Putting the "Micro" Back in Microservice

Sol Boucher*, Anuj Kalia*, David G. Andersen*, and Michael Kaminsky†
* *Carnegie Mellon University* † *Intel Labs*

Abstract

Modern cloud computing environments strive to provide users with fine-grained scheduling and accounting, as well as seamless scalability. The most recent face to this trend is the "serverless" model, in which individual functions, or microservices, are executed on demand. Popular implementations of this model, however, operate at a relatively coarse granularity, occupying resources for minutes at a time and requiring hundreds of milliseconds for a cold launch. In this paper, we describe a novel design for providing "functions as a service" (FaaS) that attempts to be truly *micro*: cold launch times in microseconds that enable even finer-grained resource accounting and support latency-critical applications. Our proposal is to eschew much of the traditional serverless infrastructure in favor of language-based isolation. The result is microsecond-granularity launch latency, and microsecond-scale preemptive scheduling using high-precision timers.

1 Introduction

As the scope and scale of Internet services continues to grow, system designers have sought platforms that simplify scaling and deployment. Services that outgrew self-hosted servers moved to datacenter racks, then eventually to virtualized cloud hosting environments. However, this model only partially delivered two related benefits:

1. Pay for only what you use at very fine granularity
2. Scale up rapidly on demand

The VM approach suffered from relatively coarse granularity: Its atomic compute unit of machines were billed at a minimum of minutes to months. Relatively long startup times often required system designers to keep some spare capacity online to handle load spikes.

These shortcomings led cloud providers to introduce a new model, known as serverless computing, in which the customer provides *only* their code, without having to configure its environment. Such "function as a service" (FaaS) platforms are now available as AWS Lambda [4], Google Cloud Functions [10], Azure Functions [18], and Apache OpenWhisk [5]. These platforms provide a model in which: (1) user code is invoked whenever some event occurs (e.g., an HTTP API request), runs to completion, and nominally stops running (and being billed) after it completes; and (2) there is no state preserved between separate invocations of the user code. Property (2) enables easy auto-scaling of the function as load changes.

Because these services execute within a cloud provider's infrastructure, they benefit from low-latency access to other cloud services. In fact, acting as an access-control proxy is a recurring microservice pattern: receive an API request from a user, validate it, then access a backend storage service (e.g., S3) using the service's credentials.

In this paper, we explore a design intended to reduce the tension between two of the desiderata for cloud functions: low latency invocation and low cost. Contemporary invocation techniques exhibit high latency with a large tail; this is unsuitable for many modern distributed systems which involve high-fanout communication, sometimes performing thousands of lookups to handle each user request. Because user-visible response time often depends on the tail latency of the slowest chain of dependent responses [7], shrinking the tail is crucial [11, 24, 16, 12].

Thus we seek to reduce the invocation latency and improve predictability, a goal supported by the impressively low network latencies available in modern datacenters. For example, it now takes $< 20\mu s$ to perform an RPC between two machines in Microsoft Azure's virtual machines [9]. We believe, however, that fully leveraging this improving network performance will require reducing microservices' invocation latencies to the point where the network is once again the bottleneck.

We further hypothesize—admittedly without much proof for this chicken-and-egg scenario—that substantially reducing both the latency and cost of running intermittently-used services will enable new classes and scales of applications for cloud functions, and in the remainder of this paper, present a design that achieves this. As Lampson noted, there is power in making systems "fast rather than general or powerful" [14], because fast building blocks can be used more widely.

Of course, a microservice is only as fast as the slowest service it relies on; however, recall that many such services are offered in the same clouds and datacenters as serverless platforms. Decreasing network latencies will push these services to respond faster as well, and new stable storage technologies such as 3D XPoint (projected to offer sub-microsecond reads and writes) will further accelerate this trend by offering lower-latency storage.

In this paper, we propose a restructuring of the serverless model centered around low-latency: *lightweight microservices* run in *shared processes* and are isolated primarily with language-based *compile-time guarantees* and *fine-grained preemption*.

