votes for agreeing on the system call and for performing the critical updates required. Shown are also the Q5 / Q95 percentile and the average costs of executing the respective system calls on a singleton-kernel.

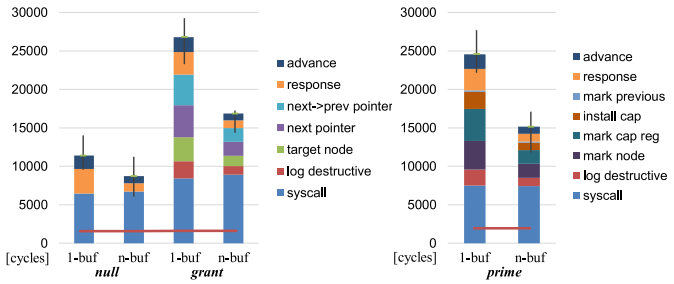

Fig. 9. Average execution times of the three system calls for consensually updated capability spaces.

following 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.

2) Consensually Updated Capability Space: Figure 9 shows a similar diagram as Figure 8, this time, however, for consensually updated capability spaces. Granting and priming capabilities now require additional votes to update the data structure.

Again, the 6.7 (/ 7.3) times slower performance relative to the singleton kernel can be explained due to the voter not benefiting from caching:

Singleton kernel: System call execution is triggered by the client writing to shared memory on one core and the kernel (on another core) reading it. From then on, all the operations

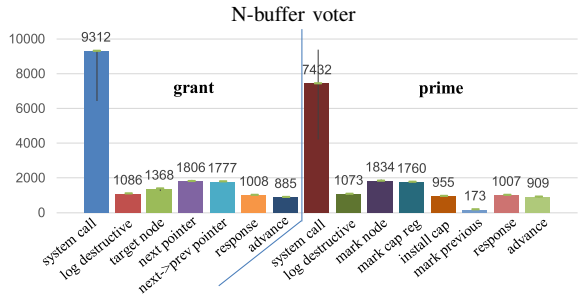

Fig. 10. System calls broken down into individual votes. Shown are the Q5 and Q95 percentile for the main system call vote and each subordinate vote for n-buffer voters. The variations for single-buffer voters are similar.

	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

Fig. 11. Code size in lines of C++ / VHDL code (logic/total).

happen locally in the core of the kernel 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 kernel: System call execution starts as well with invocation (1), but then, the leader needs to propose the request (2), followers validate it and (3) express agreement (4) upon which the voter updates the memory and all replicas wait for the vote to reach agreement (5). In (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 kernel 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 percentile. Fig. 10 shows these values. As expected, subordinate votes remain close to their average execution times, whereas agreement on the system call varies significantly.

Fig. 11 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 and the overall design of the voter and capability unit. As can be seen, the amount of code that each replica executes for the above grant

	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

Fig. 12. FPGA resources required by *T2H2* (without / with AXI interface).

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 and for mapping the corresponding internal signals.

Fig.12 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 T2H2 will not increase significantly if more complex cores are to be controlled. Hence, the relative overhead will shrink when more complex tiles are considered.

IX. Related Work

In this section, we present several classes of works that motivated *Midir*: low-level approaches for detection and containment of errors in low-level support software; analyses of the evolution of defects in system support software; attempts at preventing and/or mitigating the resulting errors and potential failures; approaches to replication-based fault/intrusion tolerance and resilience.

Mitigation measures have been studied for detection and containment of errors in OS and manycore support software [36]–[38] through an underlying, assumed-trustworthy layer. However, they still 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 [39], "simple" components with at least a few KLOCs have a non-negligible statistical fault footprint. Other studies [40], [41] reveal between 1–16 bugs per 1,000 lines of code go undetected before deployment, even in well-tested software, and operating-system kernels form no exception [42], [43]. Recent insights [44] reveal that faults in stateful core subsystems — on which we focus here — outrank driver bugs in severity. Minotaur introduces a toolkit to improve the analysis of software vulnerability to hardware errors by leveraging concepts from software testing [45].

