Pay-as-you-Go Resource Isolation

Work-In-Progress (WIP) Paper

Miguel Lourenço
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa

Marios Kogias
Imperial College London

Rodrigo Bruno
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa

1 OVERHEAD OF ISOLATING WORKLOADS

Serverless computing delegates provisioning and scaling to the cloud provider so that developers can focus on building and running applications. Its auto-scaling and finer billing granularity make it easy-to-use and cost-efficient. In this paradigm, servers remain idle until an event arises, upon which the infrastructure automatically provisions a sandbox that encapsulates the user code and efficiently handles incoming requests [15]. However, a sandboxed environment (e.g., a container, or a Virtual Machine) can take a long time to initialize, ranging from hundreds of milliseconds to a couple of seconds [25]. One of the main factors contributing to this overhead, also known as cold starts, comes from OS-level isolation primitives such as chroot and namespaces. Figure 1 shows that these operations can be up to 2-3 orders of magnitude more expensive than creating a new process (fork). The setup_chroot column includes not only the creation of a chroot but also the necessary bind mounts to pre-populate the environment with basic system libraries. Similarly, setup_network corresponds to creating a network namespace and setting up and attaching virtual network devices. These results were collected on a Ubuntu machine with Linux kernel 5.15 and Intel(R) Xeon(R) Gold 6138 @ 2.00GHz. Since most serverless functions execute within a few hundred milliseconds [18, 24], reducing the overhead of such isolation primitives is essential to guarantee good resource utilization and user experience.

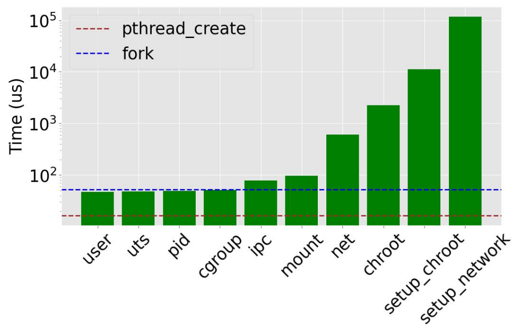

Figure 1: Latency overhead of namespaces and chroots.

2 RESOURCE ISOLATION IN THE WILD

Serverless functions run inside a sandbox that can take the form of a traditional VM or a lightweight microVM, a container, or a memory isolate.

Virtual Machines rely on a hypervisor such as Xen [3] or Linux KVM [6] to coordinate its access to the underlying hardware and keep VM resources separate from one another [12]. VMs have been a critical backbone of cloud computing but, as a result of the demand for high elasticity in serverless environments, microVMs have been proposed. MicroVMs such as Firecracker [14] are lightweight VMs with a minimalist design that excludes support for a wide range of devices, focusing instead on supporting the minimum set of architectures and devices necessary in serverless environments. However, when deployed in production, microVMs are still launched within a resource jail built on top of OS-level isolation primitives [17].

Containers, such as Docker [21] and LXC [8], share the same host OS and rely on isolation mechanisms built into the Linux kernel. These mechanisms include control groups (cgroups), responsible for monitoring and managing resource usage; namespaces, which isolate an application view of the global system resources, like process IDs, and user IDs; chroot, which provides an isolated file-system; and seccomp-bpf, which limits access to system calls [10]. Together, these tools provide a powerful toolkit for isolating containers. Nevertheless, to enforce an additional layer of security, a few studies propose a reduction of the kernel surface, for example, through an application kernel of the Linux system call interface in user space. Gvisor [13] takes this approach at the cost of higher per-system call overhead. Others have combined microVMs with containers (eg. Kata containers [19]) to improve existing secure container technologies. RunD [20] identified a solution that solves three bottlenecks: it implements a more efficient container rootfs that reduces disk usage and its creation time, generates a snapshot of a pre-patched kernel image to reduce image size, and prepopulates a cache with a fixed number of cgroups.

