Security and Safety, Vol. 2 , 2023007(2023)
https://doi.org/10.1051/sands/2023007
sands.edpsciences.org

Views OPEN ACCESS

Information Network

Multikernel: Operating System Solution to Generalized Functional Safety

YiJing Song[1,4], HuaSheng Dai[1], JinHu Jiang[1,2,3], and WeiHua Zhang[1,2,3] *

[1] *School of Computer Science, Fudan University*
[2] *Institute of Big Data, Fudan University*
[3] *Parallel Processing Institute, Fudan University*
[4] *State Key Laboratory of Mathematical Engineering and Advanced Computing*

Received: xx xxxxx 2023 / Revised: xx xxxxx 2023 / Accepted: xx xxxxx 2023 / Published online: xx xxxxx 2023

Abstract With the trend of digitalization, intelligence and networking sweeping the world, functional safety and cyber security are increasingly intertwined and overlapped, evolving into the issue of generalized functional safety. Traditional system reliability technology and network defense technology cannot provide quantifiable design implementation theory and method. As the cornerstone of software systems, operating systems in particular are in need of efficient safety assurance. The DHR architecture is a mature and comprehensive solution, and it is necessary to implement an OS-level DHR architecture, for which the multikernel operating system is a good carrier. The multikernel operating system takes the kernel as the processing scenario element and constructs redundancy, heterogeneity, and dynamism on the kernel, so it has the generalized robustness of the DHR architecture. This article analyzes the significance and requirements of OS-level DHR architecture, and systematically explains how the multikernel operating system respond to the requirements of OS-level DHR architecture by analyzing the technical routes of multikernel operating systems, and develops an operating system solution idea for the generalized functionally safety.

Keywords Multikernel operating system, Generalized functional safety (S&S), Dynamic heterogeneity redundancy (DHR) architecture

Citation YiJing Song, HuaSheng Dai, JinHu Jiang and WeiHua Zhang. Multikernel: Operating System Solution to Generalized Functional Safety. Security and Safety 2023; **2**: 2023007. https://doi.org/10.1051/sands/2023007

1 Introduction

Current systems suffer heavy troubles of safety and security problems. Traditional safety theories concentrate on system reliability, which expects correct outcomes when accidental faults or systematic failures occur on physical devices or systems. As dependence to digitization and interconnectedness surges at an explosive speedup, cyber breaches propagate correspondingly. The system vulnerabilities as well as backdoors could not be circumvented in design or eliminated during runtime, which amplify the possibility of cyber attacks in force. The inevitable design defects bring plenty opportunities to attackers and thus be enlarged beyond mathematical properties. Previous solutions consider functional safety and cyber security as orthogonal issues, hence apply standalone approaches to figure out them. However, functional safety and cyber security issues are increasingly intertwined, overlapped and inseparable. Functional safety has broken through the randomization assumption of traditional reliability theory and become a universal safety and security problem, also known as generalized functional safety problem(S&S) [1]. Therefore, there is an urgent need to propose design theories and methods as well as test and evaluation systems for generalized functional safety. The DHR architecture [1] is a mature and comprehensive solution that

* Corresponding author (email: zhangweihua@fudan.edu.cn)

can handle random failures or deliberate attacks inside the structure effectively and ensure quantifiable design and verification of S&S features. Through redundancy, heterogeneity, and dynamism, the DHR architecture makes the attack surfaces of different processing scenario elements orthogonal, and transforms deliberate attacks against hardware and software vulnerabilities or backdoors into quantifiable problems that can be measured probabilistically with the idea of redundant execution.

As the cornerstone of system software, operating system integrates both service and management functionalities, and hence has the obligation to respond to the generalized functional safety requirements at the software level. This article first analyzes the significance of constructing the DHR architecture at the operating system level and the requirements for the operating system, based on which the multikernel operating system is proposed as a carrier for the DHR architecture at the operating system level. Afterwards, the natural advantages of the multikernel operating system as a carrier are systematically explained, and the technical routes of the multikernel operating system are enumerated from the requirements of the DHR architecture for the operating system, including the basic technologies and the technologies related to the DHR architecture, so that the feasibility of constructing the OS-level DHR architecture is demonstrated. As a pioneer in implementing a stand-alone DHR architecture with generalized robustness, the multikernel operating system can effectively address the generalized functional safety problem encountered in the field of operating systems today, and illuminate heuristic ideas and key challenges for the successors.

2 Background

Safety and security problems are becoming the crux of digital systems. Distinguished from the functional availability as well as performances, the bottleneck of systems switches to reliability and trustworthiness gradually. As the computing acts a tendency of expansion and collaboration, the fault-tolerance and reliability grow much more critical in the system design, towards both mechanical and digital failures. Besides, multitudinous computing tasks are applied in various complicated scenes such as autonomous vehicles, which raises more rigorous requirements to maintain functional safety. The international standard ISO26262 named "Road vehicles – Functional safety" was published in 2011 [2], aiming to restrict the functional safety risks of electrical and electronic systems installed in road vehicles. Moreover, the digitization of mechanisms introduces increasing digital risks. A premium-class automobile may contain approximately 100 million lines of software code directing the function of 70 to 100 microprocessor-based electronic control units [3]. Enormous projects increase the complexity of engineering, and the vulnerabilities burgeon to the cyber area rapidly as the trends of connected networking. Back to autonomous vehicles, a research listed a terrific amount of cyber vulnerabilities of autonomous vehicles including attacks to low-level sensors and vehicle control modules [4]. Cyber security has become a significant topic.

