MeSHwA: The case for a Memory-Safe Software and Hardware Architecture for Serverless Computing

Anjo Vahldiek-Oberwagner
Intel Labs
Berlin, Germany

Mona Vij
Intel Labs
Hillsboro, USA

Abstract

Motivated by developer productivity, serverless computing, and microservices have become the de facto development model in the cloud. Microservices decompose monolithic applications into separate functional units deployed individually. This deployment model, however, costs CSPs a large infrastructure tax of more than 25% [27, 53]. To overcome these limitations, CSPs shift workloads to Infrastructure Processing Units (IPUs) like Amazon's Nitro [14] or, complementary, innovate by building on memory-safe languages and novel software abstractions [19, 56].

Based on these trends, we hypothesize a Memory-Safe Software and Hardware Architecture providing a general-purpose runtime environment to specialize functionality when needed and strongly isolate components. To achieve this goal, we investigate building a single address space OS or a multi-application library OS, possible hardware implications, and demonstrate their capabilities, drawbacks and requirements. The goal is to bring the advantages to all application workloads including legacy and memory-unsafe applications, and analyze how hardware may improve the efficiency and security.

1 Introduction

Serverless computing, and microservices have recently gained in popularity to deploy cloud services. They improve developer productivity by focusing on business logic and separating functional services. Function-as-a-Service (FaaS) is a special form of serverless computing. In an extreme case microservices implement functional decomposition similar to FaaS. With FaaS offering the smallest services, the techniques are on a spectrum for service size. Throughout this paper we will refer to serverless computing, microservices and FaaS interchangeably and refer to the application as a service. Cloud Service Providers (CSPs) run and provide the infrastructure software connecting each service. This abstraction comes at the cost of about 25% infrastructure tax as reported by Google [27, 53], consisting of various overheads due to small units communicating frequently over network. Large CSPs react by shifting the infrastructure software into Infrastructure Processing Units (IPUs), which operate more cost effectively than traditional server CPUs offering higher performance for workloads. While this shifts the burden and reduces cost, this approach does not reduce the communication cost, or provide a software architecture eliminating the inherent overheads associated with this deployment model. Alternatively, CSPs built special purpose environments relying on memory-safe languages [19, 56] which require specific compiler and runtime environments and deny their optimizations to legacy applications.

To avoid the inherent overheads in today's hardware and software architecture, we suggest leveraging memory-safe languages for performance and investigate hardware optimizations generalize the environment. Memory-safe languages (e.g., Rust) and runtimes (e.g., Wasm) rely on the compiler to generate binary code preventing memory access violations via static and runtime checks. Their performance is comparable to native execution in case of Rust [45] and shows moderate overheads for Webassembly (Wasm) [24]. Serverless computing could benefit from this environment by running all services in the same process with near-zero communication cost while being isolated from each other. At the same time, the runtime specializes functionality to, e.g., access devices avoiding OS overheads.

These two infliction points, the advance in serverless computing and memory-safe languages, suggests that we should revisit and explore a memory-safe software and hardware architecture providing a general-purpose runtime environment to specialize functionality when needed and strongly isolate services. Our goal is to leverage memory-safe software guarantees where possible, and describe the required hardware to further improve performance and security. The solution needs to generalize to legacy and memory-safe services and offer easy adoption. We will analyze the path of a new Single Address Space Operating System (SASOS) using Rust and Wasm, as well as a library Operating System (library OS) allowing multiple services to execute in parallel while also depending on a traditional host OS. While this idea is not new and was tried in industry [12, 21], we believe the serverless and other cloud workloads demand a drastic change in hardware and software.

By leveraging memory-safe languages our architecture can benefit from a single memory abstraction to drastically improve switching and communication between services. This is in harsh contrast to traditional hardware and software abstractions which rigidly isolate processes, operating systems (OS), address spaces, and privilege levels. To achieve this goal, services and OS functionality execute in the same address spaces and privilege level.

