Efficient Request Isolation in Function-as-a-Service

A dissertation submitted towards the degree
Doctor of Engineering
of the Faculty of Mathematics and Computer Science
of Saarland University

by
Mohamed Wael Alzayat

Saarbrücken, 2023

Date of Colloquium	March 8th, 2024
Dean of Faculty	Univ.-Prof. Dr. Jürgen Steimle
Chair of the Committee	Prof. Dr. Isabel Valera
Reporters	Prof. Dr. Peter Druschel
	Prof. Dr. Deepak Garg
	Dr. Antoine Kaufmann
Academic Assistant	Dr. Aina Linn Georges

بسم الله الرحمن الرحيم

Dedicated to my *family*!

Abstract

As cloud applications become increasingly event-driven, Function-as-a-Service (FaaS) is emerging as an important abstraction. FaaS allows tenants to state their application logic as stateless functions without managing the underlying infrastructure that runs and scales their applications.

FaaS providers ensure the confidentiality of tenants' data, to a limited extent, by isolating function instances from one another. However, for performance considerations, the same degree of isolation does not apply to sequential requests activating the same function instance. This compromise can lead to confidentiality breaches since bugs in a function implementation or its dependencies may retain state and leak data across activations. Moreover, platform optimizations that assume function statelessness may introduce unexpected behavior if the function retains state, jeopardizing correctness.

This dissertation presents two complementary systems: Groundhog and CtxTainter. Groundhog is a black-box and programming-language-agnostic solution that enforces confidentiality by efficiently rolling back changes to a function's state after each function activation, effectively enforcing statelessness by breaking all data flows at the request boundary. CtxTainter is a development-phase dynamic data flow analysis tool that detects data flows that violate the statelessness assumption and reports them to the developer for reviewing and fixing.

Zusammenfassung

Da Cloud-Anwendungen zunehmend ereignisgesteuert sind, entwickelt sich Function-as-a-Service (FaaS) zu einer wichtigen Abstraktion. FaaS ermöglicht es Cloud-Kunden, ihre Anwendungslogik als zustandslose Funktionen anzugeben, ohne die zugrunde liegende Infrastruktur verwalten zu müssen, die ihre Anwendungen ausführt und skaliert.

FaaS-Anbieter gewährleisten in begrenztem Umfang die Vertraulichkeit der Daten von Kunden, indem sie Funktionsinstanzen voneinander isolieren. Aus Leistungserwägungen gilt diese Isolierung jedoch nicht für aufeinanderfolgende Anfragen, die dieselbe Funktionsinstanz aktivieren. Dieser Kompromiss kann zu Vertraulichkeitsverletzungen führen, da Fehler in einer Funktionsimplementierung oder ihren Abhängigkeiten den Zustand beibehalten und Daten über Aktivierungen hinweg offenlegen. Auch Plattformoptimierungen, die von der Zustandslosigkeit einer Funktion ausgehen, können die Korrektheit gefährden, wenn die Funktion Zustand beibehält.

In dieser Dissertation werden zwei Systeme vorgestellt: Groundhog und CtxTainter. Groundhog ist eine Blackbox- und programmiersprachenunabhängige Lösung, die Vertraulichkeit erzwingt, indem sie Änderungen am Zustand einer Funktion nach jeder Aktivierung effizient rückgängig macht und den Datenfluss an der Anfragegrenze unterbricht. CtxTainter ist ein Tool zur dynamischen Datenflussanalyse in der Entwicklungsphase das Datenflüsse erkennt, die die Zustandslosigkeit verletzen, und sie dem Entwickler zur Überprüfung und Korrektur meldet.

Acknowledgments

Above all, I praise and thank Allah, the almighty, for providing me with this opportunity and granting me the capability to prepare and present this work.

Without the guidance and assistance of several people, this thesis would not have appeared in its current form. I would, therefore, like to offer my sincere thanks to all of them.

First and foremost, I would like to express my sincere gratitude to my advisers, Peter Druschel and Deepak Garg, for their continuous guidance and support throughout my Ph.D. journey. Their continuous non-intrusive guidance provided me with the safe environment I needed to develop as a person and as a researcher. Their advice, both at the personal and the professional level, immensely helped me persevere and be resilient through the most difficult of times. I am forever thankful for their unwavering support.

I would like to thank Antoine Kaufmann, not only for agreeing to be on my thesis committee and taking the time to review my thesis but also for his insightful suggestions and stimulating discussions.

Special gratitude and thanks to Jonathan Mace who is both a mentor and a collaborator. His advice on approaching and identifying interesting problems in new research landscapes was invaluable. I want to thank him for his valuable suggestions and input in the Groundhog project.

Special appreciation to Krishna P. Gummadi for the many thought-provoking discussions and guidance on asking the right questions. I deeply thank Bala Chandrasekaran for all our discussions that blended advice, guidance, and feedback. These discussions benefited me both personally and professionally. Many thanks to Patrick Loiseau for all the helpful suggestions and guidance. I would like to thank them all for their collaboration and guidance on the blockchain projects.

I would like to extend my thanks to all the MPI-SWS faculty and researchers for all the intriguing discussions we had. Notably, Björn B. Brandenburg for his general parenting and research advice as well as his valuable inputs to the Pacer Project, Bobby Bhattacharjee and Laurent Bindschaedler for their generous feedback and suggestions on my thesis work, and Keon Jang for his generous advice on approaching the cloud research landscape.

I would like to thank my fellow student collaborators, Aastha Mehta, Roberta De Viti, and Johnnatan Messias. Aastha provided me with technical guidance on navigating the field of systems research during our work on the Pacer project. Roberta and Johnnatan, my struggle companions during the Pacer and the blockchains projects, respectively, and Ralf Jung, my office mate who was always available and helpful. Anjo Vahldiek-Oberwagner and Eslam Elnikety for their sincere advice and suggestions that helped shape my research and professional career steps. My sincere gratitude extends to all MPI-SWS postdocs and students with whom I had many interesting and stimulating discussions.

Special thanks to all MPI's staff! Rose Hoberman for the academic writing course and for all her valuable and timely feedback on my papers, thesis draft, and talks. Justine Kaufmann for organizing the English-reading groups and discussions,

Mary-Lou and Gretchen Gravelle for their help with staying on track with regards to the program requirements. Claudia Richter, Annika Meiser, Sarah Naujoks, Alexandra Klasen-Schmitt, Danielle Dalton, Isabel Thät, and Brigitta Hansen for making all administrative issues a breeze. Many thanks to Christian Klein and Carina Schmitt for their top-notch IT expertise and their dedication. I deeply thank all of them.

On a personal level, I would like to thank my coffee-breaks venting companions at MPI, Saarland University, and CISPA. Mohamed H. Gad-elrab, Akram El-korashy, Mohamed Elgharib, Mossad Helali, Abdallah Dawoud, Ahmed Salem, and Vaastav Anand. My gratitude extends to all my friends in Saarbrücken and all over the world, for always being there for me whenever I needed them.

Last but certainly not least, words will never be enough to express how grateful and thankful I am to my family. My parents, Wael and Hala, brother Abdelrahman and sisters Safya and Sarah, for their sincere support, encouragement, prayers throughout my life and my long education journey, and their patience throughout my absence; I truly miss you all!

I was most fortunate to have taken this journey with my soul mate, Lena, my lovely wife. Without your encouragement, having my back, providing emotional as well as physical support where and when needed, and of course continuous prayers, this work would not have been completed. During this work, our family was extended by Habiba and Sulaiman, who multiplied the joy and fun in our lives. I cannot thank you enough, may Allah give you all the best in return.

This work was supported in part by the European Research Council (ERC Synergy imPACT 610150) and the German Science Foundation (DFG CRC 1223) grants.

Publications

Parts of this thesis have appeared in the following publication:

- “Groundhog: Efficient Request Isolation in FaaS”. Mohamed Alzayat, Jonathan Mace, Peter Druschel, and Deepak Garg. In Eighteenth European Conference on Computer Systems (EuroSys’23). 2023.

The following work is published for the first time in this thesis:

- “CtxTainter: Request-Aware Taint Analysis for FaaS”. Mohamed Alzayat, Peter Druschel, and Deepak Garg.

Additional publications while at MPI-SWS:

- “Pacer: Comprehensive Network Side-Channel Mitigation in the Cloud”. Aastha Mehta, Mohamed Alzayat, Roberta De Viti, Björn B. Brandenburg,Peter Druschel, and Deepak Garg. In 31st USENIX Security Symposium (USENIX Security 22). 2022.

- “Modeling Coordinated vs. P2P Mining: An Analysis of Inefficiency and Inequality in Proof-of-Work Blockchains”. Mohamed Alzayat, Johnnatan Messias, Balakrishnan Chandrasekaran, Krishna P. Gummadi, and Patrick Loiseau. arXiv:2106.02970. 2021.