Under the prospect of deep integration of human, machine and material, the functional safety of mechanical system has long surpassed the traditional production efficiency, and its influence gradually penetrates into every corner of life. With the acceleration of the fourth technological revolution and industrial change, the intersection of emerging information technology dissolves the traditional boundary between the physical world and the digital world, human cognitive activities become more and more dependent on digital media, and the subjective activities of cognition and reflection derive an online virtual third pole that bridges the material world and the cognitive world in the gap of Descartes' duality. The invisible cognitive domain breaks free from intracranial bondage and externalizes into a tangible online cognitive space, while emerging technologies such as IoT, big data, AI, 5G/6G, blockchain, etc. are giving the cultivating ground to the virtual cognitive domain based on information. And the chronic problems of functional safety and cyber security come into play. The externalized virtual cognitive domain is still built by the material hardware systems, and the danger of attacks against the underlying drive systems will be magnified by its specificity. Functional downtime of the cognitive domain will partially delay and interrupt human cognitive activities; while cyber attacks may maliciously implant distorted cognitive information, thus creating a more confined information cocoon. The instability of the cognitive domain will profoundly change the way people engage in their subjectivity in the world, and therefore requires a new functional safety paradigm to guard it. Beyond the impact on individuals and societies, cognitive safety is becoming increasingly important for national security as the world becomes more anti-globalized and the divisions

as well as conflicts between countries intensify. During the Russia-Ukraine conflict, Viasat, a U.S. satellite operator covering Ukraine, suffered a cyberattack that knocked out thousands of Ukrainian users and tens of thousands of other European users [5]. Color revolutions, fomented by cognitive barriers and falsehoods, are also a common occurrence. The thriving cognitive domain also brings us new forms of thinking about availability and reliability.

It follows that functional safety and cyber security have intertwined and mingled on an unprecedented degree, which is called generalized functional safety problem or S&S problem. Accordingly, "If there is a robust structure model that not only keeps a given mode's functions within the safety margin of quantifiable designs under the perturbation of some conventional randomness factors, but also safeguards the reliability of the model's functions against the activated cyberattacks based on in-house software/hardware vulnerabilities or backdoors, etc., the very model shall be deemed as generalized functional safety (GFS)." As the technological level of intelligent networking systems expands, there remain pressing concerns about their S&S problem.

3 Challenges of operating system

Operating system plays the role of cornerstone of the software architecture. Its obligation comprises managing the hardware resources and constructing the execution environment for user applications. Serving as a connecting intermediary between the applications upwards and the hardware downwards, operating system directs and regulates executions of the whole mechanism. Operating system is consequently responsible to respond the requirements of generalized functional safety in software layer, since a subtle failure of it may lead to collapse of the entire computing architecture.

Remarkably, current operating systems are lagging the urgent S&S demands. The deficiency could be summarized towards three aspects.

3.1 Fragility

The protection of existing operating systems is weak and flimsy. Tiny error exposure that causes failure of key functional components can lead to system-wide crashes, greatly weakening the system's resilience to disturbances. Constructing redundancy is the common practice to maintain functional stability. Redundant components can perform the same tasks to hide potential random runtime errors and share the workload of other execution units, otherwise keep spare in case of functional emergency. Previous operating systems rely on multiple hosts to provide redundant services. For example, in distributed systems, where multiple hosts provide homogeneous services, user applications are free to migrate between multiple nodes, and the crash or exit of a particular node is transparent to the user application, thus giving the system a high threshold of error tolerance. However, the modern operating systems lack single-machine level redundancy. Almost all system components are located on the critical path of the system. The breakdown of any core module will spread to the whole system. Once single-source attacks break through, it can easily destroy all application execution and hardware availability on the host.

3.2 Monotony

The properties of existing operating systems are monotonous and have distinctive individuality, which gives rise to attacks that target specific operating system features. According to analysis by cybersecurity researchers at Trend Micro, Linux servers are "increasingly coming under fire" from ransomware attacks, with detections up by 75% over the course of the last year as cyber criminals look to expand their attacks beyond Windows operating systems. [6] The current operating systems lack the rich attributes of differentiated features, making system-specific differential issues prominent, thus system security maintenance efforts that cannot eradicate inherent design flaws in the operating system. The discovery of implicit vulnerabilities relies on the existing documentation of implemented attacks, making it difficult to share common solutions among many mature operating systems. And the maintenance teams of system vendors work in isolation and have to be tired of passively coping with the endless individual problems.

3.3 Rigidity

Existing operating systems lack temporal changes, which is highly static in timing, thus providing a transparent and consistent view for cyber attacks. On the one hand, many cyber attacks depend on the runtime snooping of critical data of the system. For example, kernel leakage attacks against Linux need to first probe the kernel's randomized base address pointers under the protection of KASLR. The temporal changes in the system will make the pre-preparation of these attacks more laborious or even undone, which significantly raises the cost of cyber attacks. On the other hand, although the current functional safety and network defenses are tight with strict design specifications, the mechanical nature of the system makes it possible to gain complete control of the attack target once an escape occurs, making the existing defenses obsolete and reduced to a "Maginot Line" in the system security attack and defense. That is, system mechanics makes security defenses unsustainable after an escape occurs, and lacks a fallback against defense failure.

3.4 Summary

The fragility, monotonicity, and rigidity of current operating systems make the safety support of operating systems lag behind the generalized functional safety requirements. Therefore, there is an urgent need to empower operating system design theories, implementation methods, and evaluation systems with new security thinking paradigms to address the widespread generalized functional safety issues.

4 DHR architecture & OS supports

One of the well-established theoretical solutions to the generalized functional safety problem is dynamic heterogeneity redundancy (DHR). DHR refers to a heterogeneous and redundant set of processing scenarios that provides several processing scenario elements as the environment for executing the input sequence, while iteratively determining the response of the execution set based on a given policy, dynamically changing the verdict policy or a subset of processing scenario elements as the execution environment until their response satisfies the verdict policy, at which point the DHR reaches a steady state and generates an output sequence. The DHR is mathematically proven to be effective and can therefore be applied to the operating system domain to build an OS-level DHR to respond to generalized functional safety requirements at the software-level.