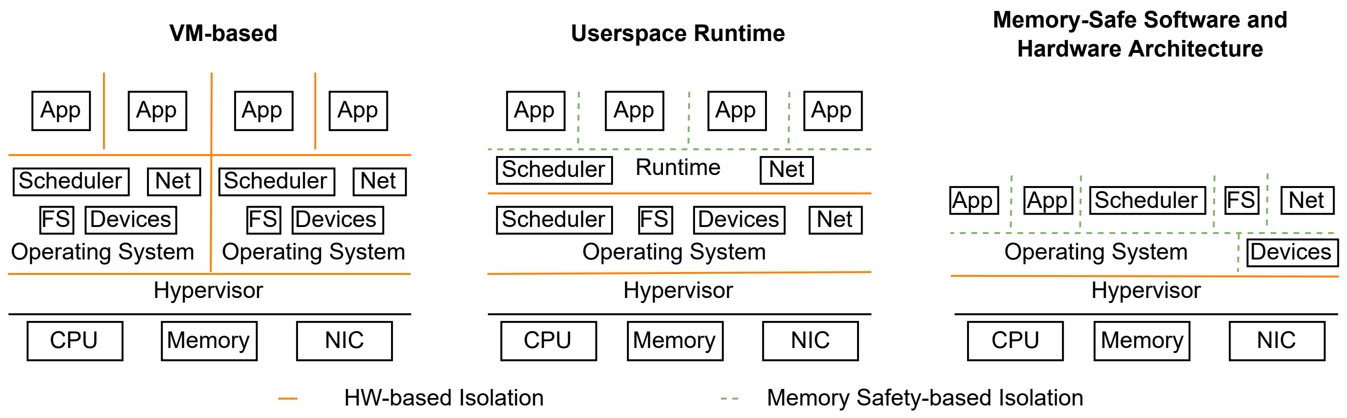

Figure 1. Comparison between cloud software architecture

Sharing becomes instant between a service and the OS. Existing research [4, 10, 20, 36, 37] on improving process-based isolation suffers from too high runtime overheads. Research has focused on SASOS [21] and library OSes [3, 5, 41, 46, 54, 63] as the two alternative approaches to bridge this gap. SASOS overcome existing inefficiencies in traditional OSes by executing the OS and the service in the same address space while offering modularity. Recent advances in memory-safe languages and runtimes offer the building blocks to restart SASOS efforts, but require extensive engineering effort to build a new OS and further improvements to secure memory-safe environments to prevent advanced attacks (e.g., transient execution attacks [40]). In contrast, library OSes move kernel functionality into the userspace and offer specialized alternatives like kernel-bypasses for networking [63]. Instead of writing an OS from scratch, library OSes focus on a specific technique and use the host OS to support the remaining functionality. These approaches typically lack the support for running multiple services at the same time while properly isolating their execution.

We discuss the main beneficiaries of MeSHwA including microservices, FaaS and memory-bandwidth intensive workloads such as machine learning. An inefficient software architecture causes the infrastructure tax to be up to 25% [27]. The remainder of this paper discusses how to build and optimize MeSHwA for such workloads.

2 Background

Modern Cloud Deployments. Server applications increasingly replicate functionality traditionally found in operating systems. Their intent is to improve performance by specializing functionality to the workload, and provide a portable environment across operating systems and hardware. In case of Fastly and Cloudflare [19, 56] most functionality is executed in userspace reducing context switch overheads and infrastructure tax. These providers achieve this goal by executing functions in the services together with a userspace scheduler, memory management, and dedicated userspace network stacks. Library OS [5, 46, 54, 63] similarly improve performance of applications via reducing system calls and bypassing the kernel for network and storage requests with specialized libraries [11]. Figure 1 shows common software and hardware architectures of CSP deployments.

The trend to replicate functionality in the userspace has two main drawbacks. First, applications require specialized implementations of functionality that has a standard OS interface hindering the reuse of specialized implementations. Second, due to the added features, a larger code base offers more opportunities to be exploited, thereby increasing the need to isolate fault domains.

Memory-Safe Languages. Memory-safe languages and runtimes [16, 29, 61] improve the developer productivity and application security by eliminating memory-safety violations such as buffer overflows [62]. While memory safety guarantees vary, memory safe languages generally limit any memory access of an application to a subset of the entire address space, and can be combined control flow integrity to avoid bypassing or altering runtime security checks. The most common forms are object-based memory safety which enforces an access granularity on an object-level, whereas virtual machine-based memory safety only limits access to the memory area defined by the virtual machine.