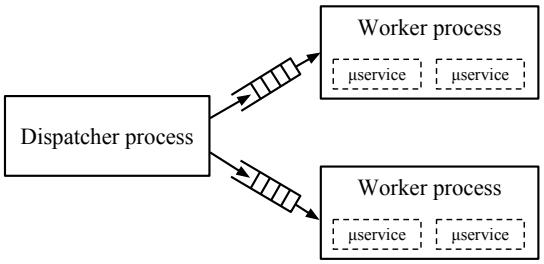

Figure 1: Language-based isolation design. The dispatcher process uses shared in-memory queues to feed requests to the worker processes, each of which runs one user-supplied microservice at a time.

2 Motivation

Our decision to use language-based isolation is based on two experimental findings: (1) Process-level isolation is too slow for microsecond-scale user functions. (2) Commodity CPUs support task preemption at microsecond scale. We conducted our experiments on an Intel® Xeon® E5-2683 v4 server (16 cores, 2.1 GHz) and Linux 4.13.0.[1]

2.1 Process-level isolation is too slow

We use a single-machine experiment to evaluate the invocation overhead of different isolation mechanisms: Microservices run on 14 *worker* CPU cores. Another core runs a *dispatcher* process that launches microservices on the workers. All requests originate at the dispatcher (which in a full serverless platform would forward from a cluster scheduler); it schedules ≤14 microservices at a time, one per worker core, choosing from a pool of 5,000.

To provide a comparison against contemporary system designs, we use two different isolation mechanisms:

1. **Process-based isolation:** Each microservice is a separate process. We expect this approach to exhibit latency at least as low as the container isolation common in contemporary serverless deployments.
2. **Language-based isolation:** Each worker core hosts a single-threaded *worker process* that directly executes different microservices, one at a time. In this approach, shown in Figure 1, a worker process runs a microservice by calling its registered function; we assume that the microservice function can be isolated from the worker process with language-based isolation techniques that we discuss in Section 3. The dispatcher schedules microservices on worker processes by sending them requests on a shared memory queue, which idle worker processes poll.

We use 5,000 copies of a Rust microservice that simply records a timestamp: latency is measured between when the dispatcher invokes a microservice and the time that microservice records. There are two experiment modes:

[1]Source code for the benchmarks in this paper is available from https://github.com/efficient/microservices_microbenchmarks.

Microservices Resident?	Isolation	Latency (µs) Median	99%	Throughput (M invoc/s)
Warm-start	Process	8.7	27.3	0.29
	Language	1.2	2.0	5.4
Cold-start	Process	2845.8	15976.0	–
	Language	38.7	42.2	–

Table 1: Microservice invocation performance

Warm-start requests. We first model a situation where all of the microservices are already resident on the compute node. In the case of process-based isolation, the dispatcher launches all 5,000 microservices at the beginning of the experiment, but they all block on an IPC call; the dispatcher then invokes each microservice by waking up its process using a UDP datagram. In the case of language-based isolation, the microservices are dynamic libraries preloaded into the worker processes.

Table 1 shows the latency and throughput of the two methods. We find that the process-based isolation approach takes 9 µs and achieves only 300,000 warm microservice invocations per second. In contrast, language-based isolation achieves 1.2 µs latency (with a tail of just 2.0 µs) and over 5 million invocations per second.

Considering that the FaRM distributed computing platform achieved mean TATP transaction commit latencies as low as 19 µs in 2015 [8], a 9 µs microservice invocation delay represents almost 50% overhead for a microservice providing a thin API gateway to such a backend. We therefore conclude that even in the average case, process-based isolation is too slow for microsecond-scale scheduling. Furthermore, IPC overhead limits invocation throughput.

Process-based isolation also has a higher memory footprint: loading the 5,000 trivial microservices consumes 2 GiB of memory with the process-based approach, but only 1.3 GiB with the language-based one. However, this benefit may reduce as microservices' code sizes increase.

Cold-start requests. Achieving ideal wakeup times is possible only when the microservices are already resident, but the tail latency of the serverless platform depends on those requests whose microservices must be loaded before they can be invoked. To assess the difference between process-based and language-based isolation in this context, we run the experiment with the following change: In the former case, the dispatcher now launches a transient microservice process for each request by `fork()/exec()`'ing. In the latter, the dispatcher asks a worker to load a microservice's dynamic library (and unload it afterward). The results in Table 1 reveal an order-of-magnitude slip in the language-based approach's latency; however, this is overshadowed by the three orders of magnitude increase for process-based isolation.