Computer systems include user applications, hardware resources, and operating systems. It is imperative to build a DHR architecture at the OS-level. On the one hand, the operating system is responsible for abstracting hardware resources and providing services for applications, and it is responsible for communicating with upper-level user applications and lower-level hardware resources, hiding the complicated details of hardware and providing easy-to-use standardized interfaces for user programs. It is the common foundation for application execution and therefore has the obligation to build a stable and safe execution environment. On the other hand, the operating system also takes responsibility for the management of both hardware resources and applications. It is responsible for the unified management of heterogeneous and complex hardware resources as well as the management of the application life cycle and resource allocation, and is the sole coordinator and manager of the computer system, so it has the obligation to maintain a stable and secure system state. Therefore, the function of the operating system requires that the DHR architecture be built on top of it to respond to the generalized functional safety requirements.

For the other participant of the computer system, it is difficult to build a DHR architecture on it with the current state of art. The difficulties with hardware-level DHRs are complexity and flexibility. For one thing, implementing the dynamics of the DHR architecture at the hardware-level introduces unnecessary complexity. The dynamic nature of the DHR requires the implementation of an iteration-based voting verdict module. The verdict policy is required to rule on the runtime system state, application execution results, external inputs, etc., and may contain complex program algorithmic logic. It is more difficult to implement the complexity embedded in the verdict policy on the circuit connected with electronic components, and it is wiser to delegate the algorithmic part to software programming. Second, the heterogeneity of implementing the DHR architecture at the hardware-level loses some of flexibility. The heterogeneous nature of the hardware execution unit is difficult to change once it is defined. Adding or

removing heterogeneous processing scenario elements or modifying the heterogeneous nature of processing scenario elements at runtime would require modifying the underlying hardware wiring logic to access or remove different hardware functional units, which is impractical to implement in engineering.

The drawbacks of user application-level DHR are vulnerability and reusability. For one thing, user applications lack system privilege level protection and thereby are more vulnerable to attacks than operating systems. On systems dedicated to serving multiple applications, OS needs to provide services while being protected from unintentional errors or malicious attacks by user applications because of their unknown origin and doubtful trustworthiness. Hence the operating system places itself at a high privilege level to provide isolation from applications. The application-level DHR architecture is more vulnerable to malicious cyber attacks due to its heterogeneous and redundant processing scenario elements that are exposed to the privileged level of protection. This vulnerability is independent of the heterogeneous nature of the processing scenario elements themselves. The other is the diversity of the applications making the application-level DHR architecture much less reusable than the underlying implementation of the architecture. Since the characteristics vary among applications, it is difficult to abstract a common construction method for heterogeneous redundancy scenario processing elements around the specialized application characteristics. It is necessary to conceive different heterogeneous design solutions for different applications, thus transferring the design work of the DHR architecture to all application providers and aggravating the development pressure of the applications. Besides, since each application task requires multiple heterogeneous variants, the number of application instances grows linearly, and the system needs to maintain the execution status of different heterogeneous instances of all applications at runtime, which brings heavy runtime overhead to the system. In contrast, the OS-level processing scenario elements can serve multiple applications after a single development due to their servicer role, and can be shared by different applications at runtime, thus showing excellent reusability over the application-level DHR architecture in both static and dynamic time. In view of above, no other participant in the computer system can be a good carrier of DHR architecture, and the operating system has to take the responsibility of building the DHR architecture.

The DHR architecture puts the following requirements on the operating system.

4.1 Redundancy

Redundancy requires the operating system to provide multiple execution environments for the application transparently.

First, it is essential for the operating system to build multiple execution environments based on a single image of the application and to run multiple copies of instances of the same application in multiple execution environments. Unlike the traditional concept of multi-threaded processing, different instances of an execution receive the same sequence of inputs, perform the same redundant work with consistent program logic, and are expected to give consistent response results.

Second, it is important for the operating system to provide isolation between multiple execution environments. Multiple instances of a program execute simultaneously in redundant execution environments, but they should do the same work independently and should not communicate or collaborate with each other. Therefore, the operating system needs to ensure strong isolation between execution environments, which shall prevent errors or attacks on one instance from spreading to other execution environments, and shall prevent multiple instances of the program from using communication to collaborate on fraud.

Finally, it is expected for the operating system to achieve redundancy that is transparent to the application. Applications written for a single execution environment do not have the necessity and legality to be aware of the existence and details of the redundant execution environment actually provided by the operating system. The implementation strategy for redundancy should be entirely determined by the operating system. The application can maintain the original hypothetical execution environment abstraction, i.e., the operating system should provide compatibility with existing common applications.

4.2 Heterogeneity

Heterogeneity requires the operating system to provide functionally equivalent heterogeneous execution environments.

First, it is necessary for the operating system to construct and provide multiple execution environments with different structures. The operating system is required to design the heterogeneity of execution environments, design the architecture to support the coexistence of multiple heterogeneous execution environments, and generate several instances of heterogeneous environments at runtime. The depth of heterogeneity of execution environments is guaranteed by the operating system. If there are commonality problems among multiple environments, the system attack surface that leads to escapes will be exposed.

Second, there is a need for the operating system to maintain the functional equivalence of multiple execution environments. Heterogeneous execution environments are required to provide functionally equivalent system services and semantically consistent environment information to ensure that applications get equivalent expected output in heterogeneous environments. At the same time, the operating system needs to support the compatibility of heterogeneous environments with a single application copy by transparently assigning different instances of the application to execute in the heterogeneous environment at runtime.

Finally, it is vital for the operating system to design a consensus mechanism among heterogeneous environments. The same application in a heterogeneous environment may produce inconsistent output, and the system needs to identify the root cause of response discrepancies, which may arise from external attacks, heterogeneous features, or non-conformance of redundant execution environments, to be specifically adjudicated by the operating system. The operating system needs to monitor and collect different key information about the heterogeneous environment, and based on this runtime information and verdict policy, obtain inter-heterogeneous consensus by voting. The type of monitoring information and the verdict algorithm are the core of the consensus mechanism.