Recently, Rust [29] and Webassembly (Wasm) [16] innovated the state of the art in two different directions. Throughout this paper we refer to Rust and Wasm as a representative for their class of memory-safe language. Rust is a compiler-enforced memory-safe language and instead of relying on garbage collection, relies on the model of borrowing and ownership of memory objects. Recent work focuses on proving security for these unsafe environments [26], providing secure environments limiting the capabilities of unsafe code [28], or isolating components [7]. The Wasm specification describes both, a byte code and a virtual machine executing the byte code, providing security guarantees regarding the control flow and memory boundaries. Recent work recognizes the importance of memory-safe languages and suggests their use in a moonshot project to build a novel software architecture with supporting hardware [43].

Single Address Space OS (SASOS). Before CPUs supported multiple address spaces and page tables, OSes were built assuming a single address space. These SASOS relied on capability systems [8] and software fault isolation [21] to isolate multiple applications and OS components. Most recently, Microsoft with the Midori and Singularity OS [12, 21] tried to build a SASOS to improve the execution of distributed workloads and formally verify security guarantees of the toolchain and system. Additionally, CSPs more and more rely on unikernels [31, 38, 54] to improve performance. The efficiency of these systems lies in the near-zero cost of sharing memory and low-cost context switching.

Memory-safe Library OS. Demikernel [63], Occlum [51], FaaSm [52], or cubicleOS [48] are library OSes (library OS) reling on memory-safe languages to reduce the attack surface of the library OS and FaaS runtimes like FaaSm [52] additionally isolate functionality using Wasm. These efforts hint towards an environment in which most software executes within a memory-safe environment, either by being written in a memory-safe language (e.g., Rust) or being compiled to Wasm which offers sandbox isolation guarantees.

3 Memory-Safe Software and Hardware Architecture

In this section we provide an understanding of how memory-safe languages and runtimes help building a novel software runtime environment and what hardware optimizations and features improve performance.

3.1 Isolating Services

MeSHwA relies on memory-safe languages and runtimes to isolate memory of services, build independent fault domains [6], and enable sharing to allow efficient use of services within the system. The goal is to build these abstractions without relying on traditional CPU capabilities such as address space identifiers, rings, supervisor mode or privileged instructions.

To isolate memory accesses of services, MeSHwA relies on Software Fault Isolation (SFI) [59] techniques in compilers or interpreters, or compiler-enforced object-granular memory-safety. Since memory safety guarantees vary between languages and runtimes, their capabilities have to be analyzed, understood and measures taken to ensure that each service is isolated. Recent work [7] analyzes the memory safety properties of Rust for isolating services.

We suggest researching a unifying abstraction layer building a foundation across languages and runtimes providing different memory-safety properties. The difference between Rust and Wasm, for example, lies in the ability of Rust to enforce memory safety on a per object level as long as the type system is respected, whereas Wasm enforces memory safety at a coarse granularity limiting memory to the entire Wasm virtual machine memory space. Rust's type system helps when two Rust implementations exchange data. In contrast, sharing data across two independent Wasm modules requires new techniques to make the same memory available to both modules at the same time. Ideally, hardware and software mechanisms enable sharing and isolation across different languages and runtimes allowing for efficient software-only sharing when possible and using hardware to enforce sharing when the software mechanisms do not enforce fine-grained memory safety.

A unified memory-safe service isolation provides the basis for private memory in each service. In case these techniques allow unsafe execution (no memory safety properties during the execution), the compiler needs to either prevent their execution or further restrict access by deploying runtime bounds checks. To avoid control flow vulnerabilities bypassing the memory-safety measures, the compiler additionally needs to restrict and control jump and call targets to the set of potential landing targets. Both, Rust and Wasm, achieve these goals with no or minor changes.

3.2 MeSHwA OS or library OS

We discuss two alternatives to realize MeSHwA, a memory-safe OS and a multi-service memory-safe library OS. Our analysis considers existing work on building Rust-based OSes [9, 33–35]. Building a new OS provides the greatest possible flexibility and avoids some of the existing inherent overheads that traditional OSes bring. Its downside is the additional implementation overhead and a long path towards adoption. Alternatively, a library OS implementation allows adoption in an existing environment such as Linux. Due to the dependence on a host OS, not all of the hardware, abstraction and scheduling optimizations can be implemented with a library OS limiting the possible performance gain.