- “Selfish & Opaque Transaction Ordering in the Bitcoin Blockchain: The Case for Chain Neutrality”. Johnnatan Messias, Mohamed Alzayat, Balakrishnan Chandrasekaran, Krishna P. Gummadi, Patrick Loiseau, and Alan Mislove. In Proceedings of the 21st ACM Internet Measurement Conference (IMC ’21). 2021.
- “On Blockchain Commit Times: Ananalysis of how miners choose Bitcoin transactions". Johnnatan Messias, Mohamed Alzayat, Balakrishnan Chandrasekaran, and Krishna P. Gummadi. In The Second International Workshop on Smart Data for Blockchain and Distributed Ledger (SDBD’20). 2020.

Contents

Abstract v

Zusammenfassung v

Acknowledgments viii

Publications x

List of Figures xv

List of Tables xvii

1. Introduction 1
- 1.1. FaaS: A Stateless Paradigm . . . 3
- 1.2. Confidentiality and Correctness: A Contract . . . 4
- 1.3. The Case for Sequential Request Isolation . . . 6
- 1.4. Thesis Contributions . . . 9
- 1.5. Organization . . . 13

2. Conceptual framework 15
- 2.1. Execution Environment Reuse . . . 15
- 2.2. The Abstraction . . . 17
  - 2.2.1. Data Sources . . . 17
  - 2.2.2. Data Sinks . . . 18
  - 2.2.3. Data Flows . . . 19
  - 2.2.4. External data flows . . . 20
- 2.3. Confidentiality and Correctness . . . 20
  - 2.3.1. The Tenant's Side of the FaaS Contract . . . 21
  - 2.3.2. The Provider's Side of the FaaS Contract . . . 22
- 2.4. Fulfilling the Contract . . . 23

3. Groundhog: Confidentiality by Design **25**
3.1. Design Preliminaries . . . 28
3.1.1. Insights . . . 29
3.1.2. Design alternatives . . . 32
3.1.3. Threat model . . . 33
3.2. Groundhog Design and Implementation . . . 34
3.2.1. Container initialization . . . 36
3.2.2. Snapshotting the function process . . . 37
3.2.3. Tracking state modifications . . . 37
3.2.4. Restoring to the snapshotted state . . . 38
3.2.5. Enforcing request isolation . . . 39
3.3. Evaluation . . . 41
3.3.1. Evaluation Overview . . . 42
3.3.2. Microbenchmarks . . . 45
3.3.3. FaaS Benchmarks . . . 49
3.3.4. Deconstructing restoration overheads . . . 60
3.3.5. Snapshotting overhead . . . 62
3.3.6. Dummy Requests . . . 63
3.4. Related work . . . 64
3.5. Summary . . . 67

4. CtxTainter: Analyzing Statelessness **69**
4.1. Design Preliminaries . . . 71
4.1.1. Design Dimensions . . . 72
4.1.2. Assumptions . . . 77
4.2. Design and Implementation . . . 78
4.2.1. Overview . . . 78
4.2.2. Inputs and Outputs . . . 80
4.2.3. Metadata Extraction . . . 82
4.2.4. Request Contexts and Taint Tracking . . . 84
4.2.5. Taint Propagation Rules . . . 85
4.2.6. Detecting Boundary-Crossing Flows . . . 86
4.2.7. Reconstructing Violating Flows . . . 87
4.2.8. Limitations . . . 88
4.3. Implementation . . . 89
4.4. Evaluation . . . 91
4.4.1. Handcrafted examples . . . 92
4.4.2. Benchmark examples . . . 99

4.5. Related Work 101
4.5.1. Static Analysis 101
4.5.2. Dynamic Analysis 102
4.5.3. Hybrid approaches 103
4.5.4. Verification 103
4.5.5. Information flow in FaaS 104
4.6. Conclusion 105

5. Conclusion and Future Work **107**
5.1. Future Work 108
5.1.1. Groundhog 108
5.1.2. CtxTainter 111

Appendix **113**

A. Detailed Groundhog results **115**

Bibliography **119**

List of Figures

3.1. *Groundhog container life cycle* 29

3.2. *Groundhog Architecture: (1) The manager (solid green box), (2) The function process (f). Groundhog relies on standard Linux kernel utilities.* 34

3.3. *Function latencies varying the number of dirtied pages (left) and the address space size (right). 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.)* 46

3.4. *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 Appendix A. The symbols (p), (c), and (n) denote Python, C, and Node.js benchmarks, respectively.* ***Lower numbers are better.*** 52

3.5. *Relative throughput of* GH*,* GH_{NOP}*, and* FORK *compared to the insecure baseline* BASE*. Detailed numbers are in Appendix A. The symbols (p), (c), and (n) denote Python, C and Node.js benchmarks, respectively.* ***Higher numbers are better.*** 55

3.6. *Restoration duration (off the critical path) of* GH*, and* FAASM*. The symbols (p) and (c) denote Python and C.* 57

3.7. *Throughput scaling with number of cores. Error bars (minute) represent the standard deviation across 6 runs.* 59

3.8. *Restoration overhead (deconstructed) and the one-time snapshotting overhead for a subset of benchmarks.* 60

3.9. *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.*** 63

List of Tables

4.1. *CtxTainter's dependency graph (adjacency matrix) for Listing 4.2. Different shades of gray represent different request contexts.* 95

A.1. Latency and throughput measurements comparing Groundhog to several other systems: **base** baseline OpenWhisk; **GH** Groundhog on OpenWhisk; **faasm** faasm; **fork** fork-based implementation on OpenWhisk; and $\mathbf{GH}_{\text{NOP}}$ Groundhog with no restoration. We run 58 benchmarks across three languages indicated by **(p)** Python, **(c)** C, and **(n)** Node.js. We highlight cells of interest if there is more than a 5% difference as follows: Green indicates results faorable to Groundhog Red indicates results unfaorable to Groundhog Blue indicates the unexpected result that Groundhog is outperforming the baseline. ***Disclaimer***: Faasm throughput measurements are provided for the curious reader, but no conclusions should be drawn based on them as they entangle many variables such as the difference in the platforms, their internal components and deployment, runtimes (native vs WebAssembly), as well as the isolation mechanism. 115

A.2. Latency and throughput measurements showing the overheads of GH_{NOP}(Groundhog with no restoration), GH,FORK, and FAASM relative to an unsecure baseline. We run 58 benchmarks across three languages indicated by **(p)** Python, **(c)** C, and **(n)** Node.js. We highlight cells of interest if there is more than a 5% difference as follows: Green indicates results faorable to Groundhog Red indicates results unfaorable to Groundhog Blue indicates the unexpected result that Groundhog is outperforming the baseline. GH's impact on throughput can be approximated by the relative restoration time compared to the invoker's latency (inv. lat.). ***Disclaimer***: Faasm throughput measurements are provided for the curious reader, but no conclusions should be drawn based on them as they entangle many variables such as the difference in the platforms,

their internal components and deployment, runtimes (native vs WebAssembly), as well as the isolation mechanism. 116

A.3. Groundhog's overhead on the throughput is a function of Groundhog's added overhead both on the critical path (#faults) which has negligible overhead on most functions, as well as the overhead off the critical path (Restoration time) which is mostly a function of the address space size (#pages (K)) and the number of restored pages (#restored (K)) as well as restoring the memory layout as demonstrated earlier in Fig. 3.8. Data is sorted by the restoration time. 117

1 Introduction

Function-as-a-Service (FaaS) is an emerging high-level abstraction for event-driven cloud applications. This abstraction allows tenants to state their application logic as stateless[1] event-triggered[2] functions[3], typically written in a high-level language like Python or JavaScript [32]. The FaaS provider exports an HTTP/S endpoint, which can be used to invoke the FaaS function with arguments and receive results. FaaS has an 'on-demand' charge model: a tenant only pays for the compute time used to execute their functions.

The FaaS provider is responsible for deploying and executing the tenants' functions, provisioning and replicating functions as workload demand fluctuates, and maintaining and multiplexing the hardware and software infrastructure across different tenants and functions. Additionally, FaaS platforms offer tenant functions access to platform services to facilitate building applications. These services typically include storage, such as file-based access to scratch storage on a local

[1] Statelessness means that the FaaS function should externalize any data changes that should be retained across invocations to an external persistent storage. This requirement is critical for the function's correctness because the platform can terminate an idle function instance at any time without any notification.

[2] FaaS functions are activated in response to invocations or triggers and handle events one at a time. Concurrent events may be handled sequentially or by separate instances of the function.