Many approaches target operating systems with the goal of improving their resilience against faults. However, typically they protect either applications [46]–[48] or specific OS subsystems [49]–[52] and only from accidental faults. Efforts for providing whole-OS fault tolerance include [53]–[59]. Furthermore, the complexity of these recovery kernels is comparable to that of a small hypervisor. For example, OSIRIS [57] directs OS recovery to a 29 KLOC reliable computing base (RCB) [60], roughly twice the size of modern microkernels [12], [22], [35], [61]. Again, this makes the likelihood of residual faults or vulnerabilities non-negligible.

Several other works have given early steps in the direction of the solutions we advocate in this paper, minimizing the threat surface, or enforcing isolation. Nohype [62] removes all but a small kernel substrate from application cores, which run functionality-rich OSs in virtual machines (VMs), reducing the threat surface. Cap [18] and M3 [22] exploit hardware capability units and Hive [21] a bus-level firewall to isolate VMs at tile granularity. However, although this avoids 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. We address this problem in *Midir*.

Cheri [7] adds capability protection on top of page-based protection, but includes the MMU and the OS page-table management in the reliable computing base (RCB), which means the former must be trustworthy. The concept behind *Midir* is independent of the protection model, not being necessarily tied to e.g., capabilities. Also, establishing the fault containment domains at the granularity of tiles, we are agnostic about the semantics and interplay of tile-internal and/or core-level components, e.g., MMUs, memory protection or page-table management. Enforced by *T2H2*, the protection mechanisms are crafted at inter-tile level, emulating the spacial isolation of distributed system nodes.

Replication has been used before in closely-coupled systems, primarily to tolerate accidental faults in cyber-physical systems (CPS), by replicating controllers to form triple modular redundant (TMR) units, or duplicated self-checking units. 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 [63]. 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 [64]. The concept was extended to multi-phase tightly synchronous message-passing protocols still in the CPS domain [65], [66]. The so-called 'Paxos' [67], and 'Byzantine' [68] Fault-Tolerant State-Machine Replication classes of protocols promote resilience to threats, respectively accidental, and both accidental and malicious, extending the concept to generic classes of applications, namely in loosely-coupled systems. For example, Castro's seminal BFT-SMR protocol [68] 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. Behind all the categories of techniques above is a baseline voting mechanism amongst the values proposed by a pre-defined number of replicated fault-independent components. *Midir* offers such baseline mechanism at a low enough

level of abstraction to serve essentially any replication-oriented application.

Architectural hybridization [17] (i.e., the inclusion of trusted-trustworthy components that follow a differentiated fault model) allows reducing n and $|Q|$ to $2f + 1$ and $f + 1$, respectively [30], [69]–[71]. The implementation of *T2H2*, the *Midir* hybrid, draws from these quorum reduction results, and further accelerates the BFT-SMR protocol that *Midir*-enabled FIT microhypervisors use to coordinate system-call execution (Sec. VI).

Paxos and BFT replication have been attempted as well inside MPSoCs [38], [72]–[75]. However, all these works were made under the assumption of a trusted low-level kernel (e.g., hypervisor or platform manager), which obviously is a single point of failure (SPoF). One of the key results of *Midir* lies in the realization of the distributed system-on-a-chip (DSoC) vision, which enables such replication management techniques in MPSoCs, whilst removing the SPoF syndrome of the low-level kernel.

X. Conclusions and Future Work

We have introduced *Midir*, an architectural concept which breaks new ground and opens promising avenues in the applicability and resilience of manycore architectures (MPSoC). Through minimalist mechanisms integrated in the MPSoC architecture, *Midir* frees MPSoCs from the SPoF syndrome, fulfilling the vision of *distributed* systems-on-a-chip (DSoC).

In this paper, we show in particular that *Midir*-enabled DSoCs achieve a quantum step towards off-the-shelf chip resilience, since these mechanisms 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 microhypervisor. We have shown the practicality of our concept, as having quite satisfying performance, since it outperforms the highly efficient MinBFT protocol by one order of magnitude. The low overhead of our approach shows as well large promise for future full hardware solutions.