The main difference between the two alternatives, is how the MeSHwA OS handles interrupts, timers, and other low-level traditional OS functionality by itself, whereas the library OS needs to communicate with the hosting OS to achieve these behaviors. A prominent example is the control over page tables which is impossible without changes to a traditional OS from the userspace. Page table-based optimizations will be much harder with a library OS-based approach and their efficient setup or modification could limit performance benefits.

System Calls and interactions. Traditionally, an application accesses and shares information with the host or network-connected services via the system call interface. Our goal is to provide an extensive interface resembling dynamic library loading rather than fixed system calls. To allow memory-safe services to exchange information, the memory access capabilities have to be transferred. When two Rust services communicate, they could agree on the same underlying type system of the exchanged memory and continue to adhere to the memory safety properties. Similarly, Rust's ownership model lends itself to this architecture to borrow the output of one service to another. To keep track of this borrowing, a runtime service has to keep track of the most recent owner to ensure proper destruction of the memory once it is no longer in use.

Today memory-safe languages do not support sharing types systems across service domains. Each language would have to be extended to support this type system. With languages like Wasm, such cross-service communication is not protected by the memory safety properties and even breaks the memory access model of Wasm virtual machines which assume a bounded memory size.

To overcome these limitations, we suggest a software-only based technique which offers proxy access. A proxy access allows a service to switch to a function with access to the shared memory object (e.g., via a list of proxy functions accessible to the function) while not switching the service's context. As a result, access to the shared memory is as quick as a function call, and such capabilities can be arbitrarily granted by generating the according proxy access function. A proxy-aware compiler could generate proxy functions and translate memory accesses into proxy functions calls to transparently switch between the two.

Common Services. Both alternatives have to provide common OS functionalities such as a file system, network connections, and per service specific metadata such as file descriptors. To allow specialization for workloads, only a minimal substrate should be provided, and dependencies should be selected by each service. The initially needed building blocks are a service loader to enable execution of services, and an interface for each service to discover other services providing certain functionality. In this regard, discovering a service is possible by name and leads to loading a function pointer table into the execution of the service. Any subsequent communication happens via this function table instead of calling into the OS via system call. As a result, a system call becomes a function call instead. All the service switching mechanics, if needed, are hidden within the function call pointers as prepared by the only real system call to discover other services. This design offers the most flexibility, like microkernels [1, 17], and offers the ability to build an abstraction layer for legacy applications mapping the new interface to previous interfaces when needed.

As part of the discovery service, the system negotiates between different memory safety properties, take advantage of fine-grained properties, or rely on hardware-provided capabilities to allow efficient access. For instance, assuming all services and services were built in Rust and the supply chain would provide evidence of this fact, the system could rely on Rust's internal type system to protect access capabilities when services exchange information.

3.3 Hardware implications

As discussed in previous sections, some of the traditional hardware functionalities can alternatively be provided by relying on memory safety instead. Consequently, MeSHwA offers several avenues to further optimize hardware for improved performance and security.

Address translation and caches. Traditional address translation in CPUs relies on page tables and caches within the CPU to reduce the number of page walks. The translation lookaside buffer (TLB) caches recent translations between virtual and physical memory. In case a miss occurs, the page miss handler (PMH) walks the actual page table to find a possible mapping. This takes several hundred cycles to complete and requires memory bandwidth to receive the page table entries from memory. Services with high memory bandwidth demands particularly suffer when frequent page walks occur which reduce the memory bandwidth [15].

Wasm simplifies the memory model in its virtual machine specification to a linearly growing memory area. For cloud services, the required memory size is typically known before launch and described in deployment files. This simplified memory layout with its allocations and mappings can be coalesced into larger regions and ideally mapped such that their address translation can be performed statically. Such a static mapping removes entries from the TLB lowering the pressure for most memory accesses and avoids any page table walks, since the mapping is statically configured. To avoid fragmentation, different power of 2 buckets could be created in physical space to allocate different maximum memory sizes. Additional research is needed to adopt these optimizations to memory-safe-only environments and further improve the efficiency and performance of hardware mechanisms.

Hardware-support for efficient memory sharing. In MeSHwA memory-safe languages restrict memory accesses of a service and do not allow arbitrary sharing between services limiting the efficiency of communication. To overcome this restriction, hardware and software techniques can provide efficient ways to communicate. Traditional OSes solve this problem via shared memory, but MeSHwA shares all virtual memory automatically. As a result, only the memory-safety guarantees restrict accesses to arbitrary memory. We can ideally design techniques to safely open this restrictive design while maintaining the performance and security guarantees that memory-safe languages provide.

