Groundhog: Efficient Request Isolation in FaaS

Mohamed Alzayat, Jonathan Mace, Peter Druschel, and Deepak Garg
Max Planck Institute for Software Systems (MPI-SWS)
Saarland Informatics Campus, Saarbrücken, Germany

Abstract

Security is a core responsibility for Function-as-a-Service (FaaS) providers. The prevailing approach isolates concurrent executions of functions in separate containers. However, successive invocations of the same function commonly reuse the runtime state of a previous invocation in order to avoid container cold-start delays. Although efficient, this container reuse has security implications for functions that are invoked on behalf of differently privileged users or administrative domains: bugs in a function's implementation — or a third-party library/runtime it depends on — may leak private data from one invocation of the function to a subsequent one.

Groundhog isolates sequential invocations of a function by efficiently reverting to a clean state, free from any private data, after each invocation. The system exploits two properties of typical FaaS platforms: each container executes at most one function at a time and legitimate functions do not retain state across invocations. This enables Groundhog to efficiently snapshot and restore function state between invocations in a manner that is independent of the programming language/runtime and does not require any changes to existing functions, libraries, language runtimes, or OS kernels. We describe the design and implementation of Groundhog and its integration with OpenWhisk, a popular production-grade open-source FaaS framework. On three existing benchmark suites, Groundhog isolates sequential invocations with modest overhead on end-to-end latency (median: 1.5%, 95p: 7%) and throughput (median: 2.5%, 95p: 49.6%), relative to an insecure baseline that reuses the container and runtime state.

CCS Concepts: • **Security and privacy** → **Systems security**; **Software and application security**.

Keywords: Request isolation, Security, Snapshot, Checkpoint, Restore, Rollback, FaaS, Serverless

1 Introduction

Function-as-a-Service (FaaS) is an emerging high-level abstraction for cloud applications. Tenants state their application logic as a function implementation written in a high-level language like Python or JavaScript. The FaaS provider exports an HTTP/S endpoint, which can be used to invoke the function with arguments and receive results. The FaaS provider is responsible for deploying and executing the tenants' functions, provisioning and scaling resources as workload demand fluctuates, and maintaining and multiplexing the hardware and software infrastructure across different tenants and functions. FaaS has an 'on-demand' charge model: a tenant only pays for the compute time used to execute their functions.

Among the core responsibilities of a FaaS provider is security. For scalability and efficiency, FaaS platforms multiplex functions of different tenants concurrently on a large pool of shared resources. FaaS platforms rely on various language-, process-, and VMM-based isolation mechanisms to isolate functions from one another: each function executes within its own execution environment and *different functions* do not share the same execution environment. *Container isolation* is a commonly-used, general, low-entry-barrier function isolation mechanism that relies on standard OS process-isolation primitives. An alternative to container isolation is VMM-based isolation, where each function executes in a separate VM. Both container and VMM-based isolation prevent a malicious or compromised function from affecting the availability, integrity, and confidentiality of other function instances.

So far, the focus of security in FaaS has been on isolating *different function instances* from one another. Ideally, a FaaS platform should provide the same degree of isolation among *sequential activations* of the *same function instance*; otherwise, bugs in a function implementation, or a third-party library/runtime it depends on, may cause a leak of information from one activation of a function to a later one. This *sequential request isolation* is critical if a function can be invoked by, or on behalf of, differently privileged callers, such as clients of a tenant's service built on top of the function. For example, if the same function container is first invoked to service Alice's request and then invoked again to service Bob's request, there is a possibility that a bug in the function, some library, or the language runtime causes Alice's data from the first request to be retained and later leaked into the response to Bob.

Such leaks may arise despite the fact that FaaS functions are normally written in memory-safe, high-level languages like JavaScript or Python. First, functions written in such languages may still contain logical bugs that leak data. Second, high-level languages rely on libraries (e.g. NumPy, PyTorch, and TensorFlow for Python) that are written in unsafe languages like C/C++ for efficiency. In fact, the preliminary OWASP report of the Top 10 serverless security risks [47] mentions "insecure shared space" not cleared between sequential uses of a container as a risk to consider. An attack exploiting the `tmp` filesystem, which is an instance of insecure shared storage in FaaS, was demonstrated in [48]. More generally, unsecured shared space in FaaS can in principle enable attacks similar to those known from traditional server environments, such as the infamous Heartbleed bug [1], the Cloudbleed bug [32], and many others affecting various programming languages, frameworks, and libraries [2, 12, 13, 30, 63, 65].

To provide request isolation, FaaS providers like AWS lambda suggest partitioning clients from different security domains by redirecting them to distinctly named functions [15]. This approach requires code duplication and does not scale to services that have many mutually distrusting clients, as is commonly the case in e-commerce, for instance. A trivial way to ensure sequential request isolation would be for the provider to execute *every activation of a function* in a freshly initialized container. However, this solution is problematic from the perspective of performance: Container initialization overheads are high, ranging from a few seconds when done naively to hundreds of milliseconds with existing solutions that reduce the cost of container cold-starts [8, 16, 23, 27, 44, 54, 57, 62], which is higher than the basic execution time of a significant fraction of FaaS functions. For example, function execution times in Microsoft Azure were reported to have a median of 900 ms and a 25th %-ile of 100 ms [51] and we observe even lower execution times in our experimental setup. Hence, this trivial solution would impose impractical overhead. A more efficient solution would be to fork a copy of a fully initialized process for each function invocation and discard the copy once it terminates. Unfortunately, fork does not work for multi-threaded functions or language runtimes [34].

This paper presents Groundhog, a lightweight sequential request isolation system for container-based FaaS frameworks.[1] Importantly, Groundhog reuses containers across requests to the same function, thus avoiding the per-activation container re-initialization cost of the trivial solution above. Groundhog is independent of the language, runtime, or libraries used to implement functions, does not require changes to function implementations, OS kernels or hypervisors, and preserves most of the performance benefits of container reuse. To the best of our knowledge, Groundhog is the first system to do so.

Groundhog exploits two properties of FaaS platforms to provide a general-purpose, lightweight, performant solution: (1) At most one function activation executes at any time in a container; and (2) functions are not expected to retain runtime state across activations. Accordingly, the core of Groundhog's sequential request isolation is a general, in-memory, *lightweight process snapshot/restore* mechanism. Groundhog encapsulates each function in a (containerized) process, and takes a snapshot of each function process' fully initialized state just before the function is invoked for the first time. While this state typically includes a fully initialized language runtime, possibly with multiple threads, the function has not yet received activation-specific arguments or credentials, and its state is therefore guaranteed to be free of secrets. Subsequently, whenever the function has finished an activation and returned its results, Groundhog restores the function's process to the clean state recorded in the snapshot.

Groundhog is secure because the restoration ensures that no data can leak from one activation to a subsequent one. Groundhog is efficient because the cost of restoring state is roughly proportional to the amount of memory modified during an activation. As we will show, most function activations modify only a small proportion of the function process' total state. Finally, Groundhog restores state *between activations of a function*, and therefore does not contribute much to a function's activation latency under low to medium server load.

We have implemented Groundhog in C using commodity Linux kernel facilities. We evaluate Groundhog in OpenWhisk using Python, Node.js, and C functions from the FaaSProfiler [50], pyperformance [60], and PolyBench/C [40] benchmarks, which cover a wide variety of use cases. We demonstrate that Groundhog achieves sequential request isolation with modest overhead on end-to-end latency (median: 1.5%, 95p: 7%) and throughput (median: 2.5%, 95p: 49.6%) relative to an insecure baseline that reuses containers and runtimes. The main contributions of this paper include:

1. The design of a language- and runtime-independent, in-memory lightweight process snapshot/restore mechanism for general-purpose sequential request isolation in FaaS while retaining the performance benefits of container reuse.

2. The design and implementation of Groundhog,[2] a system that provides lightweight sequential request isolation on commodity Linux kernels and its integration into the OpenWhisk FaaS platform. Groundhog can be retrofitted to existing commercial systems without any changes to existing functions, libraries, language runtimes, and OS kernels.

3. An experimental evaluation of Groundhog on functions from the FaaSProfiler [50], pyperformance [60], and PolyBench/C [40] benchmarks within the OpenWhisk FaaS platform, which demonstrates that Groundhog provides sequential request isolation with low to modest overhead on function latency and peak throughput.

---

[1] We describe our work in the context of FaaS platforms that use containers to isolate different functions from each other, but similar design principles should apply to VMM-based isolation as Groundhog operates on the OS-process level.

[2] Groundhog is open-source and is available at [5].

2 Background

Functions and Requests   In the Function-as-a-Service (FaaS) model, tenants upload *functions* for execution by the cloud provider. A function is usually written in a high-level language, accepts input arguments, and returns results. The FaaS platform exposes an HTTP/S endpoint to which the tenant's applications can send *requests* with arguments, and receive results in response.

Containers and Function Invocation   The FaaS provider is responsible for executing functions on demand. FaaS platforms execute functions within *containers*, which may take the form of language-enforced compartments [20, 25, 52], processes [4], OS containers [9, 26, 41, 45], or VMs [3, 6]. When a request arrives for a particular function, an instance of the container needs to execute in order to serve the request. The FaaS platform may either instantiate a new container instance for the requested function from scratch—a *cold-start*—or *reuse* an existing idle container instance for the function if one exists.

FaaS platform services   Many FaaS platforms offer tenant's function implementations access to platform services. These services include storage, such as file access to scratch storage on a local disk, persistent key-value stores, or full relational database backends. Platform services may also provide automatic invocation of tenant's functions triggered by timers, writes to certain key-value tuples, or updates to certain rows in a database.

Access control   FaaS providers support client authentication on HTTP/S endpoints and minimally check if a caller is authorized to invoke the function, based on an access control list provided by the tenant. Access to platform services by the function is controlled in this case on a per-tenant basis. Some FaaS providers like AWS-lambda, Azure FunctionApps, Google Cloud Functions, and IBM Cloud Functions [6, 26, 29, 41] associate more fine-grained, per-caller[3] credentials to a function activation. Here, activations of the same function can have access to different platform services depending on the caller. Tenants can use this facility to control information flow via platform services among differently privileged callers of the same function, such as the different end-users of a tenant's deployed application.

Security   Security is a chief concern – beyond per-tenant or per-caller access control to functions and platform services, FaaS platforms must prevent a buggy or malicious function from compromising other functions or obtaining unauthorized access to platform services. Containers are the key design choice for achieving such *function isolation*. Each container instance executes a single function. Moreover, a container may be reused for repeated invocations of the same function.

Sequential request isolation   Our focus in this work is *sequential request isolation*, which isolates repeated invocations of the same function within the same container from each other. This isolation is important because bugs or compromises in a function implementation, or a third-party library or runtime that the function relies on, can cause confidentiality breaches by either (i) exposing private arguments of an activation to a subsequent activation of the function, or (ii) using the credentials of an activation to obtain information from platform services and leak them to a subsequent, less privileged activation. In addition, such leaks can lead to integrity breaches, e.g., when an activation uses credentials leaked from a previous, more privileged activation.