2.2 Intra-process preemption is fast

In a complete serverless platform, some cluser-level scheduler would route incoming requests to individual worker nodes. Since we run user-provided microservices directly in worker processes, a rogue long-running microservice could thwart such scheduling by unexpectedly consuming the resources of a worker that already had numerous other requests queued. We hypothesize that, in such situations, it is better for tail latency to preempt the long microservice than retarget the waiting jobs to other nodes in real time. (Only the compute node already assigned a request is well positioned to know whether that request is being excessively delayed: whereas other nodes can only tell that the request hasn't yet *completed*, this node alone knows whether it has been *scheduled*.) At our scale, this means a preemption interval up to two orders of magnitude faster than Linux's default 4 ms process scheduling quantum.

Fortunately, we find that high-precision event timers (HPETs) on modern CPUs are sufficient for this task. We measure the granularity and reliability of these timers as follows: We install a signal handler and configure a POSIX timer to trigger it every T µs. Ideally, this handler would always be called exactly T µs after its last invocation; we measure the deviation from T over 65,535 iterations. We find that the variance is smaller than 0.5 µs for $T \geq 3$ µs. This shows that intra-process preemption is fast and reliable enough for our needs.

3 Providing Isolation

Consolidating multiple users' jobs into a single process requires addressing security and isolation. We aim to do it without compromising our ambitious performance goals.

Our guiding philosophy for achieving this is "language-based isolation with defense in depth." We draw inspiration from two recently-published systems whose own demanding performance requirements drove them to perform similar coalescing of traditionally independent components: NetBricks [19] is a network functions runtime for providing programmable network capabilities; it is unique among this class of systems for running third-party network functions in-process rather than in VMs. Tock [15] is an embedded microkernel whose servers ("capsules") form a common compilation unit and communicate using type-safe function calls. As their primary defense against untrusted code, both systems leverage Rust [3], a new type-safe systems programming language.

Rust is a strongly-typed, compiled language that uses a lightweight runtime similar to C. Unlike many other modern systems languages, Rust is an attractive choice for predictable performance because it does not use a garbage collector. It provides strong memory safety guarantees by focusing on "zero-cost abstractions" (i.e., those that can be compiled down to code whose safety is assured without runtime checks). In particular, safe Rust code is guaranteed to be free of null or dangling pointer dereferences, invalid variable values (e.g., casts are checked and unions are tagged), reads from uninitialized memory, mutations of non-`mut` data (roughly the equivalent of C's `const`), and data races, among other misbehaviors [22].

We require each microservice to be written in Rust (although, in the future, it might be possible to support subsets of other languages by compiling them to safe Rust), giving us many aspects of the isolation we need. It is difficult for microservices to crash the worker process, since most segmentation faults are prevented, and runtime errors such as integer overflow generate Rust panics that we can catch. Microservices cannot get references to data that does not belong to them thanks to the variable and pointer initialization rules.

Given our performance goals, there is a crucial isolation aspect that Rust does not provide: there is nothing to stop users from monopolizing the CPU. Our system, however, must be preemptive. We are unaware of existing preemption techniques that work at microsecond scales. Note that coroutine-like cooperative multitasking approaches (such as lightweight threads in Go [2] and Erlang [1]) are not preemptive, so they do not work for us. We briefly discuss our solution to this in the following section; it depends on installing a `SIGALRM` handler and ensuring that trusted code within the process handles the signal.

Our defense-in-depth comes from using lightweight operating-system protections to block access to certain system calls, as well as the proposed mechanisms in Section 6. Some system calls must be blocked to have any defense at all; otherwise, the microservice could create kernel threads (e.g., `fork()`), create competition between threads (e.g., `nice()`), or even terminate the entire worker (e.g., `exit()`). Finally, user functions should not have unmonitored file system access.

We propose to block system calls using Linux's `seccomp()` system call [20]; each worker process should call this during initialization to limit itself to a whitelisted set of system calls. Prior to lockdown, the worker process should install a `SIGSYS` handler for regaining control from any microservice that attempts to violate the policy.