Hardware can assist the memory safe language by providing low level access capabilities to allow sharing of memory when the memory safe language does not allow such fine-grained access permissions like Wasm or in case of Rust when the type system is not known at compilation time.

Existing work on process-based isolation [49, 57, 58], provides a capability-based system based on page tables. In these systems, fine grained memory sharing is enabled via specially crafted memory capabilities allowing multiple processes to share access to memory with other processes.

To achieve similar sharing within the same virtual address space, CHERI [60] or Cryptographic Computing [32] provide the ability to set capabilities from within the same virtual address space without involvement of the OS.

Constraining legacy applications. MeSHwA adoption is limited by implementing new or reimplementing existing services for this architecture. Translating existing services automatically allow faster adoption, but requires them to become memory safe. The service needs to be restricted to the service' memory and control flow boundaries.

Wasm can be used as compilation target in common compilers today and transform existing applications to a virtual machine definition limiting access of the application into a 4GB memory and providing a well-structured control flow sufficiently restricting the application to be used in MeSHwA system. Unfortunately, Wasm degrades performance compared to native and requires access to source code.

In contrast to such a software-only approach, are hardware-based techniques [18, 22, 37, 47, 50, 55] (mostly Intel® MPK-based) which rely on CPU features to restrict memory accesses. Unfortunately, these systems require elaborate security monitors [22] and the hardware features may not be readily available. As a result, novel architecture extensions could help translate legacy applications into MeSHwA.

4 Use Case: Microservices and Service Meshes

MeSHwA improves the efficiency of cloud workloads. These workloads benefit from the efficient resource sharing, specialization at the OS-level or efficiently bypassing the OS.

Several cloud services are implemented as microservices, a set of functionally independent, but highly connected components. These components are deployed via an orchestration framework and connected via a service mesh governing access between different components and allowing to observe the behavior of the service. Existing frameworks deploy such workloads via containers and virtual machines completely isolating each component from each other. In addition, service meshes [23] deploy proxies like envoy [13], intercepting all network communication in a co-located container. As a result network messages have to be copied between containers and OS multiple times and objects marshaled.

To avoid this overhead, MeSHwA deploys service mesh proxies and components in the same address space allowing them to communicate and invoke each other via shared interfaces. In addition, if components are co-located their communication can be short-circuit as well, and a service mesh proxy could even eliminate itself from the communication path. As a result, we copy network messages less and marshalling/unmarshalling may not be necessary. Netbricks [44] suggests similar optimizations for network functions.

5 Discussion

MeSHwA relies on memory-safe languages instead of traditional hardware functionality and as a result, offers several avenues to optimize hardware for improved performance and security. This direction reduces the reliance on long-used and battle-proven isolation techniques. Memory-safe languages and runtimes provide certain security properties, but their capabilities are limited with respect to hardware-based attacks and protecting against them may incur about 2x runtime overhead [40]. CHERI [60] and Cryptographic Computing [32] offer capability-based replacements that MeSHwA could rely on to strengthen the security properties. Alternative, and more light-weight approaches should be considered like bringing back 32-bit segmentation or the RISC-V J extension [2]. Recently, hardware-based light-weight subprocess isolation has been an active area of research with promising techniques like Donky [50] suggesting a RISC-V extension similar to Intel® MPK or research using Intel® MPK for isolation [18, 22, 55]. Additional exploration and research in this hardware and software co-design can bridge the gap and overcome today's performance and security limitations.

The security foundation of MeSHwA depends on the memory-safe compilation tool chain and a small, but important runtime component. The runtime component needs to load services within the single address space while respecting the security requirements of the memory-safe language's toolchain. Each toolchain may have different requirements regarding the memory layout (e.g., Wasm requires an 8 GB space around each module's heap). Some toolchains, e.g. Webassembly, lends itself to runtime verification allowing to check binary code for memory safety guarantees before starting to execute it [25]. Another path to establish trust in each toolchain is to vet and certify it, and securely record the supply chain of the service. Before starting a service the runtime would authenticate the trusted toolchain and the metadata of a service [39]. Alternatively, proof-carrying-code [42] can establish trust in the binary of a service. To further strengthen the security and reduce the dependencies, future research should explore formal verification and trusted supply chains.