[3] A FaaS function, despite the name, does not map to a single programming language function; rather, it refers to the code provided by the tenant, which can have libraries with many internal functions but collectively provide functionality through a single entry point.

disk, persistent key-value stores, and full relational database backends. Platform services may also provide automatic invocation of tenant functions triggered by timers, writes to certain key-value tuples, or updates to certain rows in a database. Applications stitch together calls to these platform services and to the developer-provided functions.

Ensuring *data confidentiality* is one of the core responsibilities of a FaaS provider. FaaS providers invest in securely multiplexing the hardware and software infrastructure among tenants. They commonly rely on hardware-assisted (e.g. OS- [4, 82, 11, 89, 45], or VMM-based[7, 3])[4] isolation primitives to isolate functions from one another: each function executes within its own execution environment, and different functions do not share the same execution environment, thus preventing a malicious or compromised function from affecting the confidentiality of other functions.

While existing isolation mechanisms effectively isolate function instances from one another, they do not provide any isolation guarantees for sequential requests activating the same function instance. Without proper sequential request isolation, the confidentiality of end-client data is in jeopardy. The lack of *sequential request isolation* in FaaS is the gap that this thesis closes.

In this thesis, we build a theoretical framework that allows us to reason about the confidentiality and correctness guarantees provided by the different components of the FaaS paradigm. Then, we discuss how sequential request isolation can be

[4]Some platforms [29, 41, 106] rely on language-based isolation, which can provide sufficient guarantees for some classes of applications.

achieved within that framework. Finally, we describe and prototype two complementary approaches that allow efficient and correct enforcement of sequential request isolation in FaaS.

1.1 FaaS: A Stateless Paradigm

A key enabler for the FaaS paradigm is an expectation of the FaaS functions to be stateless. Statelessness allows providers to decouple the control plane from the data plane while managing the infrastructure. This separation means that initializing, scaling, and terminating function instances does not risk data loss as long as these operations happen at a time when a function is not actively processing a request. Additionally, this separation gives providers high flexibility in scheduling requests to instances. Because an instance does not retain state, a request may be handled by any instance of the function. Similarly, follow-up requests from the same end-client need not be handled by the same instance.

While the FaaS paradigm, as implemented by commercial providers, expects functions to be stateless, this expectation is not backed by tools to verify or enforce functions' statelessness. In practice, it is not trivial for tenants to ensure that their FaaS functions are stateless, especially when their FaaS functions depend on third-party libraries. A FaaS function, or one of the third-party libraries it uses, might have a bug or may have been originally written for a stateful computing paradigm and retain state that is crucial for security and/or correctness.

To further understand the statelessness requirement in traditional FaaS offerings, it is important to differentiate between the different classes of state that might be *inappropriately* retained by a FaaS function. First, there is *confidentiality-critical* state, such as state derived from the tenant's end-client data. Second, there may be *correctness-critical* state that might be retained for correctness/security reasons, such as the internal state of a pseudo-random number generator (PRNG), a statistical counter, or a stream tokenizer. Finally, there is *non confidential-nor correctness-critical* state, which may be relevant for performance reasons, *e.g.* a cache within a function. *In this thesis, we focus on the confidentiality- and correctness-critical states.*

1.2 Confidentiality and Correctness: A Contract

An important aspect of cloud security is ensuring tenants' *data confidentiality* while *correctly preserving the semantics* of their deployed applications. As with any composable system, reasoning about global properties such as confidentiality and correctness requires understanding the different components of the system and the mutual contracts they implement. Each component is responsible for meeting its side of the contract and is expected to provide the required *infrastructural support* for dependent components to allow them to fulfill their side of the contract as well. A FaaS application depends on components supplied by two parties – the platform provider and the tenant.

The confidentiality and correctness contract in FaaS aims at ensuring that the data confidentiality and correctness properties are maintained throughout the system. The contract (detailed in §2.3) can be viewed as a mechanism to ensure *data confidentiality while maintaining correctness*, provided that the platform provider and the tenant honor their sides of the contract. The data confidentiality property stipulates that no request (a FaaS function activation) should breach the confidentiality of any other request's data (see §1.3 for details), while the correctness property stipulates that for functions that adhere to the contract, the semantics of the function (the tenant's logic responsible for handling a single request) when run under the FaaS provider's infrastructure should be equivalent to its semantics when invoked once and terminated – *i.e.* the semantics of handling a request should not be altered by the platform.

The confidentiality contract requires FaaS provider to implement mechanisms that prevent confidential data leakage through the platform infrastructure or services. Additionally, the platform should offer the infrastructural support that allows tenants to properly configure their FaaS applications and prevent confidential data leakage among their end-clients. As such, tenants that properly configure their FaaS application can securely use the platform services and secure their end-clients' data.

The correctness contract requires tenants to develop FaaS functions that do not depend (for correctness) on any non-externalized state from previous activations of the same FaaS function instance. Functions that fail to externalize correctness-critical state may suffer data loss or altered semantics as the platform

may introduce operational optimizations that expect the absence of retained correctness-critical state.

1.3 The Case for Sequential Request Isolation

FaaS providers implement a variety of mechanisms that provide infrastructural support to enable tenants to configure their functions and data access permissions. 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. Major FaaS providers like AWS Lambda, Azure FunctionApps, Google Cloud Functions, and IBM Cloud Functions [7, 82, 45, 52] associate fine-grained, per-caller[5] credentials to a function activation. Here, activations of the same function can access 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-clients of a tenant's deployed application.

In addition to the aforementioned infrastructural support for configuring proper data access permissions, providers isolate function instances from one another to prevent malicious or compromised functions from affecting the confidentiality of other functions. However, for performance considerations, the same level of isolation does not apply to sequential requests activating the same function instance,

[5] In this thesis, the caller is the entity causally responsible for the activation of a function.

even when those requests come from differently privileged end-clients. Hence, the confidentiality property within a FaaS function is not fully enforced by current FaaS providers. Efficiently addressing this shortcoming without compromising *correctness* is the main goal of this thesis.

While preventing confidential data leaks in the tenant-provided functions is not solely the provider's responsibility, the provider can (as we show in this thesis) provide the infrastructural support for confidentiality to be enforced, lifting a non-trivial effort off the shoulders of the tenant. Without the infrastructural support to isolate different *sequential activations* of the *same function instance*, 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. For example, if the same FaaS 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 in the response to Bob.

Such leaks may arise despite the fact that FaaS functions are typically 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 programming languages rely on libraries (e.g. NumPy, PyTorch, and TensorFlow for Python) that are written in unsafe languages like C/C++ for efficiency.

More generally, “insecure shared space” [92] that is not cleared between sequential uses of an execution environment can, in principle, enable attacks similar to those known from traditional serverful environments, such as the infamous Heartbleed bug [114], the Cloudbleed bug [56], and many others affecting various programming languages, frameworks, and libraries [128, 53, 13, 14, 126, 1].[6]

To provide request isolation, FaaS providers like AWS lambda *suggest, but do not enforce,* partitioning clients from different security domains by redirecting them to distinct functions [19]. This approach requires duplication of the execution environments, which does not scale to services with many mutually distrusting clients, as is common in some e-commerce services.

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 performance perspective: When container initialization is carried out naively, it can take seconds. But even when state-of-the-art optimizations that reduce the cost of container cold-starts [18, 46, 88, 111, 37, 118, 9, 124] are employed, initialization can still take hundreds of milliseconds. However, even this reduced cold-start time is higher than the baseline execution time of a significant fraction of FaaS functions. For example, function execution times in Microsoft Azure were reported to have a median of $900\,\mathrm{ms}$ and a 25th %-ile of $100\,\mathrm{ms}$ [105]. Hence, starting a fresh execution environment for each request would impose impractical overhead.

[6] To date, only an exploit that reuses the `tmp` filesystem (an insecure shared space) was demonstrated in [97].

A more efficient approach would be to fork a copy of a fully initialized process for each function invocation and discard the copy once it terminates, similar to what traditional web servers like Apache [10] support. Unfortunately, however, fork does not work for multi-threaded functions or multi-threaded language runtimes [69] such as NodeJS – the second most used language in FaaS [32].

Accordingly, our focus in this thesis is retrofitting *sequential request isolation* into FaaS, by isolating sequential invocations of the same function within the same execution environment from one another while preserving the correctness of the tenants' functions.

1.4 Thesis Contributions

In this thesis, we propose a conceptual framework for reasoning about the confidentiality and correctness requirements for a function that will be invoked repeatedly – reusing an execution environment – to serve differently privileged callers, as in FaaS. This conceptual framework translates the global confidentiality and correctness properties into local invariants that must be upheld within the function's execution environment.

Our framework looks at the different data flows that may be present in a function. Some of these data flows are confidentiality- and/or correctness-critical. To prevent the leakage of confidential data (*e.g.* end-client input arguments, retrieved data, or credentials) while ensuring the correctness of the application, data flows must be confined (isolated) to their appropriate scopes (*i.e.* confined to a single

request context by default, and be consciously allowed to span multiple request contexts only after code audit).