A trivial method of sequential request isolation is to start each request in a freshly-initialized container (forcing a cold-start on each request). However, container initialization is expensive, as is well-known from studies on FaaS cold start latencies. Despite excellent progress on reducing container initialization costs [8, 23, 57, 62], black-box techniques could still add hundreds of milliseconds of overhead on the critical execution path relative to standard insecure warm-container reuse. This overhead is of the same order of magnitude as a significant fraction of FaaS functions.

Consequently, we seek a different request isolation technique for FaaS that does not rely on container cold-start on every request and has low overhead relative to an insecure baseline that provides no isolation between sequential requests to the same function. Our solution adds only a few milliseconds of overhead off the critical path of a request (median: 3.7 ms, 95p: 16.1 ms) and a minimal overhead for tracking modifications on the critical path (median: 1.5%, 95p: 7%) relative to an insecure baseline that does not provide sequential request isolation. We aim for a practical technique and particularly target a *black-box* approach that can be applied directly to functions independent of language or FaaS platform.

3 Design Preliminaries

Groundhog operates at the level of OS processes. It can be readily integrated into FaaS platforms that encapsulate language runtimes and functions in processes or containers, which includes most major FaaS platforms currently in production use as far as we know. Moreover, Groundhog places no restrictions on function implementations or the programming language, runtime, or third-party libraries they rely on. Groundhog transparently interposes on API calls between a function implementation and the FaaS platform. Function implementations as well as the existing FaaS platforms can remain unchanged.

By interposing on a function's API calls, Groundhog detects when the function is invoked and when its execution finishes and returns results. Groundhog uses this information to transparently create an initial snapshot of a newly created process before its first invocation and reverts its state after it has finished executing an invocation. For this purpose, Groundhog relies on a custom in-memory process snapshot/restore facility. The facility relies on standard Linux functionality, such as soft-dirty bits to track modified pages, the /proc filesystem to

---

[3]In this paper, the caller is the entity causally responsible for the activation of a function.
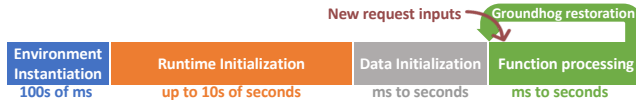
Fig. 1. *Groundhog container life cycle*

monitor changes to the process' address space mappings and read/write process memory, and `ptrace` to orchestrate state snapshot and restore.

Fig. 1 illustrates a function process's life cycle when Groundhog is being used. Groundhog avoids container, runtime, and data initialization steps when reusing a function container (process), and reverts the process' state in a median of 3.7 ms (10p: 0.7 ms, 25p: 1 ms, 75p: 5.4 ms, 90p: 13 ms). From the perspective of the FaaS platform, the Groundhog-enabled container enjoys the benefits of container reuse, while ensuring sequential request isolation irrespective of bugs in a function's implementation, libraries, or runtime.

3.1 Insights

Groundhog relies on two key properties of the FaaS model as implemented by major FaaS platforms, and one observation about typical FaaS functions. We start with the two properties of FaaS platforms.

One-at-a-time function execution In FaaS platforms, each function container executes at most one request at a time. For scalable throughput, platforms create separate containers to concurrently execute activations of different functions or multiple activations of the same function.

Stateless functions In the FaaS programming model, a function implementation cannot expect that its internal state is retained across activations. To maintain persistent state, functions must instead rely on external or platform services such as a key-value store or a database backend [14].

Some FaaS platforms support *global state* to enable performance optimizations. This state can be initialized using tenant-supplied code that is executed once a function container is initialized. Such an initialization step serves as a mechanism to pre-compute or cache data and state that can be utilized by several subsequent function activations independent of their inputs (e.g., populating data structures, downloading machine learning models). This state is retained across invocations as long as the container is reused, but is lost when the FaaS platform shuts down a container. Since the platform is free to shut down a container at any time, functions must not rely on the persistence of such global state for correctness.

The statelessness requirement implies that simple statistics counters or the internal state of a PRNG, for instance, must not be assumed to persist across invocations of a function; function implementations should use explicit platform facilities for persistent state and PRNGs instead. Data loss and/or functionality anomalies may arise if a function implementation or a library it depends on relies on internal or global state being maintained across invocations, because the FaaS runtime may destroy and refresh a container between invocations.

The one-at-a-time and statelessness properties afford FaaS providers a high degree of flexibility in placing, scheduling, and dynamically replicating function activations. In the context of Groundhog, these properties imply that each reused container has well-defined points in its life cycle—namely between sequential activations—when its state can be safely restored to a point after the initialization of global state but before the first function activation, thereby ensuring sequential request isolation.

Additionally, Groundhog relies on the following observation about typical FaaS functions for its efficiency.

Small write sets FaaS functions, particularly those written in managed languages, often use a substantial amount of memory, but only a small proportion of it is modified during an activation. This improves Groundhog's efficiency because only modified parts of memory need to be restored after an activation. Our empirical evaluation on 58 benchmarks shows that the number of pages actually modified by each function invocation is only a small fraction of the overall function memory (mean: 8.5% of the mapped address space is modified, median: 3.3%, 90p: 17%). A similar observation was reported by REAP [57], where the examined functions' working sets (i.e., modified pages and pages that were only read) were on average 9% of their memory footprints. Full measurement data for our benchmarks can be found in [5].

Groundhog's design and implementation, which is discussed in §4, were guided by these key insights.

3.2 Design options

Besides the trivial solution of using a fresh container for every request, which is inefficient, there are three broad design approaches for efficient sequential request isolation.

Language-based approaches When using appropriate safe programming languages [17], compiler instrumentation techniques [61], or runtimes [64] to implement functions, the language semantics can ensure efficient request isolation. However, this approach requires all tenants to use a particular (set of) programming languages/compilers, prevents the use of libraries written in unsafe languages for efficiency, and is vulnerable to bugs in the language runtime.

Fork A simple process-based technique is to `fork` a fully initialized function process, execute an activation within the child process, and discard the child process after the activation finishes. The main limitation of this approach is that `fork` as implemented in general-purpose operating systems cannot capture the state of a multi-threaded process. To take full advantage of container reuse, we need to be able to snapshot the fully initialized runtime of a managed language like JavaScript, which typically includes multiple active threads. Additionally, `fork` (or any copy-on-write (CoW) based approach) incurs expensive data-copying page faults during the execution of the function (i.e., on the critical path of a request).

Custom snapshot/restore facilities have been explored in prior work [8, 23, 33, 53, 54, 57] to reduce container cold-start costs by snapshotting an initialized runtime to disk/memory, and restoring it when a new container is needed. In principle, this approach could be used to instantiate a container for each activation. While substantially better than a cold-start for each activation, instantiating a container from a snapshot is still too expensive when compared to container reuse for many functions in our benchmarks.

3.3 Threat model

The FaaS platform, including the platform software, OS kernels, hypervisors, and platform services are trusted. We assume that the platform authenticates clients who connect to HTTP/S endpoints, and enforces access control to functions, as well as a function activation's access to platform services according to the authenticated client's credentials.

Legitimate tenants are expected to set up access control lists that allow only legitimate parties to invoke their functions, and prevent unwanted information flow via platform services among legitimate callers with different privileges.

Function implementations provided by tenants, including any libraries they link and the language runtimes they rely on, are untrusted and may contain bugs that retain credentials or sensitive data in the function's memory (e.g., a payment-processing/invoice-preparation function may retain credit card information) from one client request and leak it to a later request from a different client.

Under these assumptions, Groundhog prevents leaks of information from a function activation to subsequent ones, while allowing container reuse. Side-channels are out of scope.

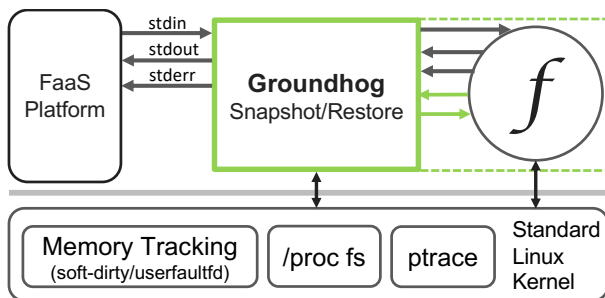## 4 Groundhog Design and Implementation

Fig. 2. *Groundhog Architecture: (1) The manager (solid green box), (2) The function process. Groundhog relies on standard Linux kernel utilities.*

Fig. 2 illustrates the Groundhog architecture. The Groundhog *manager* process (solid green box) runs within an OS container alongside the function process, and is responsible for enforcing request isolation. Groundhog uses a novel, light-weight, in-memory process snapshot/restore facility that achieves low restore times. The facility relies on standard Linux kernel features to snapshot, track, and restore processes. The design was guided by the following goals:

Generality The facility operates on a generic, multi-threaded POSIX process and makes minimal assumptions about the code (function) executing inside the process. Groundhog can be used on an opt-in basis: each tenant can decide whether to enforce sequential request isolation for their functions.

Restore cost proportional to modified pages To take advantage of the fact that most function activations only modify a small proportion of the function process' state, Groundhog tracks pages modified during a function activation using the Linux soft-dirty bits tracking facility. As a result, Groundhog need only restore modified pages after an activation.

Restore cost off the critical function execution path FaaS platform servers, and production servers in general, are less than fully utilized most of the time. Therefore, the design of Groundhog's snapshot/restore facility seeks to minimize overhead *during* a function's execution, in favor of performing all restore-related tasks *between* function activations. Groundhog performs two main operations. First, Groundhog takes an in-memory snapshot of the function process after a container is created. This operation contributes only to the cold-start latency. Second, Groundhog restores the memory layout and content to the snapshotted state after a function invocation completes. Groundhog avoids copy-on-write and the associated expensive data-copying page faults and it does not intercept memory-layout-modifying syscalls during a function's execution. Instead, Groundhog identifies and reverts changes in the memory layout by diffing the memory layout, and restores the content of modified pages as indicated by Linux's soft-dirty bits; it performs these actions during a restore operation, after a function invocation completes and has returned its result to the invoker. Hence, Groundhog performs expensive operations between function invocations, minimizing overhead during function execution.

4.1 Container initialization

The Groundhog manager process interposes between the FaaS platform and the process executing the function. The FaaS platform initializes the Groundhog manager process as if it were the process executing the function. The Groundhog manager then receives requests from the FaaS platform, relays them to the function process, and communicates results back to the FaaS platform. It communicates with the FaaS platform using the latter's standard communication channels (in OpenWhisk— the platform on which our prototype runs—these are usually stdin and stdout.)

To initialize the actual function process, Groundhog forks a new process, prepares pipes for communicating with it, drops privileges of the child process, and execs the actual function runtime in the child process.

Next, Groundhog creates a snapshot of the function process. As a performance optimization, before taking the snapshot, Groundhog invokes the function with *dummy* arguments that are independent of any client secrets. These dummy arguments

can be provided by the function deployer, once for every function they deploy, and can be part of the function's configuration. After the function returns, Groundhog snapshots the state of the function process as described in §4.2. After this snapshot is created, Groundhog informs the platform that it is ready to receive actual function invocation requests.