Furthermore, *Midir* was intentionally designed as a non-intrusive extension to current chip architectures, being anchored on simple and self-contained hardware extensions. Taken up by a hardware manufacturer or integrator, it allows a backward compatible, non-fracturing evolution. We hope that our findings may be key to enhance general MPSoC architectures towards distributed DSoCs and amongst other avenues, lead to next-generation COTS resilient chips.

After this initial work, several questions remain to be answered, namely on kernel design details, rejuvenation and diversification for sustainability, and so forth, which leave ample room for future work.

REFERENCES

[1] R. Price, "Facebook says it 'unintentionally uploaded' 1.5 million people's email contacts without their consent," Businessinsider.com, April 2019.

[2] N. Yusof, "Personal data of 808,000 blood donors compromised for nine weeks; hsa lodges police report," TODAYonline, March 2019.

[3] D. Lee, "Myfitnesspal breach affects millions of under armour users," bbc.com, March 2018.

[4] J. Tsidulko, "The 10 biggest cloud outages of 2018," https://www.crn.com/slide-shows/cloud/the-10-biggest-cloud-outages-of-2018, December 2018.

[5] R. M. Lee, M. J. Assante, and T. 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, month = March, year = 2016,.

[6] "Tesla's autopilot has had its first deadly crash," https://www.wired.com/2016/06/teslas-autopilot-first-deadly-crash/, accessed: 2017-03-12.

[7] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, "The cheri capability model: Revisiting risc in an age of risk," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 457–468.

[8] M. Ermolov and M. Goryachy, "How to hack a turned-off computer – or running unsigned code in intel management engine," in *Black hat Europe*, London, UK, Sept. 2017, 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.

[9] "Recently reported xen/critix hypervisor vulnerabilities, documented in cve-2019-18420, cve-2019-18421, cve-2019-18424, cve-2019-18425."

[10] P. Kocher, D. Genkin, D. Gruss, W. Haar, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," ArXiv e-prints 1801.01203, Tech. Rep., Jan. 2018, (see also: CVE-2017-5715, -5753, CVE-2018-3693, -3640, -3639, -3665, -3615, -3620, -3646, -9056).

[11] M. Lipp, M. Schwart, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown (cve-2017-5754)," ArXiv e-prints 1801.01207, Tech. Rep., Jan. 2018.

[12] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: Formal verification of an OS kernel," in *SOSP 2009*, J. N. Matthews and T. E. Anderson, Eds. ACM, 2009, pp. 207–220. [Online]. Available: http://doi.acm.org/10.1145/1629575.1629596

[13] S. Biggs, D. Lee, and G. Heiser, "The jury is in: Monolithic OS design is flawed," in *Asia-Pacific Workshop on Systems (APSys)*. Korea: ACM SIGOPS, Aug. 2018.

[14] S. Mullender, Ed., *Distributed Systems (2Nd Ed.)*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1993.

[15] G. B. P. V. F. W. D. Powell, D. Seaton, "The delta-4 approach to dependability in open distributed computing systems," in *18th IEEE International Symposium on Fault-Tolerant Computing (FTCS)*, June 1988, pp. 246–251.

[16] P. Verissimo, N. Neves, C. Cachin, J. Poritz, D. Powell, Y. Deswarte, R. Stroud, and I. Welch, "Intrusion-tolerant middleware - the road to automatic security," *Security and Privacy, IEEE*, vol. 4, pp. 54 – 62, 08 2006.

[17] P. E. Veríssimo, "Travelling through wormholes: A new look at distributed systems models," *SIGACT News*, vol. 37, no. 1, pp. 66–81, Mar. 2006.

[18] R. M. Needham and R. D. H. Wilkes, "Domains of protection and the management of processes," *The Computer Journal*, vol. 17, no. 2, 1974.