In FaaS, all data flows that arise while handling an end-client request are expected to be concluded by the end of the request. This ensures that data within a function instance belongs to a single caller at any given time; thus, neither confidentiality nor correctness would be in jeopardy. For such flows, confidentiality is preserved because data lives transiently only during the handling of a request and thus can not be leaked to any subsequent request. Correctness is maintained because platform optimizations that assume statelessness are not employed on function instances actively handling a request, rather on idle instances that are awaiting new requests.

However, runtimes that are provided by the FaaS providers and functions submitted by tenants may not be able to fulfill their side of the contract due to the possibility of bugs that cause data to flow across invocations. Such data flows that lives beyond the context of a single request are the ones that can put confidentiality and/or correctness on the line, and these are the ones we study. *Confidentiality-critical flows* are the flows that stem from (or operate on) a request's *confidential* data and affect the outputs of a *subsequent* request. *Correctness-critical flows*, on the other hand, are the flows that do not stem from a request's confidential data but still affect the outputs of a subsequent request.

To achieve the *sequential request isolation* property within the execution environment, both confidentiality- and correctness-critical flows must be confined to a single request context (never reach the outputs of a subsequent request). While both types of flows must be confined to a single request context, enforcing such

confinement requires different approaches that preserve the different properties. Next, we describe two complementary approaches to isolate sequential requests, thus ensuring confidentiality while preserving correctness.

Approach 1: Confidentiality by design

The first approach exploits the *expected* absence of correctness-critical flows from FaaS functions and proposes a sequential request isolation enforcement system that breaks *all* data flows at each request boundary, thus confining all data flows to a single request context and enforcing confidentiality "by design".

Following this principle, we design and implement Groundhog, a system that enforces confidentiality by implementing a simple, fixed policy: any changes to a function's internal state during the handling of a request are rolled back to a consistent, clean state, free from any confidential data, before another request is handled. Groundhog is a black-box solution that is transparent to both the developer and the provider; it is programming-language agnostic and does not require any changes to the existing code of functions, libraries, language runtimes, or OS kernels but requires provider collaboration to enable it. Groundhog isolates sequential invocations with modest overhead on requests' end-to-end latency and throughput.

Groundhog ensures that the confidentiality requirement is unconditionally met. The correctness property is maintained for FaaS functions that do not retain correctness-critical state (because the rollback restores to a valid and consistent earlier state of the function).

Approach 2: Analyzing Statelessness

The second approach tackles the problem of undesired data flows through program analysis. It employs Dynamic Data Flow Analysis (DDFA)[48, 49] on functions to detect and track data flows that span multiple requests. This approach allows developers to identify and refactor such flows according to the contract requirements. Like most program analysis tools, the analysis is best-effort – it *aids* developers in writing code that observes the confidentiality and correctness requirements but does not enforce either property (the developer ultimately retains responsibility for meeting the confidentiality and correctness requirements in their code).

Building on this idea, we design and implement CtxTainter, a system that relies on information that can be collected via standard DDFA tools. CtxTainter adds information about request boundaries, which is essential for identifying confidentiality- and correctness-critical flows that span multiple requests and outputting them to the developer. Unlike Groundhog, CtxTainter requires active developer participation to review and fix detected confidentiality and correctness violations during the development phase. Because CtxTainter is a development phase tool, it does not require any support or modifications from the platform provider and can be used entirely within the development environment (*e.g.* integrated into an IDE, or run in continuous integration/deployment pipelines).

Ensuring confidentiality while preserving correctness

Combining Groundhog's confidentiality enforcement with CtxTainter's aid to developers in finding correctness-critical flows allows FaaS applications to provide end-clients with confidentiality guarantees while preserving correctness.

1.5 Organization

The rest of this thesis is organized as follows. Chapter 2 presents our conceptual framework for the goals and requirements of sequential request isolation. Chapter 3 presents Groundhog[7], and CtxTainter is presented in Chapter 4[8]. The related work for each system is discussed separately in §3.4 and §4.5, respectively. Chapter 5 summarizes the results and discusses potential directions for future work.

[7] Content mostly derived from the published paper [5], whose technical work and writing were led by me.

[8] This work is presented for the first time in this thesis. The technical work and writing were led by me.

2 Conceptual framework

This chapter introduces the conceptual framework that guides this thesis' work. We start by describing the FaaS execution environment, then discuss and decompose the problem of sequential request isolation. After that, we derive the conditions for achieving sequential request isolation without compromising correctness and finally discuss how our two complementary approaches fit within this framework.

2.1 Execution Environment Reuse

In modern operating systems, programs run in memory-isolated environments (*processes* [15]). The isolation of processes' memory from one another is enforced by the operating system and assisted by the hardware. Each process has exclusive access to a virtual memory address space where static data is initially loaded from the program executable. At runtime, data can be introduced into this isolated environment through communication with the external environment via different interfaces, such as argument passing, environment variables, and system calls that enable inter-process communication, access to file systems, and networking.

Process isolation can be hardened through the use of containers or virtual machines. Containers allow for isolating OS services such as storage and network, while virtual machines isolate all OS services through software and hardware virtualization technologies. Most FaaS platform providers deploy the processes running the functions in containers [4, 82, 11, 89, 45], or in dedicated virtual-machines [7, 3].

FaaS functions are launched in containerized processes in response to events (*e.g.* requests). After a function runs to completion and returns the response, the function instance can be safely terminated to free resources. However, because functions are typically single-purposed and have short execution times [105], the initialization phase (provisioning the execution environment and loading the runtime binary, developer code, and data) constitutes a non-trivial overhead, especially if a new function instance was to be launched for each event. For that reason, cloud providers treat the provisioning of an execution environment as an investment that can be amortized by handling consecutive requests to the same function; once provisioned, each execution environment handles the requests it receives sequentially until the demand drops.

The problem with execution environment reuse is that 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 subsequent one. This thesis focuses on enabling request isolation while preserving correctness at the level of the primitive OS construct for running FaaS functions — within processes.

2.2 The Abstraction

To reason about the confidential information leakage from one request to a subsequent one, we need to study how data flows within the execution environment. Our conceptual framework abstracts the life cycle of a FaaS function's execution environment as an initialization phase followed by a sequence of epochs, each responsible for handling an end-client request to the main FaaS function handler. We call each epoch a *request context*. *Request contexts* serve as an abstraction that draws a conceptual boundary around each function invocation.

During the life cycle of a FaaS function's execution environment, data gets introduced into the execution environment through *data sources* and is externalized out of the execution environment through *data sinks*. A function's processing of a data item all the way from its origin at a data source to its exit from the process at a data sink constitutes a data flow.

Abstracting the function's life cycle to a set of request contexts and data flows allows us to reason about the confidentiality and correctness conditions that must be met within the execution environment.

2.2.1 Data Sources

Execution environments are isolated from the external world. Data can be introduced into the isolated execution environment through three main ways. First, *initialization data* that may be retrieved from remote storage or come integrated within the function, such as default values of variables or parameters

and program constants. This sort of initialization data is static and independent of end-clients' requests. Second, *end-client inputs* that constitute the invocation-specific data provided by, or on behalf of, the end-client (*e.g*. in the form of arguments, credentials, *etc*.); these inputs are considered confidential unless explicitly declassified by the developer, and finally, *per-context freshness* which constitute invocation-specific inputs that must be obtained from the external environment but are independent of the end-client identity or inputs. Examples for per-context freshness inputs are inputs derived from an external source of randomness (a random number that is reused is not random), timestamps (a stale timestamp does not represent time accurately), or even the latest version of data that is stored externally. These inputs must be fresh for each invocation regardless of the end-client identity or input; the conditions under which a particular input becomes stale differ by the input and the use case.

2.2.2 Data Sinks

In principle, a data sink can be any statement that consumes data. However, for the purposes of request isolation, the relevant data sinks are the ones that externalize data outside the execution environment. These data sinks include the results of the function, which are returned to the end-client as a response, passed to other services for applying further operations, or simply persisted on external storage for later lookup. These different points at which data can be externalized and exit the isolated execution environment represent the points of risk where confidential data could be leaked, or correctness violations could become externally visible.

2.2.3 Data Flows

Data flows represent the life cycle of a data item within the execution environment from its origin at a data source, throughout all the processing steps it undergoes, and until its exit from the execution environment at a data sink. Each data item can be part of one or more data flows. For reasoning about confidentialiy and correctness, we need not analyze all the data flows as some are irrelevant.

For instance, data flows that arise and conclude within a request context are not of interest because they do not pose a risk to either confidentiality or correctness. Such flows do not risk confidential data leakage because the data flow arises and concludes while handling a single end-client. Similarly, correctness is not jeopardized because the function processing is not impacted by the platform optimizations that happen before or after a request.