The purpose of the dummy invocation is to trigger lazy paging, lazy class loading, and any application-level initialization of global state, and to capture these in the snapshot. Snapshotting without a dummy request would cause these (expensive) operations to happen again after every state restoration, which would increase the latency of subsequent function activations. This is particularly relevant when the function runs in an interpreted runtime like Python or Node.js, which may heavily rely on lazy loading of classes and libraries [38]. We note that the arguments provided to a dummy invocation may affect performance but not security.

4.2 Snapshotting the function process

To take a snapshot, the manager interrupts the function process, then (a) stores the CPU state of all threads using `ptrace` [35]; (b) scans the `/proc` file system to collect the memory mapped regions, memory metadata, and the data of all mapped memory pages; (c) stores all of this in the memory of the manager process; and (d) resets the soft-dirty bits memory tracking state. Finally, the manager resumes the function process, which then waits for the first request inputs. After the request is completed, Groundhog restores the function's process state back to this snapshot before accepting a new request.

4.3 Tracking state modifications

Groundhog uses the standard *Soft-Dirty Bits* (SD) feature of the Linux kernel [36],[4] which provides a page-granular, light-weight approach to tracking memory modifications. Each page has an associated bit (in the kernel), initially set to 0, that is set to 1 if the page is modified (dirtied). When a function invocation completes, Groundhog scans the SD-bits exposed by the Linux /proc filesystem to identify the modified pages. After restoring the function process, Groundhog resets all SD-bits to 0, ready for the next invocation.

We considered using Linux's user-space fault tracking file descriptor (UFFD) [37][5] feature for memory tracking and prototyped this alternative; however we found UFFD to have significantly higher overhead compared to SD-bits due to the frequent context switches to user-space for fault handling.[6] UFFD was marginally faster than soft-dirty bits only when the number of dirtied pages was close to zero.

---

[4]Available on stock Linux kernels v3.11+. We identified and reported a bug that affected the accuracy of the SD-Bits memory tracking in v5.6; the bug was fixed in v5.12 [42].

[5]Write protection notifications available on stock Linux kernels v5.7+.

[6]A custom in-kernel facility that allows an application to request a list of modified pages presumably could be much faster, but would require kernel modifications.

4.4 Restoring to the snapshotted state

When a function invocation completes, the function process returns the result to the Groundhog manager. Groundhog's manager awaits the function response and forwards it to the FaaS platform (which then sends it to the caller). Next, the manager interrupts the function process and begins a restore. The manager identifies all changes to the memory layout by consulting `/proc/pid/maps` and `pagemap` (e.g. grown, shrunk, merged, split, deleted, new memory regions); these changes are later reversed by injecting syscalls using `ptrace` [24, 35, 54]. The manager restores `brk`, removes added memory regions, remaps removed memory regions, zeroes the stack, restores memory contents of pages that have their SD-bit set, madvises newly paged pages, resets SD-bits, and finally restores registers of all threads.

After restoration completes, the child process is in an identical state to when it was snapshotted, and the process is ready to execute the next request.

There are multiple optimizations that can be applied at the platform level. For instance, if a function is invoked consecutively by mutually trusting callers, then the FaaS platform can route the invocations to the same function instance and instruct Groundhog to skip the rollback between such invocations. Similarly, if the system is under heavy load due to invocations of different functions, then the FaaS platform may quarantine the containers and instruct Groundhog to defer the restoration.

4.5 Enforcing request isolation

Groundhog enforces request isolation by design. Groundhog prevents new requests from reaching the function's process until it has been restored to a state free from any data of previous requests. This is achieved by intercepting the end-client requests before they reach the function and buffering them in Groundhog until the function's process has been restored.

Although intercepting the communication ensures control of the function process and enforces security, it can add an overhead of copying request input/outputs to and from Groundhog's manager process. This overhead can be eliminated as follows: (1) The FaaS platform can forward inputs directly to the function process after waiting for a signal from Groundhog's manager process that the function has been restored to a clean state. This requires minor changes to the FaaS platform to wait for the signal from Groundhog. (2) Upon completion of a request, the function process can return outputs directly to the FaaS platform and, separately, signal Groundhog's manager process that its state can be rolled back. The changes needed can be made in the I/O library that handles communication with the platform in the function process (no changes are needed to the code of the individual functions submitted by the developers).[7]

---

[7]We implemented (2) to facilitate debugging. Our evaluation still intercepts all inputs and outputs to demonstrate that platform modifications were not required and show the overhead of such interception on various functions.

Assumptions: Groundhog relies on some standard Linux kernel facilities that must not be blocked by the provider, namely the `ptrace` system call, the `/proc` file system, and the soft-dirty bits tracking. Groundhog expects that function implementations do not open network connections and files directly. (None of the benchmarks we use in the evaluation require them.) Instead, functions are expected to rely on platform services for network communication, storage, and for maintaining any persistent state.

Groundhog 's design is generic and agnostic to the function logic. However, our prototype currently does not support functions that fork child processes.[8] In principle, Groundhog could be extended to intercept fork syscalls and track the child processes as well through standard ptrace tracking options. Similarly, syscalls (such as seccomp) that can limit the availability of standard Linux kernel facilities required by Groundhog can be intercepted and adjusted to remain permissive enough through standard ptrace options.

Finally, as stated in our threat model, any external state (e.g. external storage, or the state of network connections and pipe contents) is assumed to be subject to access control. This is necessary to prevent data leaks across clients with different privileges via the external state.

5 Evaluation

In this section we evaluate Groundhog's performance on a range of FaaS benchmarks. Overall, we show that:

- For a wide range of benchmark functions using three different languages/runtimes, Groundhog has modest overhead on end-to-end latency and throughput.
- Groundhog's latency overhead depends primarily on the memory characteristics of the function and is proportional to the number of pages dirtied during a function's execution. Groundhog's throughput scales nearly linearly with the number of available cores.
- Groundhog's lightweight restoration has equivalent or better performance than a strawman fork-based isolation approach, which is less general. We also compare to a WebAssembly-based isolation approach, and show that Groundhog has competitive performance despite being more general.

5.1 Evaluation Overview

Implementation. We implemented Groundhog in ~6K lines of C. Groundhog is compatible with off-the-shelf Linux and requires no kernel changes.

OpenWhisk Integration. We integrated Groundhog with OpenWhisk [46] by modifying OpenWhisk's container runtimes for Python and Node.js to include Groundhog. In addition, we implemented an OpenWhisk container runtime for native C, to enable the evaluation of native C FaaS benchmarks. Most OpenWhisk runtimes use the actionloop-proxy design,

where a distinct process acts as a proxy that communicates with the OpenWhisk platform (through HTTP connections), and forwards the requests to the runtime process (through stdin), which has a simple wrapper to process inputs, call the developer's function, and return results. Groundhog interposes between the proxy and the runtime, intercepting the stdin and stdout and forwards the stdin only when the function's process is restored to a clean state. OpenWhisk's container runtime for Node.js, on the other hand, is built using a single process that directly interacts with the platform and runs the function. We refactored it to an actionloop-proxy-like design to maintain a uniform Groundhog implementation that ensures security by blocking inputs until the function's process is restored to a clean state.[9]

For the FaaS OpenWhisk-python-runtime, we added to the FaaS-provider wrapper 15 Lines of Code (LoC) to signal the function's readiness for snapshotting and restoration to Groundhog as well as to collect timing measurements of the function handler from inside the process. One line was modified to run Groundhog instead of the runtime with the runtime command passed as an argument to Groundhog. The container image was modified to include Groundhog.

For the FaaS OpenWhisk-Node-runtime, we refactored the runtime to follow the unified proxy design, which required modifying 150 LoC. If we had chosen to run the un-refactored runtime under Groundhog, only 30 LoC (same logic as for the python runtime) would need to be added, in addition to a signaling mechanism with the platform as described in §4.5.

We implemented a new OpenWhisk-C-runtime; the baseline required 60 LoC and the Groundhog version required an additional 21 LoC (same logic as for the python runtime). An off-the-shelf cJSON parsing library (2.5K LoC) was also added.

In general, integrating Groundhog with a FaaS platform that forward requests to (and receives results from) the function process through file descriptors would require changes similar to ours for OpenWhisk. Integrating Groundhog with FaaS platforms that retrieve requests through an HTTP API can be done by modifying Groundhog to handle the request retrieval and response sending, and by updating the FaaS runtime to retrieve the request from and send the response to Groundhog. Alternatively, a signalling mechanism between Groundhog and the FaaS platform can be implemented as outlined in §4.5.

Hardware Configuration. We ran all experiments on a private cluster hosting OpenStack/Microstack (ussuri, r233). Each physical host has an Intel Xeon E5-2667 v2 2-socket, 8-cores/socket processor, 256GB RAM and a 1TB HDD.

OpenWhisk Deployment. We use the standard distributed Openwhisk deployment. Our distributed setup comprises 2VMs. One VM runs all OpenWhisk core components except for the

---

[8]We have not seen such a computational pattern in the FaaS paradigm; parallelism is typically achieved in FaaS through multiple function instances.

[9]Encapsulating the full process would require Groundhog to implement the platform API or have a small platform modification to allow blocking inputs until Groundhog signals to the platform that the function's process is being restored as described in 4.5.

invoker, which runs on a separate VM. The invoker is the component responsible for starting function containers locally and dispatching function requests to them; this is the component that interacts with the containers hosting Groundhog. We choose to isolate the invoker component in a separate VM to have more control over variables affecting the experiments.

Both VMs are placed on the same physical host to minimize network communication overhead, creating favorable baseline conditions. To reduce potential performance interference, we pin the two VMs to separate cores and ensure that their memory is allocated from the corresponding NUMA domain. VMs are configured with 64GB RAM and an experiment-dependent number of cores (SMT turned off). The VMs run Ubuntu 20.04 with a stock Linux kernel v5.4. OpenWhisk is configured to run all functions with a 2GB RAM limit and a 5 minute timeout.

Experiment Configurations. To evaluate Groundhog's overheads, we run two primary configurations: **BASE**, an insecure baseline using unmodified OpenWhisk that does not provide sequential request isolation (we prevent container cold-starts in our experiments to deliberately create an unfavorable but conservative baseline); and **GH**, which uses Groundhog on OpenWhisk to provide sequential isolation.

We also run a third configuration **GH$_{NOP}$**, which includes Groundhog but does not restore dirtied pages between consecutive invocations of the same function. This configuration represents an optimization for the case where consecutive requests are from the same security domain (through additional hints from the FaaS platform which can be implemented as described in §4.5). The configuration also helps delineate Groundhog's page tracking and restoration costs, which is the difference between the GH and GH$_{NOP}$ configurations.

Lastly, we compare Groundhog to two alternative approaches. In §5.2.3 and §5.3.2 we implement a fork-based request isolation method, **FORK**, which is applicable to single-threaded applications and runtimes only. In §5.3.3 we compare Groundhog to **FAASM**, a research FaaS platform designed to reduce cold-start latencies for WebAssembly-compatible functions. We detail these alternative approaches in the respective sections.