[19] N. Aggarwal, P. Ranganathan, N. P. Jouppi, and J. E. Smith, "Configurable isolation: building high availability systems with commodity multi-core processors," in *International Symposium on Computer Architecture (ISCA)*, 2007, pp. 470–481.

[20] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, , and A. Agarwal, "Baring it all to software: Raw machines," *IEEE Computer*, pp. 86–93, Sept. 1997.

[21] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta, "Hive: Fault containment for shared-memory multiprocessors," in *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '95. New York, NY, USA: ACM, 1995, pp. 12–25. [Online]. Available: http://doi.acm.org/10.1145/224056.224059

[22] N. Asmussen, M. Völp, B. Nöthen, H. Härtig, and G. Fettweis, "M3: A hardware/operating-system co-design to tame heterogeneous manycores," in *Architectural Support for Programming Languages and Operating Systems*. Atlanta, GA, USA: ACM, April 2016.

[23] A. Avizienis, L. Chen *et al.*, "On the implementation of n-version programming for software fault-tolerance during program execution," 1977.

[24] J. C. Knight and N. G. Leveson, "An experimental evaluation of the assumption of independence in multiversion programming," *IEEE Transactions on software engineering*, no. 1, pp. 96–109, 1986.

[25] M. K. Joseph and A. Avizienis, "A fault tolerance approach to computer viruses." in *IEEE Symposium on Security and Privacy*. Oakland, CA, USA, 1988, pp. 52–58.

[26] S. Ogg, B. Al-Hashimi, and A. Yakovlev, "Asynchronous transient resilient links for noc," in *Proceedings of the 6th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, ser. CODES+ISSS '08. New York, NY, USA: ACM, 2008, pp. 209–214. [Online]. Available: http://doi.acm.org/10.1145/1450135.1450182

[27] C. Colman-Meixner, C. Develder, M. Tornatore, and B. Mukherjee, "A survey on resiliency techniques in cloud computing infrastructures and applications," *IEEE Communications Surveys Tutorials*, vol. 18, no. 3, pp. 2244–2281, thirdquarter 2016.

[28] P. Yang, Q. Wang, W. Li, Z. Yu, and H. 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.

[29] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Verissimo, "Efficient Byzantine Fault Tolerance," *IEEE Trans. Comput.*, vol. 62, no. 1, pp. 16–30, Jan. 2013. [Online]. Available: http://dx.doi.org/10.1109/TC.2011.221

[30] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schröder-Preikschat, and K. Stengel, "CheapBFT: Resource-efficient byzantine fault tolerance," in *Proceedings of the 7th ACM European Conference on Computer Systems*, ser. EuroSys '12. New York, NY, USA: ACM, 2012, pp. 295–308. [Online]. Available: http://doi.acm.org/10.1145/2168836.2168866

[31] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel, "Inktag: Secure applications on an untrusted operating system," *SIGPLAN Not.*, vol. 48, no. 4, pp. 265–278, Mar. 2013. [Online]. Available: http://doi.acm.org/10.1145/2499368.2451146

[32] V. Costan and S. Devadas, "Intel SGX explained," Massachusetts Institute of Technology, Tech. Rep., 2016, https://eprint.iacr.org/2016/086.pdf (Accessed: 2016-07-22).

[33] L. Lamport, R. E. Shostak, and M. C. Pease, "The byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, pp. 382–401, 1982. [Online]. Available: http://doi.acm.org/10.1145/357172.357176

[34] N. Hardy, "Keykos architecture," *SIGOPS Oper. Syst. Rev.*, vol. 19, no. 4, pp. 8–25, Oct. 1985.

[35] J. Liedtke, "On micro-kernel construction," in *SOSP 1995*, M. B. Jones, Ed. ACM, 1995, pp. 237–250. [Online]. Available: http://doi.acm.org/10.1145/224056.224075

[36] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, "Trustvisor: Efficient tcb reduction and attestation," in *2010 IEEE Symposium on Security and Privacy*, May 2010, pp. 143–158.