On the other hand, data flows that cross a request context boundary might be of interest. Here we consider two classes of such flows:

1. **Confidentiality-critical flows** are all flows of data that stem from (or operate on) a request's *confidential* data sources and reach the data sinks of a subsequent request.

2. **Correctness-critical flows** are all flows of data that do not stem from a request's confidential data but still reach the data sinks of a subsequent request.

2.2.4 External data flows

Although the confidentiality and correctness conditions rule out confidentiality- and correctness-critical flows of data across requests, some functionality might require maintaining data across requests. For example, a function might aggregate confidential data from a large pool of users to train a machine-learning model. For such functions, the developer is expected to declassify the information by externalizing the aggregation process to a secure service out of the function's ephemeral execution environment. Active externalization of declassified data is necessary if the tenant does not want to lose aggregated data as part of the normal FaaS platforms' continuous recycling of the "ephemeral" environments.

Similarly, per-context freshness inputs, such as PRNs, must rely on external stateful services rather than linked libraries to prevent changing a function's semantics if it undergoes optimizations by the platform while being idle after a request is handled.

2.3 Confidentiality and Correctness

Commercial FaaS platforms expect functions to be stateless, which, if true in practice, would imply that the FaaS functions are free from confidentiality- and correctness-critical flows because no state would be retained beyond a request context. However, as discussed in §1.2 and §1.3, ensuring the statelessness of FaaS functions is non-trivial. A FaaS function, or one of the third-party libraries or the runtime it relies on, might have a bug or may have been originally designed

for a stateful computing paradigm and may retain state crucial for confidentiality and/or correctness.

The FaaS contract describes the responsibilities of the cloud provider and the tenant. According to the standard FaaS shared responsibility model, the provider and the tenant should be responsible for the data confidentiality and correctness in their respective components. However, some of the components, such as the execution environment, runs code contributed both by the provider and the tenant.[1] As we explain in Chapter 3, the provider can effectively and efficiently enforce data confidentiality across requests reaching the execution environment. By doing that, the provider alleviates the burden of ensuring confidentiality off of the tenant's shoulders. The correctness of the FaaS function's logic, however, remains the tenant's responsibility.

Next, we discuss the tenants' and the providers' sides of the contract. If each party fulfills their side of the contract, FaaS functions will operate with guaranteed confidentiality without compromising the correctness.

2.3.1 The Tenant's Side of the FaaS Contract

The commercial FaaS shared responsibility model requires that both the tenant and the provider be responsible for meeting the confidentiality and correctness requirements in the code/infrastructure they provide. However, because, as discussed earlier and shown in Chapter 3, the provider can transparently enforce

[1]The language runtime and IO handlers are provided by the FaaS provider, while the tenant provides the function handler and the helper libraries.

confidentiality for both the provider's and tenant's contributed code, the tenant is only responsible for meeting the correctness condition discussed next.

Correctness The tenant must only ensure one condition: The absence of *correctness-critical* flows. The condition we propose is *weaker* than the statelessness requirement expected from the tenants of commercial platforms, which requires eliminating *all flows* that cross a request context boundary, necessarily requiring that all code submitted by the tenant to be verifiably free from any bugs that may retain state across invocations.

2.3.2 The Provider's Side of the FaaS Contract

Confidentiality The confidentiality requirement stipulates that no request (FaaS function activation) should breach the data confidentiality of any other request. This property is enforced globally by implementing measures that isolate instances of functions from one another. Because, as common in commercial FaaS offerings, invocations of different functions are already isolated from each other, the remaining requirement is sequential request isolation, which prevents leaks of confidential data from one request handled by an execution environment to the next request handled by the same environment. This isolation translates to *sequential request isolation*, which prevents confidential data leakage from one request context to a subsequent one.

Correctness The correctness requirement stipulates that for functions that adhere to their side of the contract (§2.3.1), the semantics of the function (the

tenant's logic responsible for handling a single request) when run under the FaaS provider infrastructure should be equivalent to its semantics when invoked once and terminated (*i.e.* any optimizations or features introduced by the platform must not alter the semantics of the tenant's function). Additionally, and similar to the tenant, the provider must ensure that the language runtimes and IO wrappers contributed by the provider do not retain correctness-critical state across invocations.

2.4 Fulfilling the Contract

In this thesis, we propose two broad, complementary approaches that help the provider and the tenant fulfill their respective sides of the contract.

Our first approach, Groundhog, proposes a request-isolation enforcement system that, when enabled by the provider, breaks all data flows at the request context boundary, preventing any data from leaking from one request context to a subsequent one. This approach works for FaaS functions that adhere to their side of the contract (§2.3.1), where functions are expected to be free from any correctness-critical flows.

Our second approach, CtxTainter, is based on software analysis and assists function developers in vetting their code and the libraries they rely on, such that correctness-critical flows that cross a request context boundary can be detected

and *selectively* eliminated.[2] As opposed to Groundhog, which silently breaks all flows at the request boundary, this approach allows the tenant/developer to meet the correctness conditions during the development phase.[3]

When used together, Groundhog and CtxTainter guarantee confidentiality and aid the developer in preserving the correctness properties: Groundhog can be used to enforce confidentiality, and CtxTainter can be used as a complementary tool to help the developer identify possible correctness violations and update their function's code to avoid them, thus reconciling confidentiality and correctness.

[2]There are fundamental limitations in software-analysis that prevent having both a sound and a complete analysis simultaneously. Accordingly, only best-effort guarantees are possible.

[3]In principle, this approach can be used by developers (without FaaS-provider collaboration) as an aid to also attain confidentiality. However, because security is non-negotiable and this approach only aids the developer without promising soundness or completeness, we prefer to limit CtxTainter's scope to correctness-critical flows only.

3

Groundhog: Confidentiality by Design

This chapter presents our first key technique for achieving sequential request isolation in FaaS, Groundhog. Groundhog is a system that enforces confidentiality by implementing a simple, fixed policy: after each request, any changes to the function's internal state are rolled back to a consistent, clean state, free from any confidential data. This policy implies that all data flows are broken at each request boundary, thus confining all flows to a single request context and enforcing confidentiality "by design" for FaaS platforms where stateless functions and ephemeral execution environments are the norm.

Groundhog is designed with the goal of transparently retrofitting lightweight sequential request isolation to existing FaaS platforms that already use containers[1] to isolate different functions. Importantly, Groundhog allows the safe reuse of containers across requests to the same function, thus avoiding the per-activation container re-initialization cost of the trivial solution described in §1.3. Groundhog is independent of the language, runtime, or libraries used to implement functions,

[1]This work is described in the context of FaaS platforms that already use containers to isolate different functions from each other, but similar design principles should apply to VMM-based isolation.

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.

To implement a general-purpose lightweight solution, Groundhog exploits two properties of FaaS platforms: (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.[2] 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 guarantees confidentiality because the restoration ensures that no data can leak from one activation to a subsequent one. Groundhog is efficient because the cost of restoring the state is 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

[2]If functions intentionally retain non-confidential state between invocations, then using Groundhog can break correctness. In such scenarios, developers should externalize the state that should be retained. Our complementary system, CtxTainter, presented in Chapter 4 can assist developers in detecting the presence of such retained state and report candidate state for safe externalization.

restores state *between activations of a function*, and therefore, does not significantly affect the 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 [104], pyperformance [122], and PolyBench/C [76] 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 chapter 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,[3] 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, or OS kernels.

3. An experimental evaluation of Groundhog on functions from the FaaSProfiler [104], pyperformance [122], and PolyBench/C [76] benchmarks within the OpenWhisk FaaS platform, which demonstrates that Groundhog provides

[3]Groundhog is open-source and is available at [6].

sequential request isolation with low to modest overhead on function latency and peak throughput.

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

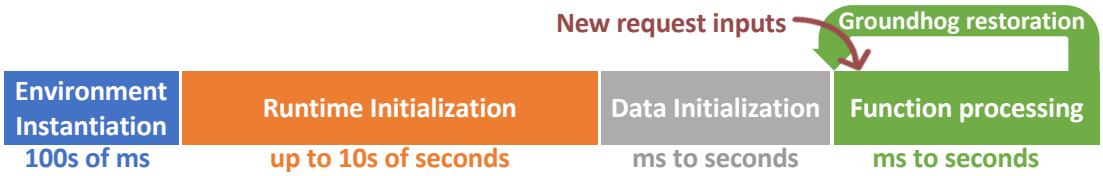


Figure 3.1.: *Groundhog container life cycle*

Fig. 3.1 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 $3.7\,\mathrm{ms}$ (10p: $0.7\,\mathrm{ms}$, 25p: $1\,\mathrm{ms}$, 75p: $5.4\,\mathrm{ms}$, 90p: $13\,\mathrm{ms}$). 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.