4.3 Dynamism

Dynamism requires the operating system to dynamically iterate over the set of execution environments.

First, it is required to support dynamic iteration of the execution environment. The operating system needs an architecture that supports dynamic changes in heterogeneous redundant collections, and a clear interface designed for easy assembly and parallel operation between any heterogeneous elements. The execution environment needs to support runtime distribution and migration of applications to ensure the speed and accuracy of iteration. If the implementation allows, the operating system can also support hot-plugging of the execution environment to reduce runtime overhead and power consumption.

Second, there is a practical need to design a dynamic iteration strategy for the operating system. The operating system initiates the next iteration or terminates it and generates output based on the consensus mechanism of the heterogeneous environment. Iteration control depends on several factors, including ruling parameters, external inputs, etc., in addition to the runtime response-based consensus mechanism of the heterogeneous environment to form the final verdict. Iterative verdict includes replacing the processing set of the execution environment, changing the heterogeneous nature of the execution environment, refreshing the environment and re-executing it, changing the iteration policy, etc., which is dependent on the operating system implementation.

4.4 Others

OS-level DHRs also need to remain cost-effective.

First, to the extent possible, the operating system should cut the runtime overhead of the DHR architecture. The system monitoring required for consensus mechanisms in heterogeneous environments should avoid excessive granularity, which may result in environmental noise during application execution. The voting process required to reach consensus in heterogeneous environments should also avoid excessive synchronization overheads that cause system delays at the tail end of application execution. Dynamic iterations of heterogeneous environment collections should be as efficient as possible to avoid significant stalls between iterations (stall).

Second, the OS-level DHR is expected to have acceptable development and maintenance costs. The design and generation of heterogeneous execution environments should be based on generic methods as much as possible to avoid duplication of development costs. Good scalable interfaces should be available for the operating system to facilitate the access and exit of heterogeneous classes.

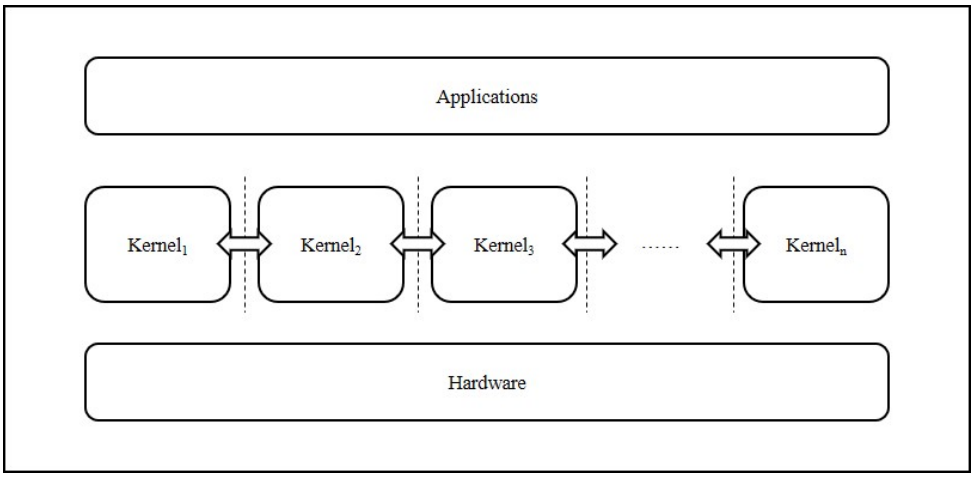


Figure 1. Multikernel operating system architecture

5 Design: multikernel architecture

We observe that the multikernel operating system is a good carrier for the OS-level DHR architecture based on its requirements.

In the multikernel operating system, multiple kernels are running simultaneously, each of which has either homogeneous or heterogeneous properties. The kernels collaborate with each other through efficient communication mechanisms. Multikernel operating systems use kernels as processing scenario elements and provide dissimilar execution environments by building dynamically heterogeneous and redundant kernels as an OS-level DHR implementation. The architecture of a multi-core operating system is shown in Figure 1.

The selection of the kernel as the processing scenario element of the OS-level DHR architecture offers natural advantages.

In regard to adequacy, the kernel is the core of the operating system, providing services to applications and arbitrating their behavior. Key system services of the operating system are implemented in the kernel or are monitored and controlled by the kernel at runtime. The functionality of the kernel therefore calls for it to assume responsibility for the OS-level DHR architecture.

In terms of necessity, there are more benefits to be gained by building dynamically heterogeneous and redundant kernels.

In the first place, the kernel takes on the primary functionality of the operating system and remains at the highest privilege level, making it even more dangerous for attacks to escape. External attacks often acquire high privileges by hacking the kernel to execute malicious program fragments, so a small kernel vulnerability or backdoor can cause an avalanche of hazards. Building an OS-level DHR with kernel as the processing scenario element can effectively protect high privilege levels from theft by hiding the attack surface of the kernel.

In the next place, using the kernel as the processing scenario element can deepen heterogeneity. Traditional DHR-like architectures build heterogeneity through randomness, dynamics, and diversity, etc., such as address space layout randomization (ASLR) and modification of compilation options, which can be compatible with current operating systems but also lose the underlying heterogeneity and are vulnerable to breakthroughs through short-circuiting and probing techniques. However, kernel-level heterogeneity can overturn the original heterogeneity generation logic and make heterogeneity reach the bottom level of the execution environment, where the probability of common mode problems depends merely on the philosophy and thinking of the kernel designer, so that the attack surface is maximally orthogonal.

Furthermore, having kernels as processing scenario elements can reduce development costs. Heterogeneous kernels can draw on existing mature commercial kernels without having to explore common generation algorithms for heterogeneous elements. At the same time, the logic of using the entire kernel as a heterogeneous element is more natural, eliminating the effort to decouple modules of the system and bridge between heterogeneous elements. Therefore, multikernel operating systems are available as implementations of the OS-level DHR architecture.