Memory Isolates, such as V8 isolates [1], Native Image isolates [5], and Wasm modules [11] rely on Software-Fault Isolation to enforce memory isolation. Each new memory isolate is created in an existing environment with its disjoint memory heap. This eliminates the cold starts of the virtual machine or container given that the language runtime has already been created and initialized [4]. However, to effectively build a lightweight isolated sandbox using memory isolates, we still need to wrap memory isolates with namespaces and chroots to isolate resource access.

In sum, different sandboxing techniques rely on OS-level isolation mechanisms, whose startup latencies easily dominate the execution time of functions in serverless [23]. Despite this, none previous work explores how to reduce the costs associated with the initialization of those primitives. Techniques such as checkpoint/restore alleviate cold start latency but still always require setting up namespaces and chroots beforehand. Approaches that focus on reusing/caching sandboxes result in high resource utilization and do not directly address the original problem [22]. To the best of our knowledge, this is the first work to directly address the costs of creating and initializing OS-level isolation primitives.

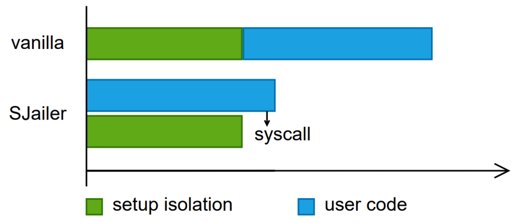

Figure 2: SJailer isolation set up in the background.

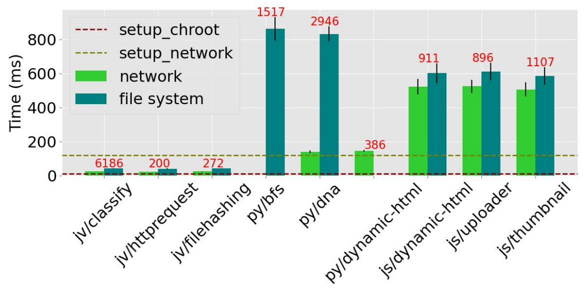

Figure 3: Time to execute the first network/file system-related system call. The total execution time is presented in red.

3 INSIGHT: LAZILY ISOLATING RESOURCES

Most serverless functions are short-lived and sparsely invoked, meaning that cloud providers can only keep a small number of functions warm. As a result, a significant amount of time is spent on setting up sandboxes [24]. When looking at how sandboxes are initialized, we identify that all user code only starts executing after resource isolation is set up. In this work, we explore the following question: *Can user code start before resource isolation is set up?* In an ideal execution, user code starts executing, and only when it requests access to a specific resource through a Linux system call, the execution is moved into an isolated environment, where the request is handled. Figure 2 demonstrates how isolation set up and user code can overlap. For example, namespaces can be created in parallel with user code execution. If the user code executes a system call (e.g. getpid), then it needs to be moved into a pid namespace. If such a namespace is already available, then the request is immediately executed. Otherwise, the system call waits until the namespace is ready.

Lazily moving code execution into namespaces and chroots can alleviate isolation initialization latency if system calls requiring access to those resources are not invoked early in the execution (in this case, there would be no benefit compared to setting up isolation before user code starts). To analyze this window of opportunity, we focused on the top two most expensive isolation primitives: file system and network namespaces. Figure 3 illustrates how different functions ranging from ML inference, to dynamic HTML generation take hundreds of milliseconds before issuing a system call requesting access to the file system and network. In other words, it is possible to delay resource isolation until the user code issues a system call that attempts to access protected resources.

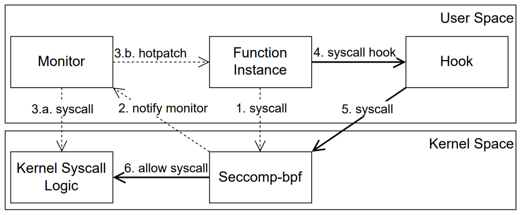

Figure 4: SJailer's lazy resource isolation overview.