3.1.1 Insights

Groundhog's design exploits two key properties of the FaaS paradigm (as implemented by major FaaS providers) 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, FaaS platforms create separate containers to concurrently execute activations of different functions or handle simultaneous 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

a persistent state, functions must instead rely on external or platform services such as a key-value store or a database backend [17].

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 the container. Since the platform is free to shut down an idle container at any time, functions must not rely on the persistence of such global state for correctness.[4]

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 instead use explicit external services or platform facilities for maintaining persistent state such as PRNGs. Data loss and/or functionality anomalies may arise if a function implementation or a library it depends on relies on an internal state being maintained across invocations, because the FaaS runtime may terminate or refresh a container between invocations.

[4]Opportunistic caching of end-client-specific data should be possible through per-host caches that are subject to access control based on the end-clients' identity/privileges. However, we are not aware of any provider that offer this kind of caching.

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 the global state but before the first function activation, thereby ensuring efficient sequential request isolation.

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

Small write sets Programs, including FaaS functions, written in managed languages often consume a substantial amount of memory due to the language runtime overhead. However, because FaaS functions are typically single-purposed, only a small proportion of the memory 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 memory 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 [118], 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 [6].

Groundhog's design and implementation, which is discussed in §3.2, was guided by these three key insights.

3.1.2 Design alternatives

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 [21], compiler instrumentation techniques [123], or runtimes [127] 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 Reliance on a custom snapshot/restore solution has been explored in prior work [9, 37, 118, 111, 108, 67] 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.

3.1.3 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.

3.2 Groundhog Design and Implementation

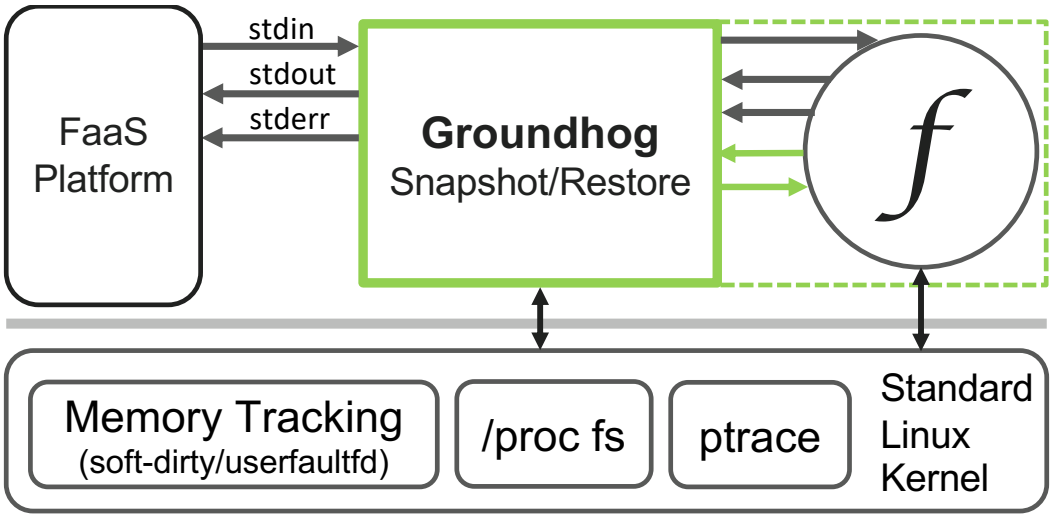


Figure 3.2.: *Groundhog Architecture: (1) The manager (solid green box), (2) The function process (f). Groundhog relies on standard Linux kernel utilities.*

Groundhog uses a novel, lightweight, in-memory process snapshot/restore facility that achieves low restore times with minimal impact on the end-to-end request latency under normal workloads. Fig. 3.2 illustrates Groundhog's architecture. The Groundhog *manager* process (solid green box) runs within an OS container alongside the function process, and is responsible for enforcing request isolation. 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 modify only 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 needs to restore only those pages that were modified during 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.

3.2.1 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 the 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 §3.2.2. 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 [73]. We note that the arguments provided to a dummy invocation may affect performance but not the confidentiality guarantees.

3.2.2 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` [70]; (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.

3.2.3 Tracking state modifications

Groundhog uses the standard *Soft-Dirty Bits* (SD) feature of the Linux kernel [71],[5] which provides a page-granular, lightweight approach to tracking memory modifi-

[5]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 [83].

cations. 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) [72][6] feature for memory tracking and prototyped this alternative; however, we found UFFD to have a significantly higher overhead compared to SD-bits due to the frequent context switches to user space for fault handling.[7] UFFD was marginally faster than soft-dirty bits only when the number of dirtied pages was close to zero.

3.2.4 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` [70, 39, 111]. The

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

[7]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.

manager restores `brk`, removes added memory regions, remaps removed memory regions, zeroes the grown 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 the restoration is completed, 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.

3.2.5 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 confidentiality, it can add an overhead of copying request input/out-

puts 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.[8])

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, for 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.[9] In principle, Groundhog could be extended to intercept fork syscalls and track the

[8]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.

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

child processes through standard ptrace tracking options. Similarly, through standard ptrace options, Groundhog can be extended to intercept and adjust syscalls (such as seccomp) that limit the availability of standard Linux kernel facilities required by Groundhog.

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.

3.3 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 a 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.

3.3.1 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 [90] by modifying OpenWhisk's container runtimes for Python and Node.js to include Groundhog. Additionally, 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). The actionloop-proxy 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 confidentiality by blocking inputs until the function's process is restored to a clean state.[10]

[10] 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 §3.2.5.

For the FaaS OpenWhisk-python-runtime, we added 15 Lines of Code (LoC) to the FaaS-provider wrapper 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 §3.2.5.

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 [35] parsing library (2.5K LoC) was also added.

In general, integrating Groundhog with a FaaS platform that forwards 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 request retrieval and response sending, and by updating the FaaS runtime to retrieve the request from and send the response to Groundhog. Alternatively, a signaling mechanism between Groundhog and the FaaS platform can be implemented as outlined in §3.2.5.

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 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 $\textbf{GH}_{\text{NOP}}$, 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 §3.2.5). The configuration also helps quantify 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. First, we implement a fork-based request isolation method, **FORK**, which is applicable to single-threaded applications and runtimes only. Next, 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.

3.3.2 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.[11] We evaluate both the *in-function* overheads that are on the critical path of function

[11] 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.

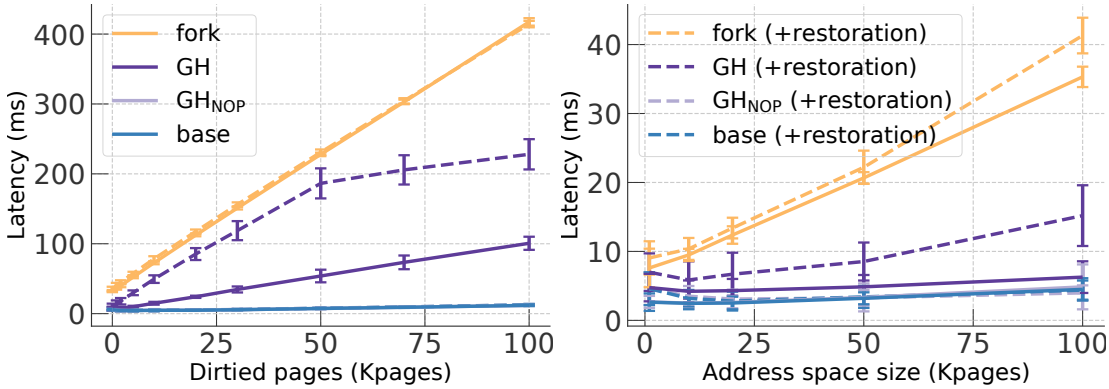


Figure 3.3.: *Function latencies varying the number of dirtied pages (left) and the address space size (right). Different colors represent different request isolation methods (or no isolation for* BASE*). Solid lines are latencies with in-function overhead but not restoration overhead, while dashed lines include both. (The lines of* BASE *and* GH$_{NOP}$ *coincide visually in the figure.)*

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 §3.3.5.

Microbenchmark. We implement a simple single-threaded 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 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.

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 restora-

tion before the next request arrives, so measurements for the low-load workload capture only the in-function overheads.

Results. The solid lines in Fig. 3.3 (left) 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 ran a variant of the experiment where we fixed the number of dirtied pages to 1K and varied the address space size from 1K to 100K pages. The solid lines in Fig. 3.3 (right) show the function latency as we vary the address space size. We observe that Groundhog's(GH) 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 in this experiment.

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. 3.3 (left) 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. 3.3 (right) 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.