[37] A. Seshadri, M. Luk, N. Qu, and A. 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*, ser. SOSP '07. New York, NY, USA: ACM, 2007, pp. 335–350. [Online]. Available: http://doi.acm.org/10.1145/1294261.1294294

[38] B. Döbel, "Operating system support for redundant multithreading," Ph.D. dissertation, Technische Universität Dresden, Dresden, Germany, Nov. 2014.

[39] M. Hoffmann, C. Dietrich, and D. 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)*, G. S. of Informatics, Ed., 2013. [Online]. Available: http://www4.cs.fau.de/Publications/2013/hoffmann_13_sobres.pdf

[40] T. J. Ostrand and E. 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, ser. ISSTA '02. New York, NY, USA: ACM, 2002, pp. 55–64. [Online]. Available: http://doi.acm.org/10.1145/566172.566181

[41] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Where the bugs are," in *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '04. New York, NY, USA: ACM, 2004, pp. 86–96. [Online]. Available: http://doi.acm.org/10.1145/1007512.1007524

[42] D. Patterson and A. Ganapathi, "Crash data collection: A windows case study," *3D Digital Imaging and Modeling, International Conference on*, pp. 280–285, 2005.

[43] R. Matias, M. Prince, L. Borges, C. Sousa, and L. Henrique, "An empirical exploratory study on operating system reliability," in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, ser. SAC '14. New York, NY, USA: ACM, 2014, pp. 1523–1528. [Online]. Available: http://doi.acm.org/10.1145/2554850.2555021

[44] N. Palix, G. Thomas, S. Saha, C. Calvès, G. Muller, and J. Lawall, "Faults in linux 2.6," *ACM Trans. Comput. Syst.*, vol. 32, no. 2, pp. 4:1–4:40, Jun. 2014. [Online]. Available: http://doi.acm.org/10.1145/2619090

[45] A. Mahmoud, R. Venkatagiri, K. Ahmed, S. Misailovic, D. Marinov, C. W. Fletcher, and S. 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.

[46] A. Depoutovitch and M. Stumm, "Otherworld: Giving applications a chance to survive os kernel crashes," in *Proceedings of the 5th European Conference on Computer Systems*, ser. EuroSys '10. New York, NY, USA: ACM, 2010, pp. 181–194. [Online]. Available: http://doi.acm.org/10.1145/1755913.1755933

[47] C. Bolchini, M. Carminati, and A. Miele, "Self-adaptive fault tolerance in multi-/many-core systems," *J. Electron. Test.*, vol. 29, no. 2, pp. 159–175, Apr. 2013. [Online]. Available: http://dx.doi.org/10.1007/s10836-013-5367-y

[48] D. Kuvaiskii, R. Faqueh, P. Bhatotia, P. Felber, and C. Fetzer, "Haft: Hardware-assisted fault tolerance," in *11th European Conference on Computer Systems (EuroSys)*, London, UK, April 2016, pp. 1–17.

[49] S. Sundararaman, S. Subramanian, A. Rajimwale, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. M. Swift, "Membrane: Operating system support for restartable file systems," *Trans. Storage*, vol. 6, no. 3, pp. 11:1–11:30, Sep. 2010. [Online]. Available: http://doi.acm.org/10.1145/1837915.1837919

[50] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy, "Recovering device drivers," *ACM Trans. Comput. Syst.*, vol. 24, no. 4, pp. 333–360, Nov. 2006. [Online]. Available: http://doi.acm.org/10.1145/1189256.1189257

[51] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. 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*, ser. OSDI '06. Berkeley, CA, USA: USENIX Association, 2006, pp. 4–4. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267308.1267312

[52] K. Elphinstone and Y. Shen, "Increasing the trustworthiness of commodity hardware through software," in *43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2013.