4 SJAILER

To achieve lazy resource isolation, we propose SJailer, a system call jail, that intercepts system calls and moves the calling application into a particular namespace depending on the requested resource. If the system call does not require any resource or requires a resource that is already isolated, then it executes as normal. SJailer's goal is to eliminate resource isolation from the sandbox initialization critical path. In the worst case, SJailer's execution becomes equivalent to traditional sandboxes where isolation set up is performed before the user code launches.

Our design relies on system call patching [1] to replace *syscall* instructions with SJailer's hook points, responsible for checking whether or not the user code needs to be further isolated according to the requested system call. However, static system call patching does not guarantee that our solution is efficient and secure as parsing the entire code section to find system call instructions imposes a long latency period and fails to cover scenarios where applications generate code (e.g., using a JIT compiler).

Consequently, SJailer relies on seccomp-bpf [10] to guarantee full coverage of system calls by running a fast kernel program that checks whether or not the calling thread is attempting to execute a system call that has been previously patched. The downside of this approach is that, for security reasons, the seccomp-bpf cannot be removed once installed, and thus, even patched system calls still need to be validated. This incurs a negligible overhead [16].

In practice, users make system calls that are intercepted by the seccomp-bpf, which checks if the system call hasn't been previously patched (Step 1 in Fig. 4). Then, it notifies the Monitor (Step 2), which handles the call accordingly (Step 3.a) and patches the user code to optimize future invocations (Step 3.b). After the first execution of a system call (dashed arrows in Fig. 4), subsequent calls are handled by SJailer's hooks (Step 4) which check if the application should be moved into a particular namespace or chroot. Once the necessary isolation is in place, the system call is issued (Step 5) and validated in the BPF verifier which allows the system call to execute (Step 6).

5 OPEN CHALLENGES

We are currently using seccomp-bpf to distinguish from patched and non-patched syscalls, and so, this layer of security must be consistent to prevent breaches that could compromise the isolation of our solution. For example, we want to guarantee that user code cannot jump directly to syscalls and completely bypass our hook points. Because of this, we are considering Intel MPK [9] or ARM

[1]Our implementation is based on *syscall_intercept* [2].

Memory Overlays [7] to strengthen our validation step. Additionally, instead of patching system calls in SJailer's set up phase, we aim to build a pre-patched version of the user code or libc as an optimization step. Lastly, we want to implement a function that allows the Monitor to patch system calls outside the libc dynamically, after it receives a notification (Step 3.b in Fig 4).

6 ACKNOWLEDGMENTS

This work was funded in part by the Fundação para a Ciência e a Tecnologia under award UIDB/50021/2020.