Comparison to Fork

A potential alternative to our lightweight restoration is to use copy-on-write techniques such as fork (§3.1.2). 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. 3.3 (left) 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 on the critical path.

Fig. 3.3 (right) 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. The first access of a page in the new process can additionally require lazy creation of physical page table entries, depending on the memory layout of the program.

3.3.3 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. We then provide an illustrative comparison to a fork-based implementation 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.

Benchmarks. We evaluate 58 functions across three benchmarks and three languages: 22 Python functions from the pyperformance benchmark [122], 23 C

functions from PolyBench/C [76], and 13 functions (6 Python, 7 Node.js) from the FaaSProfiler benchmark suite [104].

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.

Measuring Latency. To measure latency, we deploy a 4-core VM with a single FaaS function container that is limited to at most one core. In a separate VM on the same machine,[12] a closed-loop client submits requests one-at-a-time. This workload is similar to the *low-load* setting from §3.3.2 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

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

are chosen empirically to maximize throughput). This workload is similar to the *high-load* setting from §3.3.2, as it ensures the FaaS platform is always saturated with requests. Throughput measurements thus account for Groundhog's full overheads, including both the in-function and 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 Appendix A. 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.

Comparison to the Baseline

The upper graphs of Fig. 3.4 (rel. E2E lat.) show the relative 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 baseline and >1 represents 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

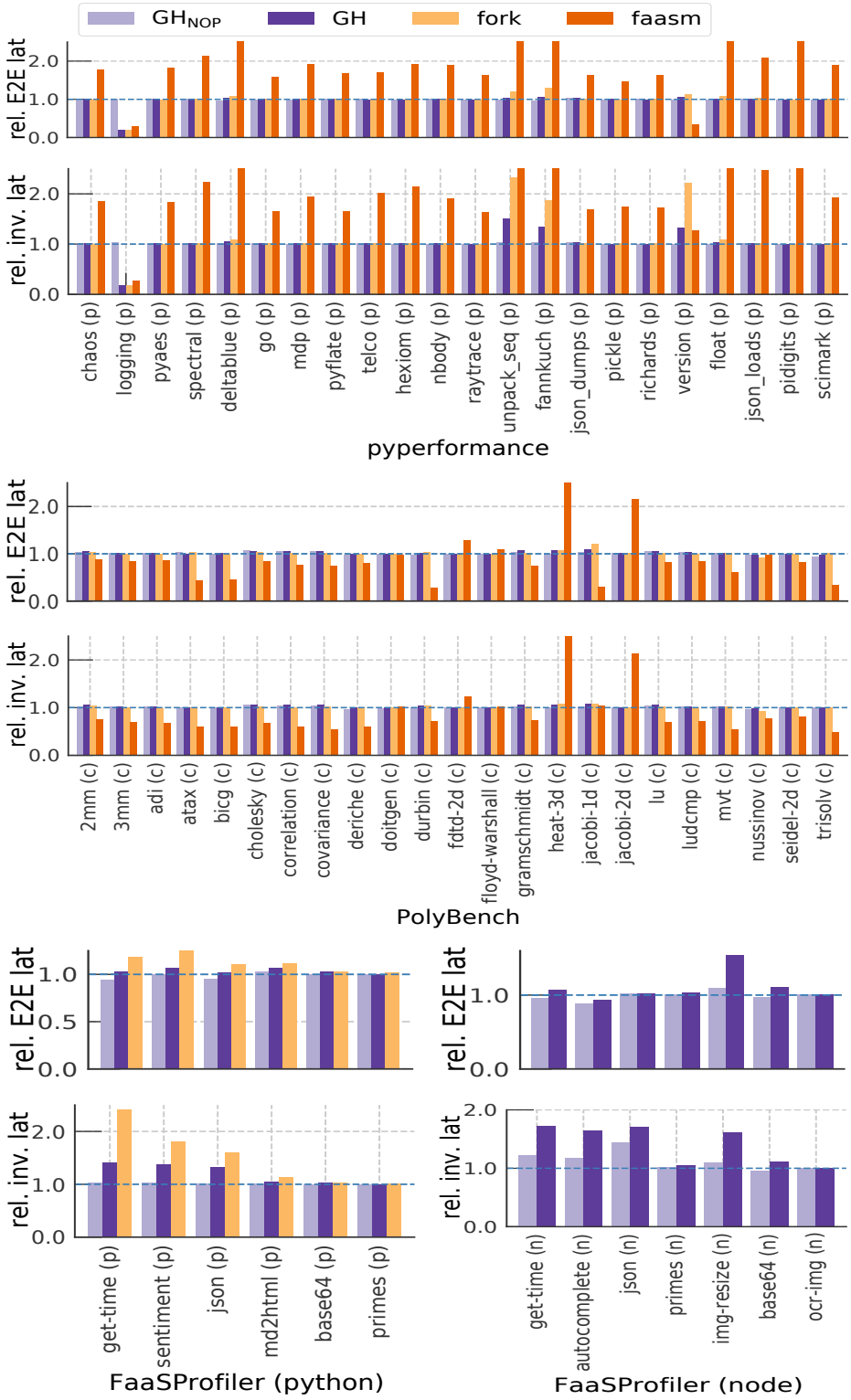


Figure 3.4.: *Relative end-to-end latency and invoker-measured latency of* GH, GH$_{NOP}$, FORK, *and* FAASM *compared to the insecure baseline* BASE. *Figures are capped at 2.5X the baseline. Detailed numbers are in Appendix A. The symbols (p), (c), and (n) denote Python, C, and Node.js benchmarks, respectively.* ***Lower numbers are better.***

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. 3.4 (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. 3.4 (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 to follow the actionloop-proxy design. This relative 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). There are several ways to tackle ths problem. For instance, time inputs could be virtualized such

that the process restoration resets the time to the original time of the snapshot. Another approach would be to modify the garbage collection behavior to be time-independent. This problem is actually a broader problem in the space of snapshot and restore techniques, where data (in this case, the timestamp) can become stale by the time a restoration happens. A comprehensive treatment of this problem is beyond the scope of Groundhog.[13]

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. 3.5 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. 3.5 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.

[13]This maintained state in the Node.js runtime does not impact confidentiality or correctness but affects performance. In Chapter 4, we propose CtxTainter, a tool that aids developers in detecting flows that span multiple FaaS requests. Runtime developers/providers who wish to ensure their language runtimes are stateless could also employ the same approach.