4 Providing Preemption

The system must be able to detect and recover from microservices that, whether maliciously or negligently, attempt to run for longer than permitted. The two parts of this problem are (1) regaining control of the CPU and (2) aborting and cleaning up after the user code.

As proposed in Section 2, regaining control of the CPU happens when a signal (`SIGALRM`) from the kernel transfers control to the worker process's handler.[2] The handler then checks how long the current microservice

[2]For defense in depth, the worker process should be prevented from subsequently modifying this signal-handling configuration.

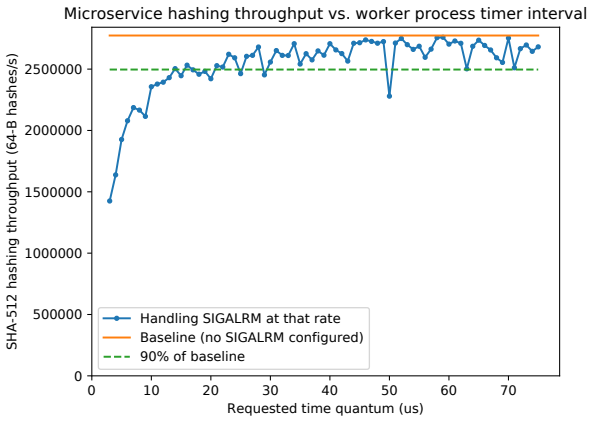

Figure 2: Effect of `SIGALRM` quantum on hashing tput.

has been running and decides whether it should be killed. (We register the handler using the `SA_RESTART` flag to `sigaction()` so that any interrupted blocking syscalls are restarted transparently.) However, there remain three important questions:

For how long should each microservice be allowed to run? Assume that each core executes one user task at a time and that all microservice functions are pre-compiled and resident (warm invocation). We define L to be the desired warm invocation latency, B to be the runtime budget allotted to each microservice, and r_c to be the remaining runtime of the microservice on CPU c. Thus, in the worst case (where all tasks are executing for their entire allotted time) the probability that the incoming microservice will have somewhere to run in time to meet the invocation latency SLO is given by:

$$p(r_{\min} \leq L) = \sum_{c \in C} p(r_c \leq L) = |C|\frac{L}{B} \qquad (1)$$

Given the 14 cores in our setup and imagining we want to keep the 99% tail, $p(r_{\min} \leq L) = 0.99$, to an L of 8 μs, we need to kill tasks running for more than B = 113 μs.

How often should the handler execute (the *quantum*)? We showed in Section 2 that microsecond-scale preemption is *achievable*, but can it be done *efficiently*? To find out, we wrote a microservice that measures the throughput of computing SHA-512 hashes over 64 B of data at a time. We then subjected its worker process to `SIGALRM`s, varying the quantum and observing the resulting hashing throughput. Figure 2 illustrates that by a quantum of about 20 µs, throughput had reached around 90% of baseline. Considering this performance degradation, acceptable we adopt this quantum and prescribe a runtime budget of 113 - 20 = 93 µs so that we can kill over-budget microservices in time to avoid violating our tail latency SLO.

How do we clean up a terminated microservice? We now discuss our mechanism for aborting and cleaning up after a microservice exceeds its runtime budget. POSIX signal handlers receive as an argument a pointer to their *context*, a snapshot of the process's PCB (process control block) at the moment before it received the signal. When the handler returns, the system will transfer control back to the point described by the context, so a naïve way for our worker processes to regain control would be to reset its GPRs (general-purpose registers) to values recorded just before the worker's tight scheduling loop. This approach, however, would not release the microservice's state or memory allocations back to the worker.

One of the few heavyweight components of the Rust runtime is panic handling, reminiscent of C++'s exception handling. The compiler inserts landing pads into each function that call the destructors for the variables in its stack frame: if the program ever panics, the standard library uses these to unwind the stack. We co-opt this functionality by having the `SIGALRM` handler set its context to raise an explicit panic in a fake stack frame just above the real top of the stack.

Section 6 discusses the limitations and security ramifications of this approach.

5 Deployment

We now describe our microservices in the broader context of our full proposed serverless system. We clarify their lifecycle, interactions with the compute nodes, and the trust model for the cloud provider.