[53] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum, "Construction of a highly dependable operating system," in *Proceedings of the Sixth European Dependable Computing Conference*, ser. EDCC '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 3–12. [Online]. Available: https://doi.org/10.1109/EDCC.2006.7

[54] R. Nikolaev and G. Back, "Virtuos: An operating system with kernel virtualization," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: ACM, 2013, pp. 116–132. [Online]. Available: http://doi.acm.org/10.1145/2517349.2522719

[55] F. M. David, E. M. Chan, J. C. Carlyle, and R. H. Campbell, "Curios: Improving reliability through operating system structure," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 59–72. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855746

[56] A. Lenharth, V. S. Adve, and S. 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*, ser. ASPLOS XIV. New York, NY, USA: ACM, 2009, pp. 49–60. [Online]. Available: http://doi.acm.org/10.1145/1508244.1508251

[57] K. Bhat, D. Vogt, E. v. d. Kouwe, B. Gras, L. Sambuc, A. S. Tanenbaum, H. Bos, and C. 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.

[58] K. Govil, D. Teodosiu, Y. Huang, and M. Rosenblum, "Cellular disco: Resource management using virtual clusters on shared-memory multiprocessors," in *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, ser. SOSP '99. New York, NY, USA: ACM, 1999, pp. 154–169. [Online]. Available: http://doi.acm.org/10.1145/319151.319162

[59] D. Gens, "Os-level attacks and defenses: From software to hardware-based exploits," Ph.D. dissertation, Technische Universitt Darmstadt, Dec. 2018.

[60] M. Engel and B. Dbel, "The reliable computing base: A paradigm for software-based reliability," in *Workshop on SOBRES*, 2012.

[61] A. Lackorzynski, A. Warg, M. Hohmuth, and H. Härtig, "L4re," https://l4re.org/doc/index.html, 2018.

[62] J. Szefer, E. Keller, R. B. Lee, and J. Rexford, "Eliminating the hypervisor attack surface for a more secure cloud," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS '11. New York, NY, USA: ACM, 2011, pp. 401–412. [Online]. Available: http://doi.acm.org/10.1145/2046707.2046754

[63] Y. C. Yeh, "Triple-triple redundant 777 primary flight computer," in *1996 IEEE Aerospace Applications Conference. Proceedings*, vol. 1. IEEE, 1998, pp. 293–307.

[64] P. Traverse, I. Lacaze, and J. Souyris, "Airbus fly-by-wire: A total approach to dependability," in *Building the Information Society*. Springer, 2004, pp. 191–212.

[65] L. Mancini, "Modular redundancy in a message passing system," *IEEE Transactions on Software Engineering*, no. 1, pp. 79–86, 1986.

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

[67] N. Schiper, V. Rahli, R. Van Renesse, M. Bickford, and R. 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.

[68] M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance," in *3rd Symposium on Operating Systems Design and Implementation*. New Orleans, USA: ACM, Feb. 1999.

[69] M. Correia, N. F. Neves, and P. Verissimo, "How to tolerate half less one byzantine nodes in practical distributed systems," in *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems, 2004.*, Oct 2004, pp. 174–183.

[70] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda, "TrInc: Small trusted hardware for large distributed systems." in *NSDI 2009*, vol. 9, Boston, Massachusetts, USA, 2009, pp. 1–14.

[71] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Verissimo, "Efficient byzantine fault-tolerance," *IEEE Transactions on Computers*, vol. 62, no. 1, pp. 16–30, Jan. 2013. [Online]. Available: http://dx.doi.org/10.1109/TC.2011.221

[72] T. C. Bressoud and F. B. Schneider, "Hypervisor-based fault tolerance," in *15th ACM Symposium on Operating Systems Principles (SOSP)*, Copper Mountain, Colorado, USA, 1995, pp. 1–11.

[73] E. G. Esposito, P. Coelho, and F. Pedone, "Kernel paxos," in *37th Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 2018.

[74] L. Lamport, "The part-time parliament," *Transactions on Computer Systems*, vol. 16, no. 2, pp. 133–169, 1998.

[75] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The multikernel: A new os architecture for scalable multicore systems," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 29–44. [Online]. Available: http://doi.acm.org/10.1145/1629575.1629579