Several research prototypes [36, 40, 52, 55] run services in the same address space to improve communication overhead. Fastlane [30] is the first to automatically combine multiple services running inside containers to be combined into a single container. Their prototype efficiently combines Python-based services and carefully parallelizes service invocations to improve performance. While Fastlane proposes an interesting direction to improve the current environment, it is limited to a single language, weaker hardware isolation with limited security guarantees. In contrast, MeSHwA focuses on the use of memory-safe languages and leveraging hardware optimizations for optimal performance and security tradeoff between hardware and software.

6 Conclusion

Cloud workloads have shifted towards deploying small functional units heavily communicating within a single machine or across a set of machines leading to an unsustainable infrastructure tax of about 25%. In this paper, we hypothesize a Memory-Safe Software and Hardware Architecture to reduce these overheads and generalize the environment to a larger set of applications. We argue that existing languages and runtimes provide sufficient security guarantees to implement a MeSHwA and take advantage of the benefits of memory-safe languages and runtimes for these workloads. We highlight open research questions and potential optimizations to further improve the architecture.

Acknowledgments

We would like to thank the reviewers of SecDev'22, WORDS'22, Nathan Dautenhan, Fangfei Yang, Shravan Narayan, Mansour Alharthi and several Intel colleagues for their insightful feedback and discussion.

References

[1] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for UNIX development. 1986.

[2] Martin Maas RISC-V TEE Adam Zabrocki, Lee Campbell and J Extension Task Groups. RISC-V Pointer Masking proposal. https://github.com/riscv/riscv-j-extension/blob/master/pointer-masking-proposal.adoc, 2021. Access: 2022-5-4.

[3] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. Sand: Towards high-performance serverless computing. In *Proc. of the Usenix Annual Technical Conference (ATC)*, 2018.

[4] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged CPU features. In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[5] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. Ix: a protected dataplane operating system for high throughput and low latency. In *Proc. of the Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[6] D Elliott Bell and Leonard J La Padula. Secure computer system: Unified exposition and multics interpretation. Technical report, MITRE CORP BEDFORD MA, 1976.

[7] Anton Burtsev, Dan Appel, David Detweiler, Tianjiao Huang, Zhaofeng Li, Vikram Narayanan, and Gerd Zellweger. Isolation in Rust: What is Missing? In *Proc. of the 11th Workshop on Programming Languages and Operating Systems*, 2021.

[8] Jeff Chase, Hank Levy, Miche Baker-Harvey, and Eld Lazowska. Opal: a single address space system for 64-bit architecture address space. In *Proc. of the Workshop on Workstation Operating Systems*, 1992.

[9] Redox Community. Redox os. https://www.redox-os.org/, 2022. Accessed: 2022-5-4.

[10] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. Nested kernel: An operating system architecture for intra-kernel privilege separation. In *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[11] DPDK.org. DPDK: Data Plane Development Kit. http://www.dpdk.org, 2022. Accessed: 2022-5-4.

[12] Joe Duffy. Blogging about Midori. http://joeduffyblog.com/2015/11/03/blogging-about-midori/, 2015. Accessed: 2022-5-4.

[13] Envoy.io. Envoy Proxy. http://www.envoyproxy.io, 2022. Accessed: 2022-5-4.

[14] Brendan Gregg. AWS EC2 Virtualization 2017: Introducing Nitro. https://www.brendangregg.com/blog/2017-11-29/aws-ec2-virtualization-2017.html, 2021. Accessed: 2022-5-4.

[15] Siddharth Gupta, Atri Bhattacharyya, Yunho Oh, Abhishek Bhattacharjee, Babak Falsafi, and Mathias Payer. Rebooting virtual memory with midgard. In *Proc. of the International Symposium on Computer Architecture (ISCA)*, 2021.

[16] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with WebAssembly. In *Proc. of the ACM SIGPLAN Conference on*

Programming Language Design and Implementation (PLDI), 2017.

[17] Per Brinch Hansen. The nucleus of a multiprogramming system. *Communications of the ACM*, 13(4), 1970.