5.2 Microbenchmarks

In this experiment, we evaluate Groundhog's impact on request latency and how that impact varies with the memory size and the number of pages dirtied.[10] We evaluate both the *in-function* overheads that are on the critical path of function execution, and the *restoration overhead* which occurs off the critical path. We defer evaluating Groundhog's snapshotting overheads, which occur only once after a new container starts, to §5.5.

Microbenchmark. We implement a simple function in C that pre-allocates an address space of a fixed size. Each invocation (a) dirties a subset of the pages by writing a word to

---

[10]We also considered address space fragmentation (same overall address space size but a varying number of memory maps) as an independent variable, but found that it has no statistically significant impact on the overhead of GH or FORK.
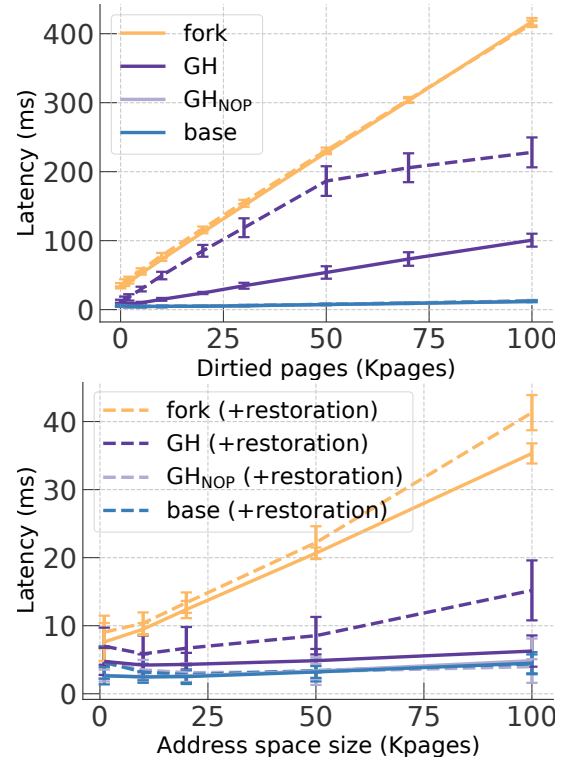

Fig. 3. *Latencies varying the number of dirtied pages (top) and the address space size (bottom). Different colors represent different request isolation methods (or no isolation for* BASE*). Solid lines are latencies with in-function overhead but not restoration overhead, while dashed lines include both. (The lines of* BASE *and* GH$_{NOP}$ *coincide visually in the figure.)*

each page of that subset, then (b) reads one word from each mapped page, even those that were not dirtied. We set up a 4-core VM with a single function-hosting container (this container is limited to 1 core), initialize the container, and then repeatedly invoke the function. We measure function latencies at the OpenWhisk invoker.

5.2.1 In-Function Overheads

Low-load Workload. We run the microbenchmark with a *low load* workload comprising 150 requests submitted one-at-a-time, with a small delay between consecutive requests. This delay is sufficient for Groundhog to complete restoration before the next request arrives, so measurements for the low-load workload capture only the in-function overheads.

Results. The solid lines in Fig. 3 (top) plot function latency as we vary the number of pages dirtied from 0 to 100K with a fixed 100K mapped pages. As expected, GH introduces some latency overhead proportional to the number of dirtied pages. This overhead is due to a minor page fault to set the soft-dirty (SD) bit when a page is dirtied, which is required by the SD-bit mechanism on our hardware. In contrast, GH$_{NOP}$ has negligible overhead relative to BASE since the SD-bits set in the first run are not reset (there is no memory restoration), and thus these page faults are not incurred in subsequent runs.

We also run a variant of the experiment where we fix the number of dirtied pages to 1K and vary the address space size from 1K to 100K pages. The solid lines in Fig. 3 (bottom) show the function latency as we vary the address space size. We observe that GH's overhead is constant with respect to address space size because the in-function overheads depend only on the number of dirtied pages, which is fixed now.

5.2.2 In-function + Restoration Overheads

High-Load Workload. We repeat the two experiments above with a *high load* workload comprising 150 requests submitted back-to-back with no delay between consecutive requests. This leads to additional delays while waiting for Groundhog to complete restoration after the previous request. In contrast to the low-load workload, the high-load workload thus reflects *both* the in-function and the off-critical-path restoration overheads.

Results. The dashed lines in Fig. 3 (top) show the function latency as we vary the number of pages dirtied from 0 to 100K with a fixed 100K mapped pages. We observe higher latency overheads for the high-load workload (dashed lines) compared to the low-load workload (solid lines), and these overheads grow linearly as the percentage of dirtied pages increases. There is a change in slope at 60K because Groundhog is able to coalesce individual page restorations into fewer, larger memory copy operations, which are more efficient.

Next, we repeat the second experiment variant. Fig. 3 (bottom) shows the function latency as we vary the address space size from 1K to 100K pages while fixing the number of dirtied pages to 1K. Although in-function overheads are constant, restoration overheads in this experiment increase linearly with the address space size, because during restoration Groundhog must scan the SD-bits of the whole mapped address space to determine the pages to restore.

5.2.3 Comparison to Fork

A potential alternative to our lightweight restoration is to use copy-on-write techniques such as fork (§3.2). Fork is not general purpose – it only works for single-threaded functions – however we provide a performance comparison for the purpose of illustration. We implement fork-based isolation and repeat the two microbenchmark experiments. In our fork-based implementation, we initialize the function up to the same point where Groundhog takes its snapshot (a safe clean state after a dummy request). Instead of lightweight restoration, each request is then handled by a separate copy of the process forked at that state.

Fig. 3 (top) shows the function latency of FORK as we vary the number of pages dirtied from 0 to 100K with a fixed 100K mapped pages. We observe that the overhead of FORK is higher than GH because each page fault is significantly more expensive than for GH, entailing an additional page copy.

Fig. 3 (bottom) shows the function latency of FORK as we vary the address space size from 1K to 100K pages while fixing the number of dirtied pages to 1K. We see significantly higher

overhead for FORK compared to Groundhog, and a linear increase in latency with the address space size. This increase is predominantly due to the additional overhead caused by dTLB misses on the first accesses to each page (even if unmodified) of the new process. This access can additionally require lazy creation of physical page table entries depending on the memory layout of the program.

5.3 FaaS Benchmarks

In this section we evaluate Groundhog's impact on request latency and throughput for a range of FaaS benchmarks written in three different languages. We first compare Groundhog to an insecure baseline in OpenWhisk (§5.3.1). We then provide an illustrative comparison to a fork-based implementation (§5.3.2) and to FAASM, an alternative WebAssembly-based FaaS platform designed to optimize cold-starts, but that can also be used for request isolation in limited cases (§5.3.3).

Benchmarks. We evaluate 58 functions across three benchmarks and three languages: 22 python functions from the pyperformance benchmark [60], 23 C functions from Poly-Bench/C [40], and 13 functions (6 python, 7 Node.js) from the FaaSProfiler benchmark suite [50].

These functions cover a wide variety of real FaaS use cases such as Web applications, JSON and HTML parsing/conversion, string encoding, data compression, image processing (2D, 3D), optical character recognition (OCR), sentiment analysis, matrix computations (e.g. multiplication, triangular solvers), and statistical computations.

Latency. To measure latency we deploy a 4-core VM with a single function container that is limited to at most one core, and run a closed-loop client in a separate VM on the same machine,[11] which submits requests one-at-a-time. This workload is similar to the *low-load* setting from §5.2.1 and enables Groundhog to complete restoration in between consecutive requests, so latency measurements reflect Groundhog's in-function overheads only. We report two latency measurements: the end-to-end latency of requests as experienced by the end-client (including all FaaS platform delays); and the invoker latency, which measures only the function execution time at the invoker, excluding overheads of the remaining FaaS platform components, which Groundhog does not affect at all. All measurements are averages of 1,200 invocations, except for C functions longer than 10 seconds, where we report averages of 90 invocations.

Measuring Throughput. To measure throughput we deploy a 4-core VM with 4 function containers in a separate VM that maintains a large number of in-flight requests (both the number of function containers and in-flight requests are chosen empirically to maximize throughput). This workload is similar to the *high-load* setting from §5.2.2 as it ensures the

---

[11]This placement minimizes network latencies to achieve best baseline performance and to allow easy and efficient scheduling of our 608 benchmark configurations on our resources.

FaaS platform is always saturated with requests. Throughput measurements thus account for Groundhog's full overheads including both the in-function overheads and the restoration overheads. Unless otherwise specified, we report the peak sustained throughput in 4 runs, each at least 1.5 minutes long.

Detailed Measurements. In addition to the figures presented in this section, full measurement data for our benchmarks can be found in [5]. Table 1 shows the absolute latency and throughput measurements for the BASE, GH, GH_{NOP}, FORK, and FAASM configurations. Table 2 shows the relative overheads compared to an insecure baseline. Table 3 shows the relation between the latency, overheads, and throughput of Groundhog.

5.3.1 Baseline Comparison

Fig. 4 (rel. E2E lat.) shows the end-to-end request latency for all benchmarks. For each benchmark, we normalize the latency measurements relative to BASE; thus values <1 indicate better latency than the base line and >1 represent worse latency.

We first consider the results for GH and GH_{NOP}. The main takeaway is that GH overhead on end-to-end latency relative to BASE is low overall. In most cases it is negligible (within one standard deviation). The median, 95th-percentile and maximum relative overheads are 1.5%, 7% and 54%, respectively, and the overhead is below 10.5% in all benchmarks except img-resize(n), where it is 54.2% (discussed in the next paragraphs). The low overhead in most benchmarks is unsurprising, because end-to-end latency measurements include delays within the FaaS platform that are significant relative to the overhead added by the SD-bit tracking. These significant platform overheads are the same in the baseline and Groundhog. Unless otherwise specified, GH_{NOP}'s performance is on par with that of BASE.

GH overheads are more apparent when we inspect invoker latencies. Fig. 4 (rel. inv. lat) plots the invocation latency for all benchmarks, normalized to BASE. We observe that for python and C benchmarks the Groundhog overhead is relatively low. However, for some specific Node.js benchmarks (Fig. 4 (FaaSProfiler (node)) the overhead is more pronounced, up to 70% in the worst case. This occurs for two reasons.

First, GH and GH_{NOP} proxy inputs to functions, which causes additional overheads for some of the Node.js functions with large inputs such as `json` and `img-resize` (which take inputs of 200kB and 76kB, respectively). This cost arises due to our refactoring of OpenWhisk's Node.js runtime wrapper. This overhead can be reduced by integrating Groundhog with the original single-process version of OpenWhisk Node.js.