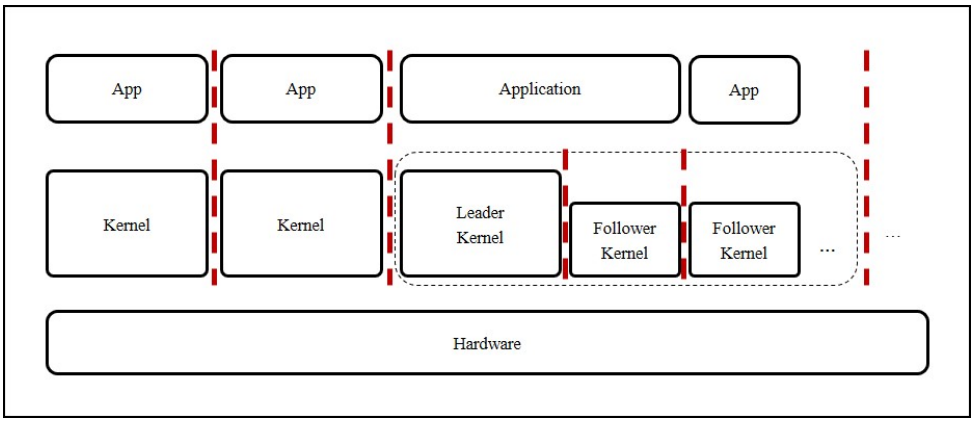


Figure 2. Multikernel architecture

The following is an overview of the technical route to the multikernel operating system.

5.1 Underlying technologies

The underlying technologies include multikernel architecture, hardware resources partitioning and inter-kernel communication.

5.1.1 Multikernel architecture

Multikernel architectures are designed to explore how multiple kernels can work together in a single operating system. Flexible multikernel symbiotic relationships can be constructed, for instance peer, leader/follower, and so on. During OS startup, the first kernel to be booted acts as the manager kernel to allocate resources for all kernels and wake up the others. For the runtime, each kernel has exactly the same status in the peer relationship, i.e. the manager kernel has the same privileges as the kernel it wakes up; while in the leader/follower architecture, the behavior and privileges of the follower kernels are restricted and the leader kernel dominates the global application scheduling and resource allocation.

From the application perspective, each kernel supports several applications independently. In the peer relationship, any application can run on arbitrary kernel. In the leader/follower relationship, the leader kernel is responsible for application assignment and lifecycle management, while the follower kernels are only responsible for application execution and monitoring.

From the hardware perspective, hardware resources are partitioned and clustered, which are managed by different kernels. In the peer architecture, processor cores and memory are partitioned into different kernels, while non-partitionable external devices can be managed by a particular kernel and invoked by the other peer kernels through message communication. In the leader/follower architecture, external devices can be managed and dispatched by the leader kernel, while other kernels invoke hardware services provided by the leader kernel through message communication.

The core data structures supporting multikernel architecture are explored based on inter-kernel relationship design to reproduce the service and management functionalities of the operating system, including multikernel booting, memory management, and hardware resource scheduling. The multikernel architecture is finally presented in a hybrid way as shown in Figure 2

5.1.2 Hardware resources partitioning

Hardware resources partitioning. With respect to generalized functional safety, resource partitioning in multikernel operating system entails a balance between security and resilience. Hardware resources in a computer system can be divided into three categories: cores, memories, and other devices. To avoid data leakage due to resource sharing, core and memory resources need to be partitioned first to ensure that there are no identical cores or overlapping memory areas between resource partitions, and then allocated

to each kernel separately. Other devices are scarce and more diverse than cores and memories, and need to be designed according to the symbiotic relationship between multiple kernels. In addition, hardware resource allocation can be adjusted at runtime. Since there are no manager kernels in a peer architecture, dynamic resources allocation as well as recycling is only supported for multikernel operating systems with a leader/follower architecture.

5.1.3 Inter-kernel communication

Inter-kernel communication aims to investigate how to implement a secure and efficient communication mechanism on a single host. Traditionally, inter-process communication or remote procedure calls require a high level of privilege to ensure security and isolation, thus the kernel implements and monitors the communication process. However, in the multikernel architecture, the kernel turns from the intermediary to the subject of the communication process and runs in privileged mode, so a new type of communication with high privileges is desired. The existing means of remote procedure calls, including network communication, do not apply to single-host communication. Instead, efficient on-chip and inter-chip communication methods such as memory-based communication should be explored. Therefore, inter-kernel communication requires both the efficiency of single-host communication and the privacy of inter-kernel communication, which is a balance between sharing and isolation.

5.2 DHR architecture technologies

The DHR architecture technologies will be expanded from three properties respectively: redundancy, heterogeneity, and dynamism.

5.2.1 Redundancy

Redundancy mechanisms include multi-channel execution and inter-kernel isolation.

In the first place, applications are assigned to different kernels for multi-channel execution. The application is handed off to multiple kernels by the dispatcher. All kernels in the execution set take over the execution task and provide a semantically consistent execution environment, while executing a replica of the same application and providing system services to it. External input, environment variables, system configuration, and other information is also replicated and dispatched to multiple kernels simultaneously. The kernels are executed separately and independently, monitoring and checking their respective user programs. At the end, multiple kernels should produce consistent output as expected.

It is also desirable that multiple kernels be isolated from each other. Traditionally, applications run at the low privilege and are isolated from one another as well as from the kernel by the kernel. The kernel has exclusive access to the high privilege, and dominates all system permissions. The multikernel architecture requires inter-kernel isolation while maintaining the high privilege of the kernel over the applications. Inter-kernel isolation is critical to the safety of the multikernel architecture. In order to provide the redundancy required for broad functional security, random errors in one kernel and malicious attacks on one kernel should not penetrate into other kernels. A crash of one kernel should not cause a system-wide disruption. This is to say that the multikernel operating system should be fault-tolerant at the kernel-level. And since the kernel as an element of the processing scenario may have unknown vulnerabilities or backdoors, it may be transformed into a malicious kernel. Therefore the privileged capability of the kernel should be limited to prevent malicious kernels from deliberately snooping or attacking other kernels, thus affecting the stability of the full operating system. In other words, the multikernel operating system should have kernel-level protection capability.