[18] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L Scott, Kai Shen, and Mike Marty. Hodor:Intra-Process Isolation for High-Throughput Data Plane Libraries. In *Proc. of the USENIX Annual Technical Conference (ATC)*, 2019.

[19] Pat Hickey. Announcing Lucet: Fastly's native WebAssembly compiler and runtime. https://www.fastly.com/blog/announcing-lucet-fastly-native-webassembly-compiler-runtime, 2019. Accessed: 2021-7-27.

[20] Terry Ching-Hsiang Hsu, Kevin Hoffman, Patrick Eugster, and Mathias Payer. Enforcing least privilege memory views for multithreaded applications. In *Proc. of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[21] Galen C Hunt and James R Larus. Singularity: rethinking the software stack. *ACM SIGOPS Operating Systems Review*, 41(2), 2007.

[22] Bumjin Im, Fangfei Yang, Chia-Che Tsai, Michael LeMay, Anjo Vahldiek-Oberwagner, and Nathan Dautenhahn. The Endokernel: Fast, Secure, and Programmable Subprocess Virtualization. *arXiv preprint arXiv:2108.03705*, 2021.

[23] Istio.org. Istio Service mesh. http://www.istio.io, 2022. Accessed: 2022-5-4.

[24] Abhinav Jangda, Bobby Powers, Emery D Berger, and Arjun Guha. Not so fast: Analyzing the performance of webassembly vs. native code. In *Proc. of the USENIX Annual Technical Conference (ATC)*, 2019.

[25] Evan Johnson, David Thien, Yousef Alhessi, Shravan Narayan, Fraser Brown, Sorin Lerner, Tyler McMullen, Stefan Savage, and Deian Stefan. Доверя́й, но проверя́й: SFI safety for native-compiled Wasm. In *NDSS*. Internet Society, 2021.

[26] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: Securing the foundations of the Rust programming language. *Proc. of the ACM on Programming Languages (POPL)*, 2017.

[27] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *Proc of the International Symposium on Computer Architecture (ISCA)*, 2015.

[28] Paul Kirth, Mitchel Dickerson, Stephen Crane, Per Larsen, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. PKRU-safe: automatically locking down the heap between safe and unsafe languages. In *Proc. of the European Conference on Computer Systems (EuroSys)*, 2022.

[29] Steve Klabnik and Carol Nichols. *The Rust Programming Language (Covers Rust 2018)*. No Starch Press, 2019.

[30] Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu. Faastlane: Accelerating Function-as-a-Service workflows. In *Proc. of the USENIX Annual Technical Conference (ATC)*, 2021.

[31] Simon Kuenzer, Vlad-Andrei Bădoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Ştefan Teodorescu, Costi Răducanu, et al. Unikraft: fast, specialized unikernels the easy way. In *Proc. of the European Conference on Computer Systems (EuroSys)*, 2021.

[32] Michael LeMay, Joydeep Rakshit, Sergej Deutsch, David M Durham, Santosh Ghosh, Anant Nori, Jayesh Gaur, Andrew Weiler, Salmin Sultana, Karanvir Grewal, et al. Cryptographic Capability Computing. In *Proc. of the Annual IEEE/ACM International Symposium on Microarchitecture (Micro)*, 2021.

[33] Amit Levy, Michael P Andersen, Bradford Campbell, David Culler, Prabal Dutta, Branden Ghena, Philip Levis, and Pat Pannuto. Ownership is theft: Experiences building an embedded os in rust. In *Proc. of the Workshop on Programming Languages and Operating Systems (PLOS)*, 2015.

[34] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. Multiprogramming a 64kb computer safely and efficiently. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, 2017.

[35] Amit Levy, Bradford Campbell, Branden Ghena, Pat Pannuto, Prabal Dutta, and Philip Levis. The case for writing a kernel in rust. In *Proceedings of the Asia-Pacific Workshop on Systems (APSys)*, 2017.

[36] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. Light-Weight Contexts: An OS Abstraction for Safety and Performance. In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[37] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In *Proc. of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[38] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. *ACM SIGARCH Computer Architecture News*, 41(1), 2013.

[39] Marcela S. Melara and Mic Bowman. Enabling security-oriented orchestration of microservices. *arXiv preprint arXiv:2106.09841*, 2021.