Second, Node.js has a time-dependent behavior in garbage collection, namely, garbage collection can be triggered by the passage of time. Snapshotting and restoration can adversely affect this behavior, because restoration reverts the garbage collection state. The impact of this garbage collection was particularly pronounced on some benchmarks such as img-resize (n). The problem can be alleviated by virtualizing time such that the process restoration resets the time to the original time
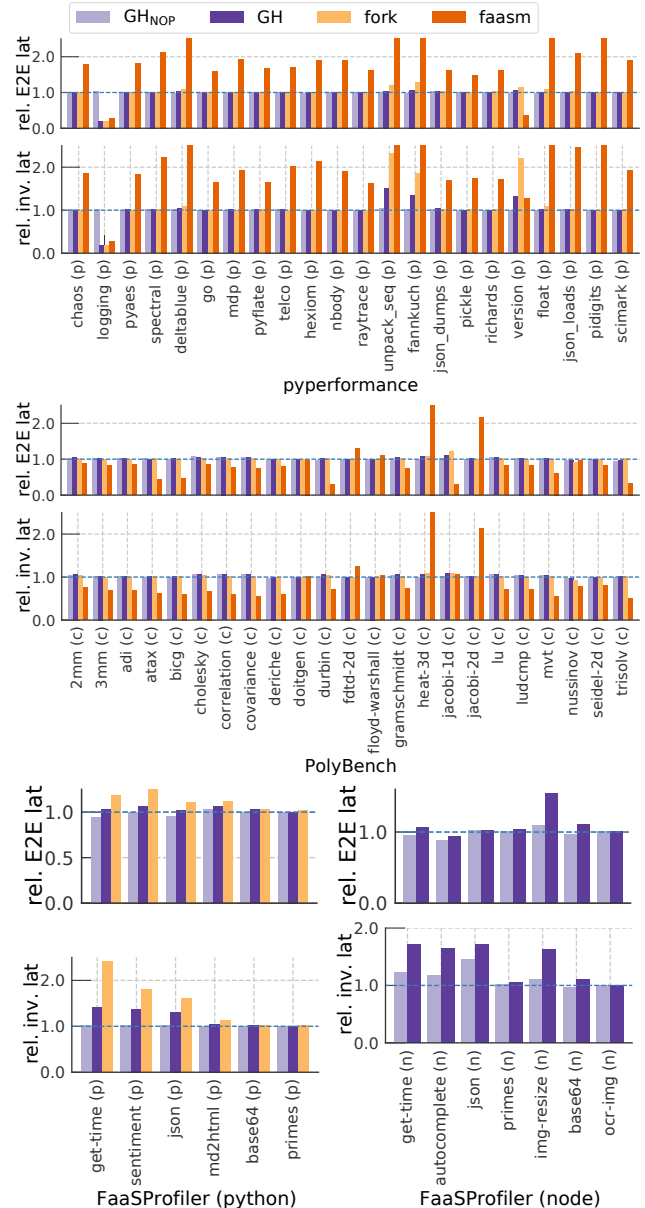

Fig. 4. *Relative end-to-end latency and invoker-measured latency of* GH, GH_{NOP}, FORK, *and* FAASM *compared to the insecure baseline* BASE. *Figures are capped at 2.5X the baseline. Detailed numbers are in [5]. The symbols (p), (c) and (n) denote Python, C and Node.js benchmarks, respectively.* **Lower numbers are better.**

of the snapshot, or by modifying the garbage collection to be time-independent. This is actually a broader problem in the space of snapshot and restore techniques; a comprehensive treatment of this topic is beyond the scope of this paper and left for future work.

Surprisingly, GH is faster than BASE on the pyperformance benchmark `logging (p)`. We discovered that this occurred due to a memory leak in the function's original implementation causing it to slow down with repeated invocations. GH's restoration rolls back the leaked memory, thus avoiding the slowdown.
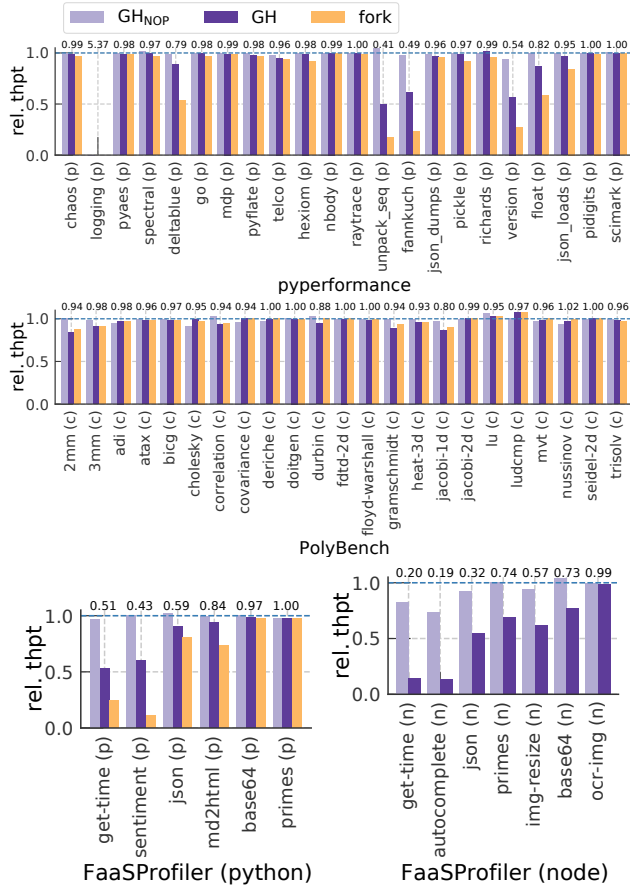
Fig. 5. *Relative throughput of* GH, GH$_{NOP}$, FORK *compared to the insecure baseline* BASE. *Detailed numbers are in [5].The symbols (p), (c) and (n) denote Python, C and Node.js benchmarks, respectively.* **Higher numbers are better.**

Fig. 5 shows the request throughput for all benchmarks, normalized to BASE. Since functions are invoked sequentially, the throughput of GH relative to BASE should be inversely proportional to GH's relative invoker overhead, which is roughly 1 + (in-function overhead + restoration overhead)/(baseline invoker latency). Our observations are consistent with this calculation: The throughput plots in Fig. 5 show the reciprocal of this calculation above each benchmark, and the heights of the GH bars are approximately equal to this value, as expected. For 40 out of 51 C/Python benchmarks the GH throughput is within 10% of BASE. It is up to 50% lower on the remaining, mostly very short benchmarks. On Node.js benchmarks, where GH's relative invoker latencies can be very high (as explained above), GH's throughput is between 2% and 86% less than BASE's. GH's Node.js restoration overheads tend to be higher than other runtimes as Node.js's runtime performs aggressive memory layout changes [12] (see Fig. 8 for the restoration overheads of selected benchmarks). Across all benchmarks, the median and 95th-percentile throughput reductions are 2.5% and 49.6%, respectively.

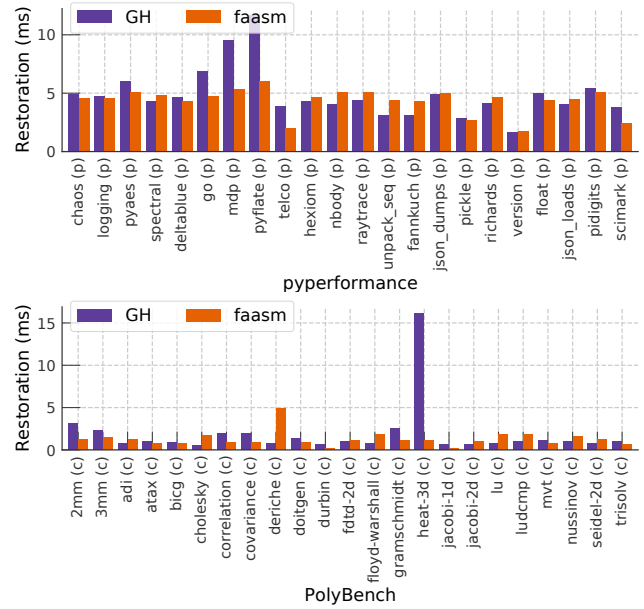[12]A less aggressive Node.js runtime would incur lower overheads.

Fig. 6. *Restoration duration (off the critical path) of* GH, *and* FAASM. *The symbols (p) and (c) denote Python and C.*

5.3.2 Comparison to Fork

We also provide a comparison to the FORK alternative described in §5.2.3. Recall that fork is only applicable to single-threaded functions, thus we are unable to provide measurements for the Node.js runtime.

Fig. 4 also plots results for FORK for single-threaded benchmarks. The latency overhead of GH is slightly less than that of FORK since GH's page faults are lighter than those of FORK (FORK's page faults also require page copying, while GH's page faults only set a SD-bit each).

Fig. 5 shows that the throughput of FORK follows a similar rule to that of GH. When compared to GH, FORK's throughput is similar on all but very short benchmarks, where GH's throughput is noticeably higher than FORK's.

5.3.3 Comparison to Request Isolation using Faasm

A potential alternative to Groundhog's process-based request isolation is to implement request isolation in the language runtime. To illustrate the performance trade-offs of the two approaches, we compare Groundhog to FAASM [52], a state-of-the art FaaS platform where functions are isolated from each other not using OS containers but by compiling them to WebAssembly, and relying on spatial isolation within WebAssembly's runtime. FAASM is designed to reduce FaaS cold-start latencies, but it can be used for efficient request isolation: WebAssembly limits each function to a *contiguous* 4GB memory map, which FAASM can quickly restore simply by a copy-on-write remapping after each request. Note that FAASM is not a fully general solution to the request isolation problem since it places restrictions on the functions – most notably, they must compile to WebAssembly.
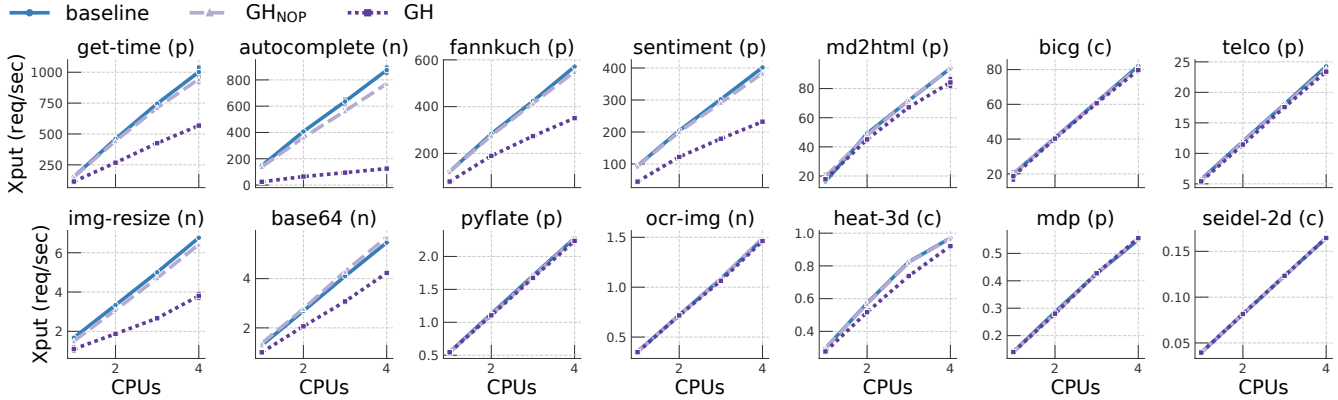
Fig. 7. *Throughput scaling with number of cores. Error bars (minute) represent the standard deviation across 6 runs.*

FAASM comes with its own FaaS platform, which is significantly different from OpenWhisk. Despite the differences in the platforms, which make a direct comparison difficult, we compare Groundhog and FAASM for completeness. For the comparison, we use the pyperformance and PolyBench/C benchmarks, both of which can be compiled to WebAssembly as demonstrated in [52]. We rely on FAASM's microbenchmarking infrastructure that reports both the overall latency (end-to-end and invoker) and the restoration (reset) cost.