5.2.2 Heterogeneity

Heterogeneity involves heterogeneous kernel architecture, heterogeneous execution mechanism, and kernel consensus mechanism.

First, the multikernel operating system should accommodate heterogeneous kernels at the same time. Different core data structures and intrinsic algorithmic logic are required for heterogeneous kernels. It

is therefore necessary to build abstraction layers that support the coexistence of heterogeneous kernels, including global data structures, heterogeneous boot mechanism, and heterogeneous resource abstractions. Rather than simply replicating the system-wide data structure across multiple kernels, custom data structures should be designed to include management information for the multikernel architectures and adapt to the characteristics of heterogeneous kernels, and heterogeneous kernels should be modified to support the core data structure of the system. Heterogeneous kernels also have different booting steps, device management models, and more, all of which need to be designed at a global level based on the multikernel architecture, extracting abstract commonalities and preserving concrete individuality.

The second is that multikernel operating systems are required to support heterogeneous execution of applications. Heterogeneous kernels may have different application execution frameworks, but applications need to be assigned to multiple heterogeneous kernels for execution, so it is necessary to design interfaces for heterogeneous kernels that are compatible with a single program file format. In addition, applications may have special requirements for the features or services of the execution kernel, leading to the need to implement an aware dispatcher that selects a subset of execution kernels based on the application's features and requirements.

In addition, there is a need for the multikernel operating system to implement a consensus mechanism for heterogeneous kernels. Due to the heterogeneous operating logic of heterogeneous kernels and the different characteristics of the executing applications, they may show different execution results, such as different types and numbers of system calls, different inter-process communication counts, and so forth. This requires implementing a heterogeneous monitoring component to collect runtime information specific to the heterogeneous kernels and to synthesize the responses of all kernels to obtain consensus through voting. According to the requirements of generalized functional safety, the multikernel architecture demands a Byzantine-like consensus algorithm that allows fraud, with predefined verdict algorithms and policies to reach a consensus on whether a random error or a deliberate attack has occurred. Typical Byzantine algorithms such as PBFT [7] can provide ideas for multikernel heterogeneous consensus mechanisms. This should be achieved through efficient inter-core communication.

5.2.3 Dynamicity

Dynamicity mechanisms involve kernel set dynamic changes and dynamic iteration strategies.

First, the multikernel operating system should support dynamic changes in the subset of kernels which are elements of the processing scenario. Transparent migration of applications while running on heterogeneous kernels needs to be supported by the multikernel operating system to enable seamless migration to the new set of executing kernels between application iterations. What needs to be migrated with the application includes application control information and data structures, runtime state, and the hardware resources at its disposal. In addition, deciding on the full set of kernels as processing scenario elements at boot time increases the work overhead at boot time and prevents the addition or removal of kernels at run time, compromising boot performance and operational flexibility. For this reason, kernel hot-plugging mechanisms need to be designed. Hence, the multikernel operating system should support dynamic kernel loading, dynamic unloading, dynamic scheduling, dynamic migration, and dynamic resource scheduling for multikernel architectures, so that the kernel and related system services can be packaged as an image and idle on the secondary storage, selected and loaded by the kernel scheduler when needed, allocated several hardware resources and assigned to executing applications, and reloaded to the secondary storage when not needed. In addition, kernels also should be able to change their characteristics dynamically at runtime to present uncertainty in processing scenario elements, as a result of which, the system functionality of the kernel should be loosely coupled and modularized.Multi-attribute module components should be generated, and on-demand elastic combination and customization of multi-attribute modules should be supported so that kernels and system services with different characteristics can be generated quickly and thus be more dynamic and diverse in iteration.

Besides, the multikernel operating system should also realize an iteration policy to control the dynamism. On the basis of the heterogeneous consensus mechanism, the system receives a safety and security verdict of the application execution process and proceeds to the next iteration if the output does not satisfy the predefined policy. The iterative policy allows refreshing the runtime data and state of a kernel, dynamically changing the characteristics of certain kernels, dynamically adding or removing subsets of execution kernels, or forcing the application to execute again on only a subset of the original

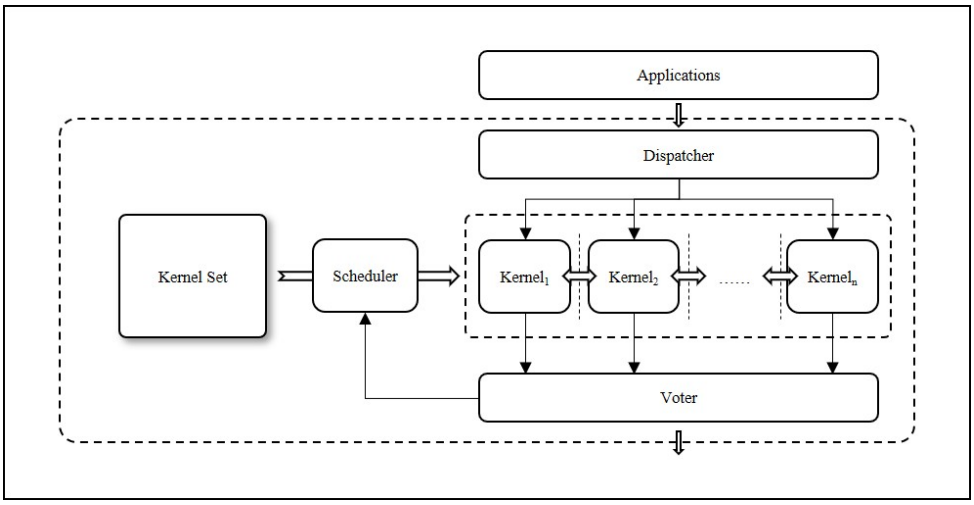


Figure 3. Multikernel operating system corresponding to OS-level DHR architecture

kernels until the heterogeneous kernel consensus reaches steady state. The dynamic iteration policy is a manifestation of dynamism and a control pivot for the multikernel operating system to achieve an OS-level DHR architecture. Therefore, an efficiently convergent iteration policy needs to be designed in conjunction with a fast and accurate heterogeneous consensus algorithm so that the system can ensure generalized functional safety with minimal mechanism overhead.