[40] Shravan Narayan, Craig Disselkoen, Daniel Moghimi, Sunjay Cauligi, Evan Johnson, Zhao Gang, Anjo Vahldiek-Oberwagner, Ravi Sahita, Hovav Shacham, Dean Tullsen, et al. Swivel: Hardening WebAssembly against Spectre. In *Proc. of the USENIX Security Symposium (USESEC)*, 2021.

[41] Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, Zhaofeng Li, Gerd Zellweger, and Anton Burtsev. RedLeaf: Isolation and Communication in a Safe Operating System. In *Proc. of the Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[42] George C Necula. Proof-carrying code. In *Proceedings of the ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL)*, 1997.

[43] Hamed Okhravi. A cybersecurity moonshot. *IEEE Security & Privacy*, 19(3), 2021.

[44] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. NetBricks: Taking the V out of NFV. In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[45] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Energy efficiency across programming languages: how do energy, time, and memory relate? In *Proc. of the ACM SIGPLAN International Conference on Software Language Engineering (SLE)*, 2017.

[46] Simon Peter, Jialin Li, Irene Zhang, Dan RK Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. *ACM Transactions on Computer Systems (TOCS)*, 33(4), 2015.

[47] Elijah Rivera, Samuel Mergendahl, Howard Shrobe, Hamed Okhravi, and Nathan Burow. Keeping safe rust safe with galeed. In *Proc. of the Annual Computer Security Applications Conference (ACSAC*, 2021.

[48] Vasily A Sartakov, Lluís Vilanova, and Peter Pietzuch. CubicleOS: a library OS with software componentisation for practical isolation. In *Proc. of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021.

[49] Vasily A. Sartakov, Lluís Vilanova, Eyers David, Takahiro Shinagawa, and Peter Pietzuch. Coffers: Capability-based isolation and sharing for microservices. In *Proc. of the Symposium on Operating Systems Design and Implementation (OSDI)*, 2022.

[50] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. Donky: Domain Keys–Efficient In-Process Isolation for RISC-V and x86. In *Proc. of the USENIX Security Symposium (USESEC)*, 2020.

[51] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. Occlum: Secure and efficient multitasking inside a single enclave of Intel SGX. In *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[52] Simon Shillaker and Peter Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. In *Proc. of the USENIX Annual Technical Conference (ATC)*, 2020.

[53] Akshitha Sriraman and Abhishek Dhanotia. Accelerometer: Understanding acceleration opportunities for data center overheads at hyperscale. In *Proc. of the Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[54] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E Porter. Cooperation and security isolation of library OSes for multi-process applications. In *Proc. of the European Conference on Computer Systems (EuroSys)*, 2014.

[55] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *Proc. of the USENIX Security Symposium (USESEC)*, 2019.

[56] Kenton Varda. Mitigating Spectre and Other Security Threats: The Cloudflare Workers Security Model. https://blog.cloudflare.com/mitigating-spectre-and-other-security-threats-the-cloudflare-workers-security-model/, 2020. Accessed: 2021-7-27.

[57] Lluís Vilanova, Muli Ben-Yehuda, Nacho Navarro, Yoav Etsion, and Mateo Valero. CODOMs: Protecting software with code-centric memory domains. *Proc. of the ACM SIGARCH Computer Architecture News*, 42(3), 2014.

[58] Lluís Vilanova, Marc Jordà, Nacho Navarro, Yoav Etsion, and Mateo Valero. Direct Inter-Process Communication (dIPC) Repurposing the CODOMs Architecture to Accelerate IPC. In *Proc. of the European Conference on Computer Systems (EuroSys)*, 2017.

[59] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. Efficient software-based fault isolation. In *Proc. of the Symposium on Operating Systems Principles (SOSP)*, 1993.

[60] Robert NM Watson, Jonathan Woodruff, Peter G Neumann, Simon W Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, et al. Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*, 2015.

[61] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*, 2009.

[62] Google Project Zero. The More You Know, The More You Know You Don't Know. https://googleprojectzero.blogspot.com/2022/04/the-more-you-know-more-you-know-you.html, 2021. Accessed: 2022-5-4.

[63] Irene Zhang, Jing Liu, Amanda Austin, Michael Lowell Roberts, and Anirudh Badam. I'm not dead yet! the role of the operating system in a kernel-bypass era. In *Proc. of the Workshop on Hot Topics in Operating Systems (HotOS)*, 2019.