Fig. 4 shows latencies for FAASM next to those for GH. On most pyperformance benchmarks, the latency of FAASM is considerably higher than that of GH, whereas the restoration time is comparable (Fig. 6). This is because the Python interpreter and runtime are less efficient when compiled to WebAssembly (which FAASM uses) compared to a natively compiled interpreter (which GH uses).

On PolyBench functions, FAASM's latencies are generally lower than those of GH. However, GH's poorer relative performance is not because of Groundhog's overheads. Rather, WebAssembly's runtime is specifically optimized for program patterns that occur in PolyBench, so WebAssembly compiled PolyBench outperforms natively compiled PolyBench even in the baseline. (This observation has been noted in prior work [28, 31, 52].)

The same trends continue to manifest in throughput measurements, where FAASM has lower throughput than GH on most pyperformance functions, and higher throughput than GH on most PolyBench functions. We omit the detailed throughput comparison here as it entangles many variables such as the differences in the platforms, which have nothing to do with request isolation, the platforms' internal components, runtimes (native vs WebAssembly), as well as the isolation mechanisms. The reader can find these numbers in [5].

Overall, the performance differences between FAASM and GH are dominated by differences between native and WebAssembly compilation rather than request isolation costs.

5.3.4 Throughput scaling with cores

We expect GH's throughput to scale linearly with cores as each

core can run a completely independent container instance with its own function and Groundhog copy. To confirm this, we repeat the throughput experiment above, varying the number of cores available to the VM from 1 to 4 (and an equal number of function container instances, each limited to 1 core). Fig. 7 shows absolute throughputs as a function of the number of available cores for a subset of 14 representative benchmarks of varying duration, number of mapped pages and number of dirtied pages. Reported numbers are sustained throughputs averaged over 6 runs of at least 1.5 minutes each (excluding a warm-up). Error bars are standard deviations (which were minimal) over the 6 runs. As expected, the scaling is nearly linear in all cases. We expect this nearly-linear trend to continue beyond 4 cores until a bottleneck in the kernel or memory buses arises.

5.4 Deconstructing restoration overheads

Groundhog restoration involves several steps that we outlined in §4.4. In this section we break down the cost of restoration for the same 14 representative benchmarks (selection criteria in §5.3.4). The overall restoration cost breaks down into the following components:

- ▇ interrupting the function process.
- ▇ reading the process' memory mapped regions
- ▇ scanning all mapped pages to identify which are dirtied
- ▇ diffing the memory layout to identify how it has changed
- ▇ restoring the original memory layout by injecting syscalls (brk, mmap, munmap, madvise, and mprotect)
- ▇ restoring the contents of modified and removed pages
- ▇ restoring registers
- ▇ resetting the soft-dirty bits of all modified pages
- ▇ detaching from the process

Each of these costs depends on different factors. The costs of interrupting, restoring registers, and detaching are functions of the number of threads in the process. The costs of reading, scanning, diffing the memory layout, and resetting soft-dirty bits are functions of the address space size and layout. The syscall injection cost depends on the number of memory layout changes and is heavily dependent on the language runtime.
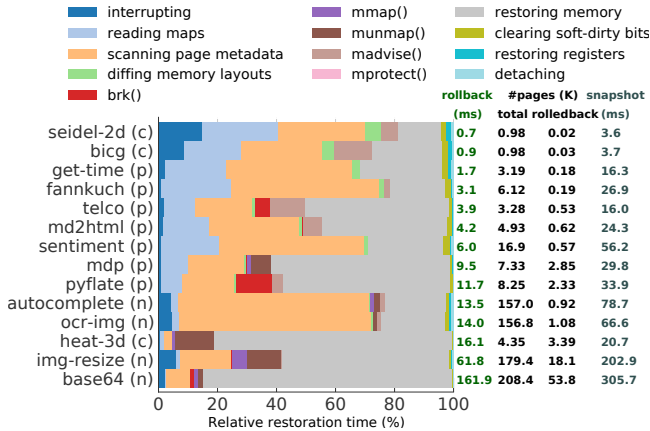
Fig. 8. *Restoration overhead (deconstructed) and the one-time snapshotting overhead for a subset of benchmarks.*

Lastly, the cost of restoring the contents of pages depends on the number of pages dirtied or unmapped during an invocation.

Fig. 8 shows these costs normalized to the total restoration cost for our 14 representative functions. For each benchmark we also detail the absolute restoration time, the number of pages, and the time for Groundhog to take its initial snapshot. (We revisit the snapshotting overhead in §5.5.) In particular, we note that the memory restoration cost (▨) is strongly correlated with the total number of pages restored. Similarly, the time spent scanning page metadata (▨) is strongly correlated with the total number of pages. (As discussed in §4.3, optimizations can make the costs correlate to the number of dirtied pages instead.)

5.5 Snapshotting overhead

The rightmost column of Fig. 8 outlines Groundhog's snapshotting latency overhead for the same 14 functions that we used in §5.3.4. Recall that snapshotting is a one-time operation that occurs upon container initialization. It involves pausing the process, copying the process's state to Groundhog's manager process memory, and resuming the process. Snapshotting requires scanning the memory layout of the process and copying its memory. The time and memory costs are primarily proportional to the total number of paged memory pages. The snapshotting latency overhead can be alleviated using techniques that reduce cold start latencies (Catalyzer [23], REAP [57], FaaSnap [8], Replayable [62], Prebaking [53], Pagurus [33]) by checkpointing the initialized Groundhog process along with the function's process. Groundhog's memory overhead could be easily reduced to be proportional to the number of dirtied memory pages. The reduction of the memory overhead comes at the cost of a one-time on-critical-path copy-on-write per unique modified page. Since snapshotting is an infrequent operation in Groundhog, we have not attempted these optimizations.

5.6 Dummy Requests

Groundhog optimizes for the on-critical path latency and chooses to take a snapshot after a dummy request is processed
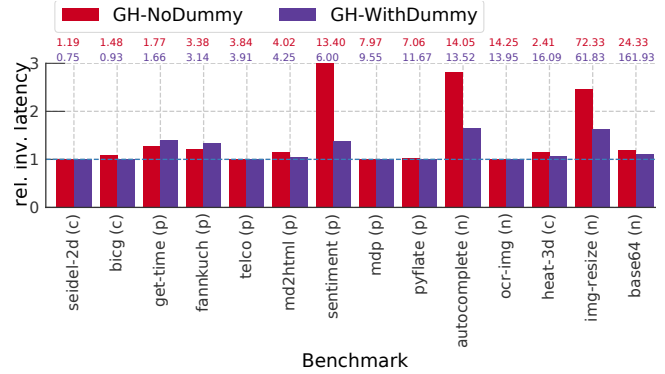

Fig. 9. *Latency of function invocations, normalized to the insecure warm baseline* BASE*, with and without executing a dummy request prior to taking the snapshot. The off-critical-path restore time (in ms) is shown over each benchmark. The symbols (p), (c) and (n) denote Python, C and Node.js benchmarks, respectively. The figure is capped at 3X the insecure baseline latency. The sentiment (p) benchmark relative invoker latency is 10X that of the insecure baseline when snapshotting without a dummy request.* **Lower numbers are better.**

to allow high-level languages to trigger lazy paging, lazy class loading and any application-level initialization of global state, and to capture these in the snapshot. Snapshotting without a dummy request would cause these relatively expensive operations to be re-executed after every state restoration, which would increase the latency of subsequent function activations. This is particularly relevant when the function runs in an interpreted runtime like Python or Node.js, which heavily rely on lazy loading of classes and libraries [38]. On a representative set of benchmarks, Fig. 9 shows that most benchmarks benefit in terms of latency when the snapshot captures the function's state after a dummy request. However, taking a snapshot after the dummy invocation can lead to increased restoration costs in some cases, if the language runtime aggressively modifies the memory layout.

6 Related work

Fork-based request isolation A standard technique for request isolation in services, not FaaS specifically, is to fork a clean state to serve every request. For example, the Apache web server [10], using the default Apache Prefork MPM. uses this approach to isolate client sessions from each other. The same idea can be used for request isolation in FaaS. However, fork() does not work with multithreaded functions or runtimes without extensive modifications to prepare all threads for a consistent snapshot [23]. Even for single-threaded functions, a fork-based approach is less performant than Groundhog (see §5) due to the high cost of forking a new process and the page-copying faults on the critical path for all written pages. The cost of fork itself can be reduced using lighter process-like abstractions such as lightweight contexts (lwCs) [39], but this does not reduce the cost of page copying on the critical path.

Advances in reducing container cold-start latencies Reducing container cold-start latencies is an active area of research. Several techniques have been proposed, including maintaining pre-warmed idle containers for a function [7, 11], maintaining a pool of containers that can be repurposed [43, 54], maintaining partially initialized runtimes with loaded libraries as in SOCK [44], relaxing isolation between functions by allowing functions from the same app developer to share containers (SAND [4], Azure [41]), and starting from slim container images and adding non-essential functions only when needed (CNTR [55]). These techniques do not provide request isolation, the problem that Groundhog targets, but they can be combined with Groundhog to solve the cold-start latency and the request isolation problems simultaneously.

Other methods of reducing cold-start latencies rely on snapshotting and restoration, which Groundhog also uses. Replayable [62], making use of the phased nature of runtime initialization, proposed lowering cold-start latencies by snapshotting after the initialization phase and then starting cold invocations from this snapshot. In principle, this approach can also be used for request isolation by starting each invocation from such a snapshot but existing snapshot/restore techniques have overheads that can be orders of magnitude higher than those of Groundhog because they start a new execution environment for each request rather than reusing an existing environment as Groundhog does.

Snapshotting techniques based on CRIU [19, 21, 22, 49, 58] serialize snapshots to persistent storage and are insufficient for request isolation due to the high overhead of deserialization during restoration, which is on the order of seconds.

CRIU-based techniques that store snapshots in memory lower this overhead, but not sufficiently. For example, VAS-CRIU [59] treats the address space as a first-class OS primitive, allowing an address space to be attached to any process. However, container restoration time is still on the order of ~0.5s. SEUSS [18] takes a unikernel approach, building a customized VM for each function where everything runs in kernel space. SEUSS allows incremental snapshots to jump-start functions. However, SEUSS (and VAS-CRIU) rely on copy-on-write, thus increasing the in-function latency, like the fork-based approach.