5.3 DHR architecture by multikernel OS

As a result, the design of the multikernel operating system to implement the OS-level DHR architecture is shown in Figure 3. As processing scenario elements, all kernels are pooled into a collection of processing scenarios. A subset of kernels is selected by the scheduler for executing an application. The application is sent by the dispatcher to the heterogeneous kernels selected by the scheduler for execution. The heterogeneous kernels continuously reach consensus by the voting machine during execution and choose the execution method for the next iteration based on the given iteration policy until the response result satisfies the presumption, at which point the system reaches steady state and outputs the execution result.

5.4 Discussion

As shown above, the multikernel operating system can fulfill the redundancy, heterogeneity, and dynamism requirements of the DHR architecture respectively, and competently serve as a good carrier for the OS-level DHR architecture. It has therefore a generalized robustness to handle random failures within the architecture or cyber attacks against endogenous security issues, which makes it possible to respond to the need for generalized functional safety. The multikernel operating system does not require the entire set of its kernels to be trusted, avoiding the assumption of a trusted computing base and its self-evidence, and hence goes beyond the "soul problem" of traditional trusted computing to quantify the system's generalized functional safety problem with a global probabilistic model. Specifically, the multikernel operating system solves many of the nightmares that plague current operating systems: it provides a stable application execution environment with a high threshold of error tolerance, which is extremely useful for error-sensitive programs; it hides kernel-specific attacks and minimizes the operating system differential problem through attack surface orthogonality; it can shield trial-and-error type attacks on the kernel and prevent the operating system defense from being brute-force cracked or short-circuited; it can ensure the temporal safety of the system, so that attacks against the kernel lose temporal persistence as a result of which the attack escape cannot reach the steady state. Multikernel operating system can comfortably cope with emerging high-safety complex scenarios, including both the information technology industry

that affects the material life of society (e.g., autonomous driving scenarios, etc.) and the cognitive domain scenarios that affect the activities of human subjects, and can ensure that mechanical systems provide stable material processing and actions and cognitive replication and communication, thus providing stable generalized functional safety for the information technology industry both online and offline. Therefore, the multikernel operating system is a good operating system solution for generalized functional safety at present, proposing a new operating system design theory and quantitative testing scheme.

Debugging multikernel operating system requires additional attention to communication and collaboration between multiple kernels compared to single kernel operating systems. Existing debugging tools can only support single kernel debugging, and cross-kernel debugging currently requires federating multiple debuggers. The complete toolchain support for multikernel operating system overturns the previous developing and debugging model for kernels and is yet to be supported by efforts.

Currently, the operating system team at the Institute of Parallel Processing, Fudan University is trying to develop a prototype multi-core operating system for generalized functional safety issues, and quantitatively analyze and evaluate its features. The prototype will be compiled and improved, and the corresponding open source project will be built.

6 Related work

Moving Target Defense [8]. Moving Target Defense provides attackers with a random and changing view of the underlying system through random, diverse, and dynamic techniques to amplify the network attack overhead. However, these features do not change the logical nature of the vulnerability or backdoor, making it possible for the attack to short-circuit or co-opt the defenses. The multi-core operating system, as a carrier of the OS-level DHR architecture, introduces the concept of redundant execution, so that the differential problem is limited to a particular kernel and does not penetrate system-wide, as a result of which the potential hazard of unknown vulnerabilities or backdoors is hidden.

Distributed systems. Distributed systems build crash-consistent fault tolerance through multi-host heterogeneous redundancy. The multikernel operating system draws on the experience of distributed systems and attempts to apply the design concepts of distributed operating systems to a single machine in order to implement the OS-level DHR architecture on a single machine and cope with the endogenous security problems on a single machine. The network- and time-out-based algorithms of distributed systems and the multi-host resource management and application assignment are no longer adapted to the needs of single-computer DHR architectures, and require a specific adaptation based on the multikernel operating system.

Multi-Variant Execution. Multi-variant execution is the product of combining the idea of moving target defense with redundant execution by running a set of multiple variants of the same software with equivalent functionality and different structures [9]. First, multi-variants only target a single application change, which is an application-level defense, while the multi-core operating system constructs the DHR architecture by dynamically heterogeneous redundancy transformation of the kernel, which implements the modification of the entire execution environment and makes the defense more integrated. Österlund S proposed the multi-variant architecture kMVX [10] for kernels in 2019, providing a solution for kernel-level heterogeneous redundancy. Second, multi-variant execution achieves defense goals through heterogeneous redundancy, and dynamism is rarely reflected; the multikernel operating system iteratively executes by dynamically changing the set of kernels, thus giving temporal continuity to the generalized functional safety defense. Multikernel operating systems are more comprehensive than multi-variant executions, and are certainly more difficult to implement.

Lightweight VM. Lightweight VM solutions can implement DHR-like architecture by running a heterogeneous kernel within each VM. However its protection boundary is different from that of a multikernel OS solution. Architecturally, each kernel in a multikernel OS runs on its own physical resources, which provides more complete isolation while ensuring that physical resources are not directly shared; whereas multiple lightweight VMs still need to share the VM hypervisor resulting in weak isolation, and the impact of a particular VM may still penetrate to other VMs through the shared layer. However, due to the mature virtualization support of current hardware, multikernel solutions can still derive benefits from the hardware isolation of virtualization technologies, such as using EPT to isolate memory. We also expect hardware support for multikernel architectures to be introduced in the future.