Users submit their microservices in the form of Rust source code, allowing the serverless operator to pass the `-Funsafe-code` compilation flag to reject any `unsafe` code. This process need not occur on the compute nodes, provided the deployment server tasked with compilation runs the same version of the Rust compiler.[3] The operator needs to trust the compiler, standard library, and any libraries against which it will permit the microservice to link (since they might contain `unsafe` code), but importantly need not worry about the microservice itself.

We believe that restricting microservices to a specific list of permitted dependencies is reasonable. Any library that contains only safe Rust code could be whitelisted without review. To approximate the size of such a list given the current Rust ecosystem, we turn to a 2017 study [6] by the Tock authors that found just under half of the Rust package manager's top 1000 most-downloaded libraries to be free of unsafe code. They caution that many of those packages have unsafe dependencies, but reviewing a relatively small number of popular libraries would open up the majority of the most popular packages.

If the application compiles (is proven memory-safe) and links (depends only on trusted libraries) successfully, the deployment server produces a shared object file, which the provider then distributes to each compute node on which it might run. Then, in order to ensure that invokers will experience the warm-start latencies discussed in Section 2,

[3] This restriction exists because, as of the latest release (1.23.0) of the compiler, Rust does not have a stable ABI.

those nodes' dispatcher processes should instruct one or more of their workers to preload the dynamic library. If the provider experiences too many active microservices for its available resources, it can unload some libraries; on their next invocation, they will experience higher (cold start) invocation latencies as they synchronously load the dynamic library.

6 Future Work

As noted above, our exploration is preliminary; this section outlines several open questions. These questions fall into two categories: shortcomings in our current implementation and defense-in-depth safeguards against unexpected failures (e.g., compiler bug or the operator allowing use of a buggy or malicious library).

Non-reentrancy. Our use of Rust panics to unwind the stack during preemption can corrupt the internal state of non-reentrant functions (e.g., Rust's dynamic allocator). Possible fixes include blacklisting these functions and delaying preemption until they are finished or replacing the problematic function with a safe one (e.g., a custom memory allocator).

Host process. Our current implementation does not provide isolation between the dispatcher and worker processes. We plan to apply standard OS techniques to reduce the chance of interference by a misbehaving worker. Examples include auditing interactions with the shared memory region to ensure invalid or inconsistent data originating from a worker cannot create an unrecoverable dispatcher error; handling the `SIGCHLD` signal to detect a worker that has somehow crashed; and keeping a recovery log in the dispatcher process so that any user jobs lost to a failed worker process can be reassigned to operational workers.

Further defense in depth with ERIM. ERIM outlines a set of techniques and binary rewriting tools useful for using Intel's Memory Protection Keys to restrict memory access by threads within a process [23]. While preliminary and without source yet available, this appears to be an attractive approach for defense-in-depth both within worker processes and between the workers and the dispatcher.

Library functions. As with system calls, there may exist library functions in Rust (and certainly in libc, which we deny by default) that are unsafe for microservices to access. Because the Rust standard library requires unsafe code, defense-in-depth suggests that a whitelisting-based approach should be employed for access to its functions. Certainly library functions must be masked—for example, our use of Rust's panic handler for preemption means that we must deny microservice code the ability to catch the panic and return to execution. Although we mitigate this possibility by detecting and blacklisting microservices that fail to yield under a SIGALRM, it would be desirable to block such behavior entirely. Possible options include using the dynamic linker to interpose stub implementations or linking against a custom build of the library, or using more in-depth static analysis.

Resource leaks. Safe Rust code provides memory safety, but it cannot prevent memory leaks [21]. For example, destructor invocation is not guaranteed using Rust's default reference counting-based reclamation; therefore, unwinding the stack during preemption is not guaranteed to free all of a microservice's memory or other resources. Potential solutions are interposing on the dynamic allocator to record tracking information (likely proving expensive) or using per-microservice heaps that main worker process can simply deallocate when terminating a microservice. The worker can also deallocate other resources, such as unclosed file descriptors. If these checks end up being too expensive, the worker could execute its cleanup after a certain number of microservices have run or when the load is sufficiently low.