Catalyzer [23] trades function-start latency for in-function latency using a lazy restoration that incurs page faults. REAP [57] reduces the cost of these page faults by eagerly prefetching pages that were part of the active working set of the function in the past. However, overall function latencies after a restoration are still high: For a simple hello-world function that executes in 1ms without restoration, Catalyzer and REAP latencies with restoration are 232ms and 60ms, respectively. In contrast, Groundhog can restore a C hello world function in ~0.5 ms and an equivalent Python function in ~1.7 ms off the critical path. Systems such as Catalyzer [23] offer a warm-boot configuration that clones a running function instance by sharing its base-EPT memory mappings on a CoW basis.

Warm boot configurations, if used for request isolation (i.e., a clone is created to handle each request), will have a fork-like performance profile.

FaaSnap [8] performs a different optimization – it enhances the pre-fetching of pages. For instance, it does concurrent prefetching while the VM is loading, and fetches pages in the approximate order of loading such that pages have a higher chance of being fetched by the time the function needs them. These optimizations further reduce the latency of cold-starts by 1.4x relative to a baseline without the optimizations. Nonetheless, overheads are high: The restoration of a simple hello world in FaaSnap takes as much time as it does in REAP.

Cloudflare Workers [20], Faastly [25, 56], and FAASM [52] solve the cold-start problem by relying on software-fault isolation (SFI) using V8 isolates and WebAssembly [64]. Here, several function spaces – called Faaslets in FAASM– are packed into a single running process, relying on SFI to isolate them from each other. Obtaining a fresh Faaslet for a function invocation amounts to remapping an unused Faaslet's heap to a previously checkpointed, pre-warmed state of the function. WebAssembly limits the heap to a contiguous 4GB region, so this remapping is fast and effectively solves the cold-start problem. The FAASM paper notes that the same idea can be used for efficient request isolation by applying the remapping between requests. We compared the performance of this request isolation approach to that of Groundhog in §5.3.3. Unlike Groundhog, this technique is limited to languages, runtimes and threading models that can be compiled to WebAssembly.

7   Conclusion

Groundhog builds an efficient in-memory process state snapshot and restore facility to provide sequential request isolation in FaaS platforms. Groundhog's design is agnostic to the FaaS platform, OS kernel, programming languages, runtimes, and libraries used to write functions. Groundhog overheads on end-to-end latency and throughput are modest, and lower than what could be achieved by repurposing state-of-the art techniques for solving the container cold-start problem to provide sequential request isolation.

Acknowledgments

References

[1] Cve-2014-0160: The Heartbleed Bug.

[2] AARON PATTERSON. CVE-2022-23633: Action Pack Rails possible leak of response to subsequent requests. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-23633, Accessed 12.07.2022.

[3] AGACHE, A., BROOKER, M., IORDACHE, A., LIGUORI, A., NEUGEBAUER, R., PIWONKA, P., AND POPA, D.-M. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)* (2020), pp. 419–434.

[4] AKKUS, I. E., CHEN, R., RIMAC, I., STEIN, M., SATZKE, K., BECK, A., ADITYA, P., AND HILT, V. SAND: Towards high-performance serverless computing. In *2018 Usenix Annual Technical Conference (USENIX ATC 18)* (2018), pp. 923–935.

[5] ALZAYAT, MOHAMED AND MACE, JONATHAN AND DRUSCHEL, PETER AND GARG, DEEPAK . Groundhog Project Website. https://groundhog.mpi-sws.org/.

[6] AMAZON AWS. AWS Lambda. https://aws.amazon.com/lambda/.

[7] AMAZON AWS. Release: AWS Lambda on 2014-11-13. https://aws.amazon.com/blogs/aws/new-provisioned-concurrency-for-lambda-functions//, Accessed 24.11.2021.

[8] AO, L., PORTER, G., AND VOELKER, G. M. Faasnap: Faas made fast using snapshot-based vms. In *Proceedings of the Seventeenth European Conference on Computer Systems* (2022), pp. 730–746.

[9] APACHE. OpenWhisk. https://openwhisk.apache.org/.

[10] APACHE. Apache MPM prefork. https://httpd.apache.org/docs/2.4/mod/prefork.html, Accessed 02.03.2021.

[11] APACHE. Pre-Warmed actions in Openwhisk. https://github.com/apache/openwhisk/blob/master/docs/actions.md/, Accessed 24.11.2021.

[12] APACHE TOMCAT SECURITY team. CVE-2020-13943: Apache Tomcat possible leakage of previous request headers. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-13943, Accessed 12.07.2022.

[13] APACHE TOMCAT SECURITY team. CVE-2022-25762: Apache Tomcat request mix-up. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-25762, Accessed 12.07.2022.

[14] AWS LAMBDA. Implementing statelessness in functions. https://docs.aws.amazon.com/lambda/latest/operatorguide/statelessness-functions.html, Accessed 01.09.2022.

[15] AWS LAMBDA. Security Overview of AWS Lambda. https://docs.aws.amazon.com/whitepapers/latest/security-overview-aws-lambda/lambda-executions.html, Accessed 01.09.2022.

[16] AWS LAMBDA. Predictable start-up times with Provisioned Concurrency). https://aws.amazon.com/blogs/compute/new-for-aws-lambda-predictable-start-up-times-with-provisioned-concurrency/, Accessed 02.03.2021.

[17] BOUCHER, S., KALIA, A., ANDERSEN, D. G., AND KAMINSKY, M. Putting the" micro" back in microservice. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (2018), pp. 645–650.

[18] CADDEN, J., UNGER, T., AWAD, Y., DONG, H., KRIEGER, O., AND APPAVOO, J. Seuss: skip redundant paths to make serverless fast. In *Proceedings of the Fifteenth European Conference on Computer Systems* (2020), pp. 1–15.

[19] CHEN, Y. Checkpoint and restore of micro-service in docker containers. In *Proceedings of the 3rd International Conference on Mechatronics and Industrial Informatics* (2015/10), pp. 915–918.

[20] CLOUDFLARE. Cloudflare Workers. https://workers.cloudflare.com/, Accessed 25.11.2021.

[21] COOPERMAN, G., ANSEL, J., AND MA, X. Transparent adaptive library-based checkpointing for master-worker style parallelism. In *Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)* (2006), vol. 1, IEEE, pp. 9–pp.

[22] CRIU. Checkpoint/Restore In Userspace). https://www.criu.org/, Accessed 03.12.2020.

[23] DU, D., YU, T., XIA, Y., ZANG, B., YAN, G., QIN, C., WU, Q., AND CHEN, H. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (2020), pp. 467–481.

[24] EMPTYMONKEY . A ptrace library designed to simplify syscall injection in Linux. https://github.com/emptymonkey/ptrace_do, Accessed 03.12.2020.

[25] FASTLY. https://www.fastly.com/). https://www.fastly.com/, Accessed 25.11.2021.

[26] GOOGLE. Google Cloud Functions. https://cloud.google.com/functions, Accessed 12.01.2022.

[27] GOOGLE CLOUD FUNCTIONS. Tips & Tricks for Cold Start). https://cloud.google.com/functions/docs/bestpractices/tips, Accessed 02.03.2021.

[28] HAAS, A., ROSSBERG, A., SCHUFF, D. L., TITZER, B. L., HOLMAN, M., GOHMAN, D., WAGNER, L., ZAKAI, A., AND BASTIEN, J. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2017), pp. 185–200.

[29] IBM. IBM Cloud functions. https://cloud.ibm.com/functions/, Accessed 12.01.2022.

[30] JACOB ROTHSTEIN. CVE-2020-26281: async-h1 HTTP/1.1 parser for Rust leak different user's request. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-26281, Accessed 12.07.2022.

[31] JANGDA, A., POWERS, B., BERGER, E. D., AND GUHA, A. Not so fast: Analyzing the performance of webassembly vs. native code. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)* (2019), pp. 107–120.

[32] JOHN GRAHAM-CUMMING (CLOUDFLARE). Cloudbleed: Incident report on memory leak caused by Cloudflare parser bug. https://blog.cloudflare.com/incident-report-on-memory-leak-caused-by-cloudflare-parser-bug/, Accessed 11.07.2022.

[33] LI, Z., CHEN, Q., AND GUO, M. Pagurus: Eliminating cold startup in serverless computing with inter-action container sharing. *arXiv preprint arXiv:2108.11240* (2021).

[34] LINUX. Fork system call. https://man7.org/linux/man-pages/man2/fork.2.html/, Accessed 21.04.2021.

[35] LINUX. ptrace – process trace interface. https://man7.org/linux/man-pages/man2/ptrace.2.html/, Accessed 21.04.2021.

[36] LINUX. SOFT-DIRTY PTEs. https://www.kernel.org/doc/Documentation/vm/soft-dirty.txt/, Accessed 21.04.2021.

[37] LINUX. Userfaultfd. https://www.kernel.org/doc/html/latest/admin-guide/mm/userfaultfd.html, Accessed 21.04.2021.

[38] LION, D., CHIU, A., SUN, H., ZHUANG, X., GRCEVSKI, N., AND YUAN, D. Don't get caught in the cold, warm-up your JVM: Understand and eliminate JVM warm-up overhead in data-parallel systems. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (2016), pp. 383–400.

[39] LITTON, J., VAHLDIEK-OBERWAGNER, A., ELNIKETY, E., GARG, D., BHATTACHARJEE, B., AND DRUSCHEL, P. Light-weight contexts: An OS abstraction for safety and performance. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (2016), pp. 49–64.

[40] LOUIS-NOEL POUCHET. Polybench/C). http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/, Accessed 21.11.2021.

[41] MICROSOFT. Azure Functions. https://azure.microsoft.com/en-us/services/functions/, Accessed 04.01.2022.

[42] MOHAMED ALzayat. Detecting a bug in soft-dirty bits Kernel v5.6+. https://lore.kernel.org/linux-mm/daa3dd43-1c1d-e035-58ea-994796df4660@suse.cz/T/, Accessed 20.04.2021.

[43] MOHAN, A., SANE, H., DOSHI, K., EDUPUGANTI, S., NAYAK, N., AND SUKHOMLINOV, V. Agile cold starts for scalable serverless. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud*

19) (2019).

[44] OAKES, E., YANG, L., ZHOU, D., HOUCK, K., HARTER, T., ARPACI-DUSSEAU, A., AND ARPACI-DUSSEAU, R. SOCK: Rapid task provisioning with serverless-optimized containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (2018), pp. 57–70.

[45] OPENFAAS. OpenFaaS. https://www.openfaas.com/, Accessed 12.01.2022.

[46] OPENWHISK. OpenWhisk commit. https://github.com/apache/openwhisk/commit/ed3f76e38d89468d11e862ee0539e74f02ac7f8e.

[47] OWASP . OWASP Serverless Top 10. https://owasp.org/www-project-serverless-top-10/, Accessed 02.03.2021.

[48] RICH JONES. Gone in 60 Milliseconds: Intrusion and Exfiltration in Server-less Architectures. https://media.ccc.de/v/33c3-7865-gone_in_60_milliseconds, Accessed 02.03.2021.