Other multikernel operating systems. Barrelfish [11], an experimental operating system led by ETH Zurich, first proposed the multikernel operating system model with one operating system kernel per processor core to improve scalability. FOS [12] reduces resource contention by decomposing the full-featured operating system into individual services running on a single processor core. HeliOS [13] treats programmable devices as cores of equal status to the processor, providing a unified abstraction of heterogeneous devices. Popcorn Linux [14] replicates the Linux kernel to run on multiple cores, again addressing the issue of multicore scalability of operating systems. Although these multikernel operating systems do not involve consideration of S&S issues, their architectural designs and system implementations are still worthy of consideration and provide ideas and lessons for solving the problem of broad functional security of systems.

7 Conclusion

Attributed to the generalized functional security problems prevailing in the current cyberspace, there is an urgent need to propose new quantifiable and reliable system design theories and implementation methods. The DHR architecture is a mature and comprehensive solution with generalized robustness. Operating systems, as the cornerstone of system software, are obliged to respond to the generalized functional safety requirements. Multikernel operating systems as a good carrier for OS-level DHR architectures, can effectively implement the design theory of DHR architecture by using the kernel as a processing scenario element and creating redundancy, heterogeneity, and dynamism on it. This article systematically describes the solution and design ideas of the multikernel operating system for the generalized functional safety problem, demonstrates the rationality of the solution, composes the critical technologies and core issues, proposing a clear technical route for the implementation of the OS-level DHR architecture.

Conflict of Interest

The author declares no conflict of interest.

Data Availability

No data are associated with this article.

Authors' Contributions

Yijing Song, Huasheng Dai, Jinhu Jiang and Weihua Zhang jointly design the multikernel architecture targeting generalized functional safety problems in the operating system domain based on the DHR architecture. Yijing Song and Huasheng Dai contributed to the article drafting. Jinhu Jiang and Weihua Zhang contributed to the article revision.

Acknowledgements

We appreciate the anonymous reviewers for their constructive comments. We are grateful to support from the National Natural Science Foundation of China (No. 62141211) and the Open Project Program of the State Key Laboratory of Mathematical Engineering and Advanced Computing.

Funding

This work was supported in part by National Natural Science Foundation of China (No. 62141211), and in part by the Open Project Program of the State Key Laboratory of Mathematical Engineering and Advanced Computing.

References

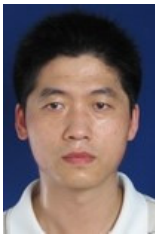
[1] Wu J X. Problems and solutions regarding generalized functional safety in cyberspace. Security and Safety, 2022, 1: 2022001.
[2] ISO – International Organization for Standardization. ISO 26262 Road Vehicles Functional Safety. 2011. https://quality-one.com/iso-26262/.
[3] Charette R N. This car runs on code. In: IEEE spectrum. United States: IEEE, February 2009.
[4] Parkinson S, Ward P, Wilson K, et al. Cyber threats facing autonomous and connected vehicles: Future challenges. IEEE transactions on intelligent transportation systems, 2017, 18(11): 2898-2915.
[5] AcidRain Malware and Viasat Network Downtime in Ukraine: Assessing the Cyber War Threat. 2022. https://www.justsecurity.org/83021/acidrain-malware-and-viasat-network-downtime-in-ukraine-assessing-the-cyber-war-threat/.
[6] Linux devices 'increasingly' under attack from hackers, warn security researchers. 2022. https://www.zdnet.com/article/linux-devices-increasingly-under-attack-from-hackers-warn-security-researchers/.

[7] Ongaro D, Ousterhout J. In search of an understandable consensus algorithm. In: 2014 USENIX Annual Technical Conference. USENIX Association, 2014, 305-319.
[8] Jajodia S, Ghosh AK and Swarup V. Moving Target Defense. China: Springer, August 2011.
[9] Yao D, Zhang Z, Zhang G F, et al. A Survey on Multi-Variant Execution Security Defense Technology. Journal of Cyber Security, 2020, 5(5): 77-94.
[10] Österlund S, Koning K, Olivier P, et al. kMVX: Detecting kernel information leaks with multi-variant execution. In: Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems. ACM Press, 2019, 559-572.
[11] Baumann A, Barham P, Dagand P E, et al. The multikernel: a new OS architecture for scalable multicore systems. In: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. ACM Press, 2009, 29-44.
[12] Wentzlaff D, Agarwal A. Factored operating systems (fos) the case for a scalable operating system for multicores. ACM SIGOPS Operating Systems Review, 2009, 43(2): 76-85.
[13] Nightingale E B, Hodson O, McIlroy R, et al. Helios: heterogeneous multiprocessing with satellite kernels. In: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. ACM Press, 2009, 221-234.
[14] Barbalace A, Ravindran B, Katz D. Popcorn: a replicated-kernel OS based on Linux. In: Proceedings of the Linux Symposium. Ottawa, 2014.

YiJing Song is now working toward the graduate degree in the Computer Science School, Fudan University and working in the Parallel Processing Institute. Her work is related to operating systems and so on.

HuaSheng Dai is now working toward the graduate degree in the Computer Science School, Fudan University and working in the Parallel Processing Institute. His work is related to operating systems and so on.

JinHu Jiang received the master degree in computer science from Shanghai Jiao Tong University, in 2004. He is currently a senior engineer of Parallel Processing Institute, Fudan University. His research interests include operating systems, computer architecture as well as distributed systems.



WeiHua Zhang received the PhD degree in computer science from Fudan University, in 2007. He is currently a professor of Parallel Processing Institute, Fudan University. His research interests include compilers, computer architecture, parallelization, and systems software.