Side channels. Our current approach is vulnerable to side-channel attacks [17, 13]. For example, microservices have access to the memory addresses and timings of dynamic memory allocations, as well as the numbers of opened file descriptors. Although side-channels exist in many systems, the short duration of microservice functions may make mounting such attacks more challenging; nevertheless, standard preventative practices found in the literature should apply.

Despite the security challenges of running microservice as functions, worker processes are still well-isolated from the rest of the system. Worst case, the central dispatcher process can restart a failed worker and automatically ban suspect microservices.

7 Conclusion

In order to permit applications to fully leverage the 10s of μs latencies available from the latest datacenter networks, we propose a novel design for serverless platforms that runs user-submitted microservices within shared processes. This structure is possible because of language-based *compile-time memory safety guarantees* and *microsecond-scale preemption*. Our implementation and experiments demonstrate that these goals of high throughput, low invocation latency, and rapid preemption are achievable on today's commodity systems, while potentially supporting hundreds of thousands of concurrently available microservices on each compute node. We believe that these two building blocks will enable new FaaS platforms that can deliver single-digit microsecond invocation latencies for lightweight, short-lived tasks.

Acknowledgements

This work was supported by the U.S. National Science Foundation under award CCF-1535821 and the Intel Science and Technology Center for Visual Cloud Systems.

References

[1] Erlang programming language. https://www.erlang.org, 2018.

[2] The Go programming language. https://golang.org, 2018.

[3] The Rust programming language. https://www.rust-lang.org, 2018.

[4] Amazon. AWS Lambda. https://aws.amazon.com/lambda.

[5] Apache Software Foundation. OpenWhisk. https://openwhisk.apache.org.

[6] Brad Campbell. Crates.io ecosystem not ready for embedded Rust. https://www.tockos.org/blog/2017/crates-are-not-safe.

[7] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, Feb. 2013.

[8] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *Proc. 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, Oct. 2015.

[9] D. Firestone et al. Azure accelerated networking: SmartNICs in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, Renton, WA, Apr. 2018.

[10] Google. Cloud Functions. https://cloud.google.com/functions.

[11] V. Jalaparti, P. Bodik, S. Kandula, I. Menache, M. Rybalkin, and C. Yan. Speeding up distributed request-response workflows. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, 2013.

[12] M. Jeon, Y. He, H. Kim, S. Elnikety, S. Rixner, and A. L. Cox. TPC: Target-driven parallelism combining prediction and correction to reduce tail latency in interactive services. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[13] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. *ArXiv e-prints*, Jan. 2018.

[14] B. W. Lampson. Hints for computer system design. In *Proceedings of the ninth ACM symposium on operating systems principles*, SOSP '83, pages 33–48. ACM, 1983.

[15] A. Levy, B. Campbell, B. Ghena, D. B. Giffin, P. Pannuto, P. Dutta, and P. Levis. Multiprogramming a 64 kB computer safely and efficiently. In *Proceedings of the 26th ACM symposium on operating systems principles*, SOSP '17, pages 234–251. ACM, 2017.

[16] J. Li, N. K. Sharma, D. R. K. Ports, and S. D. Gribble. Tales of the tail: Hardware, OS, and application-level sources of tail latency. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2014.

[17] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown. *ArXiv e-prints*, Jan. 2018.

[18] Microsoft. Azure Functions. https://azure.microsoft.com/services/functions.

[19] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker. NetBricks: Taking the V out of NFV. In *Proc. 12th USENIX OSDI*, Savannah, GA, Nov. 2016.

[20] seccomp(). seccomp(2) manual page from Linux man-pages project, Nov. 2017.

[21] The Rust Reference. Behavior not considered unsafe. https://doc.rust-lang.org/stable/reference/behavior-not-considered-unsafe.html, 2018.

[22] The Rust Reference. Behavior considered undefined. https://doc.rust-lang.org/stable/reference/behavior-considered-undefined.html, 2018.

[23] A. Vahldiek-Oberwagner, E. Elnikety, D. Garg, and P. Druschel. Erim: Secure and efficient in-process isolation with memory protection keys. *arXiv preprint arXiv:1801.06822*, 2018.

[24] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey. Bobtail: Avoiding long tails in the cloud. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2013.