[49] RIEKER, M., ANSEL, J., AND COOPERMAN, G. Transparent user-level checkpointing for the native posix thread library for linux. In *PDPTA* (2006), vol. 6, pp. 492–498.

[50] SHAHRAD, M., BALKIND, J., AND WENTZLAFF, D. Architectural implications of function-as-a-service computing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (2019), MICRO '52, p. 1063–1075.

[51] SHAHRAD, M., FONSECA, R., GOIRI, I., CHAUDHRY, G., BATUM, P., COOKE, J., LAUREANO, E., TRESNESS, C., RUSSINOVICH, M., AND BIANCHINI, R. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)* (July 2020), pp. 205–218.

[52] SHILLAKER, S., AND PIETZUCH, P. FAASM: Lightweight isolation for efficient stateful serverless computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)* (July 2020), pp. 419–433.

[53] SILVA, P., FIREMAN, D., AND PEREIRA, T. E. Prebaking functions to warm the serverless cold start. In *Proceedings of the 21st International Middleware Conference* (2020), pp. 1–13.

[54] STENBOM, O. Refunction: Eliminating serverless cold starts through container reuse. Master's thesis, Imperial College London, 2019.

[55] THALHEIM, J., BHATOTIA, P., FONSECA, P., AND KASIKCI, B. Cntr: Lightweight OS containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (2018), pp. 199–212.

[56] TYLER MCMULLEN (FASTLY). Lucet: A Compiler and Runtime for High-Concurrency Low-Latency Sandboxing). https://popl20.sigplan.org/details/prisc-2020-papers/13/-Lucet-A-Compiler-and-Runtime-for-High-Concurrency-Low-Latency-Sandboxing, Accessed 03.12.2020.

[57] USTIUGOV, D., PETROV, P., KOGIAS, M., BUGNION, E., AND GROT, B. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21)* (2021).

[58] VASAVADA, M., MUELLER, F., HARGROVE, P. H., AND ROMAN, E. Comparing different approaches for incremental checkpointing: The showdown. In *Linux Symposium* (2011), vol. 69.

[59] VENKATESH, R. S., SMEJKAL, T., MILOJICIC, D. S., AND GAVRILOVSKA, A. Fast in-memory criu for docker containers. In *Proceedings of the International Symposium on Memory Systems* (2019), pp. 53–65.

[60] VICTOR STINNER. The Python Performance Benchmark Suite). https://pyperformance.readthedocs.io, Accessed 21.11.2021.

[61] VOGT, D., GIUFFRIDA, C., BOS, H., AND TANENBAUM, A. S. Lightweight memory checkpointing. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (2015), IEEE, pp. 474–484.

[62] WANG, K.-T. A., HO, R., AND WU, P. Replayable execution optimized for page sharing for a managed runtime environment. In *Proceedings of the Fourteenth EuroSys Conference 2019* (2019), pp. 1–16.

[63] WAYNE BEATON. CVE-2020-27218: Eclipse jetty possible data injection into subsequent request. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-27218, Accessed 12.07.2022.

[64] WEBASSEMBLY. WebAssembly). https://webassembly.org/.

[65] WESLEY BEARY. CVE-2019-16779: RubyGem excon leak previous response. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-16779, Accessed 12.07.2022.

A Artifact Appendix

A.1 Abstract

Our artifact comprises Groundhog which runs on standard Linux kernels as well as Groundhog's integration with Open-Whisk for Python, NodeJS, and C. Our artifact is available on MPI-SWS's institutional public repository. The functional correctness of Groundhog is automatically tested through CI/CD. We provide automated scripts for deploying OpenWhisk with Groundhog-enabled runtimes and reproducing the results.

A.2 Description & Requirements

Our artifact was produced on standard server hardware. We ran all our experiments on VMs hosted on a private cloud hosting OpenStack/Microstack (ussuri, revision 233). Each physical host had an Intel Xeon E5-2667 v2 2-socket, 8-cores/socket processor (SMT turned off), 256GB RAM and a 1 TB HDD.

We evaluated Groundhog on microbenchmarks that we have developed, in addition to functions from the pyperformance benchmark [60], the PolyBench/C [40], and the Python/Node.js FaaSProfiler benchmark [50] suites.

A.2.1 How to access Visit the Groundhog website at https://groundhog.mpi-sws.org which links to the project repositories at https://gitlab.mpi-sws.org/groundhog. Groundhog's source code can be found in the project "Groundhog" which has a README file for getting started. The "Groundhog" repository is setup to run automatic functionality tests. For reproducing the results of this paper, an "Instructions" repository has been setup, which provides instructions on how to use the scripts that automate deploying OpenStack (if needed) and OpenWhisk, as well as running and plotting the results of the experiments.

In addition to the automated deployment scripts, we provide an Ubuntu VM with a Groundhog-enabled OpenWhisk deployment. Details are in https://gitlab.mpi-sws.org/groundhog/instructions.

A.2.2 Hardware dependencies For Groundhog: standard x86 hardware suffices. For the evaluation, one or more standard Linux servers (256+ GB of RAM, 2-sockets with 8-cores/socket processor, and 500GB+ of disk space), or the ability to provision VMs on a cloud.

A.2.3 Software dependencies For Groundhog: A standard Linux Operating System (e.g., Ubuntu 20.04) with standard development packages installed (build-essential and libglib2.0-dev). The current Groundhog implementation depends on Linux kernel features available in v3.11+. We identified and reported a bug that affected the accuracy of the SD-Bits memory tracking in v5.6-v5.11 [42]. For the evaluation, we used the stock kernel v5.4 shipped with Ubuntu 20.04. For the experiments: Groundhog + OpenWhisk. For plotting: Python (jupyter-lab, pandas, numpy, matplotlib, seaborn).

A.2.4 Benchmarks A copy of all used benchmarks is included in our source code release (to maintain a uniform structure and facilitate automation).

A.3 Setup

If you are not able to create VMs with custom resolvable names, you may want to manage you own private cloud by installing and configuring OpenStack (automatic OpenStack deployment scripts available at https://gitlab.mpi-sws.org/groundhog/automation → openstack).

In our experiments, OpenWhisk is deployed on 2 VMs to allow proper performance isolation (core components on one VM, and the invoker component, which launches functions containers that can be Groundhog enabled, on another VM). Latency experiments can be done using 2 VMs, while throughput experiments need 3 VMs.

A single VM image can be prepared (through our automated scripts at https://gitlab.mpi-sws.org/groundhog/automation -> prepare-vm) or downloaded from https://groundhog.mpi-sws.org/downloads/groundhog-ow-ubuntu20.qcow2. Once the VM image has been prepared or downloaded, it can be used to launch all needed VMs with the following naming convention: ow-core-X, ow-invoker0-X, and ow-client-X (where X $\in \mathbb{N}$).

A.4 Evaluation workflow

After the setup, experiments can be run using the scripts provided in the automation repository (https://gitlab.mpi-sws.org/groundhog/automation → experiments)

A.4.1 Major Claims

- *(C1): Groundhog achieves request isolation at a cheaper cost compared to CoW-based techniques (e.g., fork()). This is shown by the experiment (E1) described in 5.2. Results are illustrated in Fig. 3 and discussed in 5.2.3.*
- *(C2): Groundhog has a modest latency overhead. This is shown by the experiment (E2) described in 5.3. Results are illustrated/reported in Fig. 4.*
- *(C3): Groundhog throughput overhead is lower than that of CoW-based techniques (fork). This is shown by the experiment (E3) described in 5.3. Results are illustrated/reported in Fig. 5.*
- *(C4): Groundhog throughput scales near-linearly with cores. This is indicated by the experiment (E4) discussed in 5.3.4. Results are illustrated/reported in Fig. 7.*
- *(C5): Groundhog's restoration overheads are mostly a function of the function's memory footprint and the number of dirtied pages. This is discussed in §5.4. The data from experiment (E1) can be used to produce Fig. 8.*
- *(C6): Groundhog is transparent to the tenant. This is described in the design section (3) and can be verified by noticing that no change was required to the tenants' provided functions (https://gitlab.mpi-sws.org/groundhog/automation/-/tree/main/benchmarks/func.*

Table 1. A summary of the time required (human, compute) to run the down-sized experiments. The human only has to verify the script configuration on the setup (e.g. VMs IPs/names) and run the script.

	Scripts to run	Time Human	Time Compute	Corresponding Figure	#Experiments
E1	run_latency_microbenchmark_vary_dirtied_fast.sh run_latency_microbenchmark_vary_dirtied_slow.sh run_latency_microbenchmark_vary_pages_fast.sh run_latency_microbenchmark_vary_pages_slow.sh	~5 mins	~2 hours	Fig. 3	120
E2	run_latency_python.sh run_latency_nodejs.sh run_latency_pyperf.sh run_latency_polybench_long.sh # > 10s per request run_latency_polybench_short.sh # < 10s per request	~5 mins	~10 hours	Fig. 4	232
E3	run_xput_python.sh run_xput_nodejs.sh run_xput_pyperf.sh run_xput_polybench.sh	~5 mins	~20 hours	Fig. 5	232
E4	run_scalability_1core.sh run_scalability_2core.sh run_scalability_3core.sh run_scalability_4core.sh	~5 mins	~15 hours	Fig. 7	144

A.4.2 Experiments

Experiment automation scripts are available at https://gitlab.mpi-sws.org/groundhog/automation → experiments. The scripts default to experiments of a down-sized length to keep the total runtime reasonable. Full-length raw experiment data from the paper can be found at https://groundhog.mpi-sws.org/downloads/DATA-EUROSYS23.tgz

[How to (all experiments)]

[Preparation]   After the experiment setup is ready (VM groups are created and running), we need an additional controller node/VM/server (which can use the same VM image) that can communicate with the OpenWhisk core VM named ow-core-X. On this controller node, experiments will be started and data will be automatically retrieved. A summary of the down-sized execution time required (human, compute), and the scripts used to run each experiment (down-sized) can be found in Table 1.

[Execution] Run the corresponding scripts (as shown in Table 1) on one or more VM groups (a latency VM group consists of 2 VMs — ow-core-X and ow-invoker0-X, while a throughput/scalability VM group consists of 3 VMs — ow-core-X, ow-invoker0-X, and ow-client-X). All scripts are expected to run to completion without terminating with an error/exception.

[Results] Results will be automatically copied back to the controller node at /local/workspace/automation/benchmarks/benchmarks/. A one-time data clean-up should be run for

experiments E2-4 by passing the results directory as an argument to (https://gitlab.mpi-sws.org/groundhog/automation → plot/prepare_data.sh). Once the data preparation script finishes, a table of the number of requests per configuration will be printed.

[Plotting the results] A jupyter notebook for plotting the results is provided at https://gitlab.mpi-sws.org/groundhog/automation → plot. In the top section, update the path to the data source and run the notebook. Details on how to structure the directories for plotting are available in the repository.

[Comparison with FAASM *]* The latency comparison against FAASM uses FAASM's microbenchmarks repo https://github.com/faasm/experiment-microbench.