REFERENCES

[1] [n. d.]. Fine-Grained Sandboxing with V8 Isolates. https://www.infoq.com/presentations/cloudflare-v8/. Accessed: 2024-01-21.
[2] [n. d.]. GitHub - pmem/syscall_intercept: The system call intercepting library. https://github.com/pmem/syscall_intercept. Accessed: 2024-01-21.
[3] [n. d.]. Home - Xen Project. https://xenproject.org/. Accessed: 2024-01-24.
[4] [n. d.]. How Workers works · Cloudflare Workers docs. https://developers.cloudflare.com/workers/reference/how-workers-works/. Accessed: 2024-01-21.
[5] [n. d.]. Isolates and Compressed References: More Flexible and Efficient Memory Management via GraalVM | by Christian Wimmer | graalvm | Medium. https://medium.com/graalvm/isolates-and-compressed-references-more-flexible-and-efficient-memory-management-for-graalvm-a044cc50b67e. Accessed: 2024-01-21.
[6] [n. d.]. KVM. https://www.linux-kvm.org/page/Main_Page. Accessed: 2024-01-24.
[7] [n. d.]. Learn the architecture - AArch64 memory attributes and properties. https://developer.arm.com/documentation/102376/0200/Permission-indirection-and-permission-overlay-extensions/An-example-of-using-permission-indirection-and-permission-overlay-features. Accessed: 2024-01-25.
[8] [n. d.]. Linux Containers. https://linuxcontainers.org/. Accessed: 2024-01-24.
[9] [n. d.]. Memory Protection Keys — The Linux Kernel documentation. https://www.kernel.org/doc/html/next/core-api/protection-keys.html. Accessed: 2024-01-28.
[10] [n. d.]. Seccomp BPF (SECure COMPuting with filters) — The Linux Kernel documentation. https://www.kernel.org/doc/html/v4.19/userspace-api/seccomp_filter.html. Accessed: 2024-01-21.
[11] [n. d.]. WebAssembly Core Specification. https://www.w3.org/TR/wasm-core-1/. Accessed: 2024-01-24.
[12] [n. d.]. What are virtual machines? | IBM. https://www.ibm.com/topics/virtual-machines. Accessed: 2024-01-21.
[13] [n. d.]. What is gVisor? - gVisor. https://gvisor.dev/docs/. Accessed: 2024-01-21.
[14] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX symposium on networked systems design and implementation (NSDI 20)*. 419–434.
[15] Kuldeep Chowhan. 2018. *Hands-on Serverless Computing: Build, run, and orchestrate serverless applications using AWS Lambda, Microsoft Azure Functions, and Google Cloud functions*. Packt Publishing.
[16] Jake Edge. [n. d.]. A seccomp overview [LWN.net]. https://lwn.net/Articles/656307/. Accessed: 2024-01-27.
[17] Radek Gruchalski. [n. d.]. The jailer | gruchalski.com. https://gruchalski.com/posts/2021-02-19-the-jailer/. Accessed: 2024-01-24.
[18] Artjom Joosen, Ahmed Hassan, Martin Asenov, Rajkarn Singh, Luke Darlow, Jianfeng Wang, and Adam Barker. 2023. How Does It Function? Characterizing Long-term Trends in Production Serverless Workloads. In *Proceedings of the 2023 ACM Symposium on Cloud Computing* (, Santa Cruz, CA, USA,) *(SoCC '23)*. Association for Computing Machinery, New York, NY, USA, 443–458. https://doi.org/10.1145/3620678.3624783
[19] Tytus Kurek. [n. d.]. What is Kata Containers and why should I care? | Ubuntu. https://ubuntu.com/blog/what-is-kata-containers. Accessed: 2024-01-21.
[20] Zijun Li, Jiagan Cheng, Quan Chen, Eryu Guan, Zizheng Bian, Yi Tao, Bin Zha, Qiang Wang, Weidong Han, and Minyi Guo. 2022. RunD: A Lightweight Secure Container Runtime for High-density Deployment and High-concurrency Startup in Serverless Computing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 53–68. https://www.usenix.org/conference/atc22/presentation/li-zijun-rund
[21] Dirk Merkel et al. 2014. Docker: lightweight linux containers for consistent development and deployment. *Linux j* 239, 2 (2014), 2.
[22] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 57–70. https://www.usenix.org/conference/atc18/presentation/oakes
[23] Shijun Qin, Heng Wu, Yuewen Wu, Bowen Yan, Yuanjia Xu, and Wenbo Zhang. 2020. Nuka: A Generic Engine with Millisecond Initialization for Serverless Computing. In *2020 IEEE International Conference on Joint Cloud Computing*. 78–85. https://doi.org/10.1109/JCC49151.2020.00021
[24] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 205–218. https://www.usenix.org/conference/atc20/presentation/shahrad
[25] Paulo Silva, Daniel Fireman, and Thiago Emmanuel Pereira. 2020. Prebaking functions to warm the serverless cold start. In *Proceedings of the 21st International Middleware Conference*. 1–13.