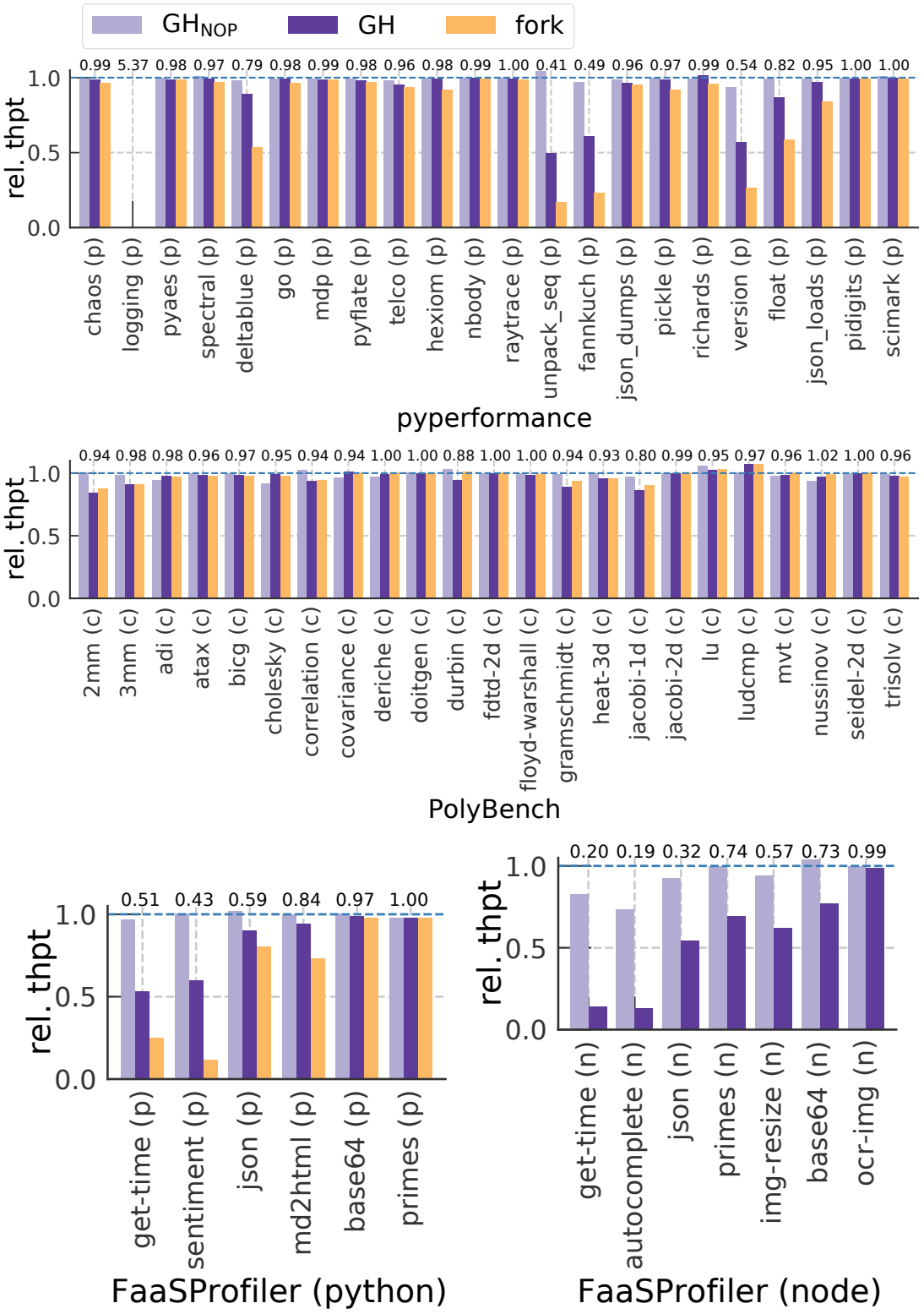


Figure 3.5.: *Relative throughput of* GH, GH_{NOP}, *and* FORK *compared to the insecure baseline* BASE. *Detailed numbers are in Appendix A. The symbols (p), (c), and (n) denote Python, C and Node.js benchmarks, respectively.* ***Higher numbers are better.***

GH's Node.js restoration overheads tend to be higher than other runtimes as Node.js's runtime performs aggressive memory layout changes[14] (see Fig. 3.8 for the restoration overheads of selected benchmarks). Across all benchmarks, the median and 95th-percentile throughput reductions are 2.5% and 49.6%, respectively.

Comparison to Fork

We also provide a comparison to the FORK alternative described in §3.3.2. Recall that `fork` only applies to single-threaded functions; thus, we cannot provide measurements for the Node.js runtime.

Fig. 3.4 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. 3.5 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.

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 [106],

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

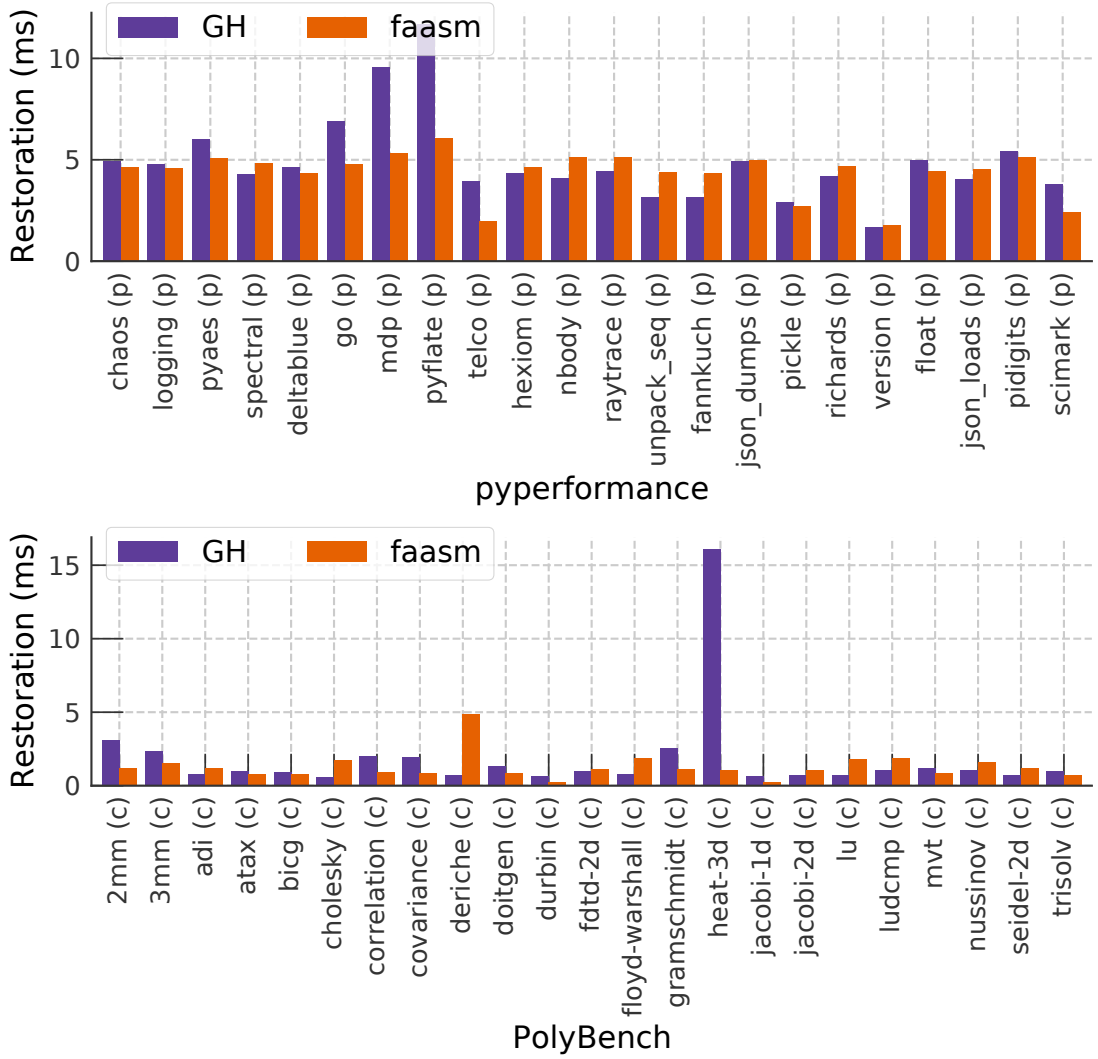


Figure 3.6.: *Restoration duration (off the critical path) of* GH, *and* FAASM. *The symbols (p) and (c) denote Python and C.*

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.

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 [106]. We rely on FAASM's microbenchmarking infrastructure that reports both the overall latency (end-to-end and invoker) and the restoration (reset) cost.

Fig. 3.4 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. 3.6). 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 [106, 47, 54].)

The same trends continue to manifest in throughput measurements, where FAASM has lower throughput than GH on most pyperformance functions, and higher

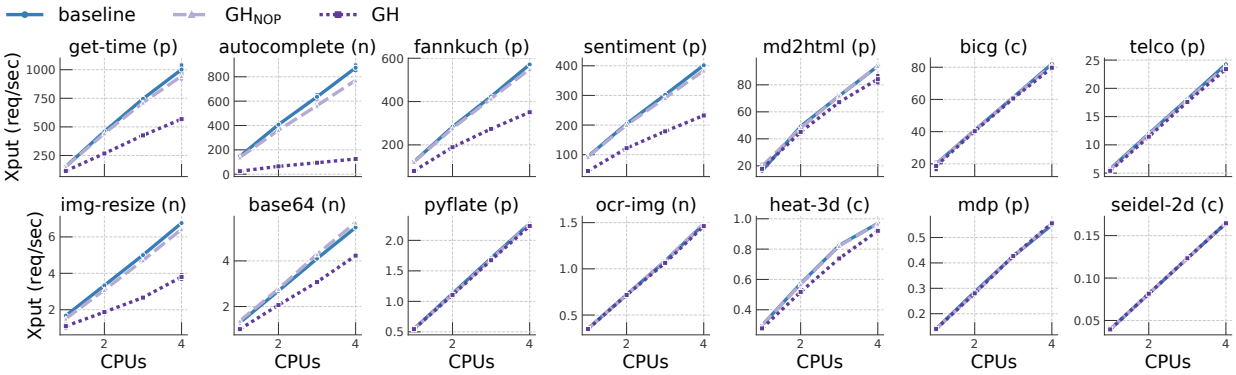


Figure 3.7.: *Throughput scaling with number of cores. Error bars (minute) represent the standard deviation across 6 runs.*

throughput than GH on most PolyBench functions. We omit the detailed throughput comparison from the discussion as the results entangle many variables with the isolation mechanism, such as the differences in the platforms' internal components and the runtimes (native vs WebAssembly). However, for reference, the numbers can be found in Appendix A.

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

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. 3.7 shows absolute

throughputs as a function of the number of available cores for a subset of 14 representative benchmarks of varying durations, 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.

3.3.4 Deconstructing restoration overheads

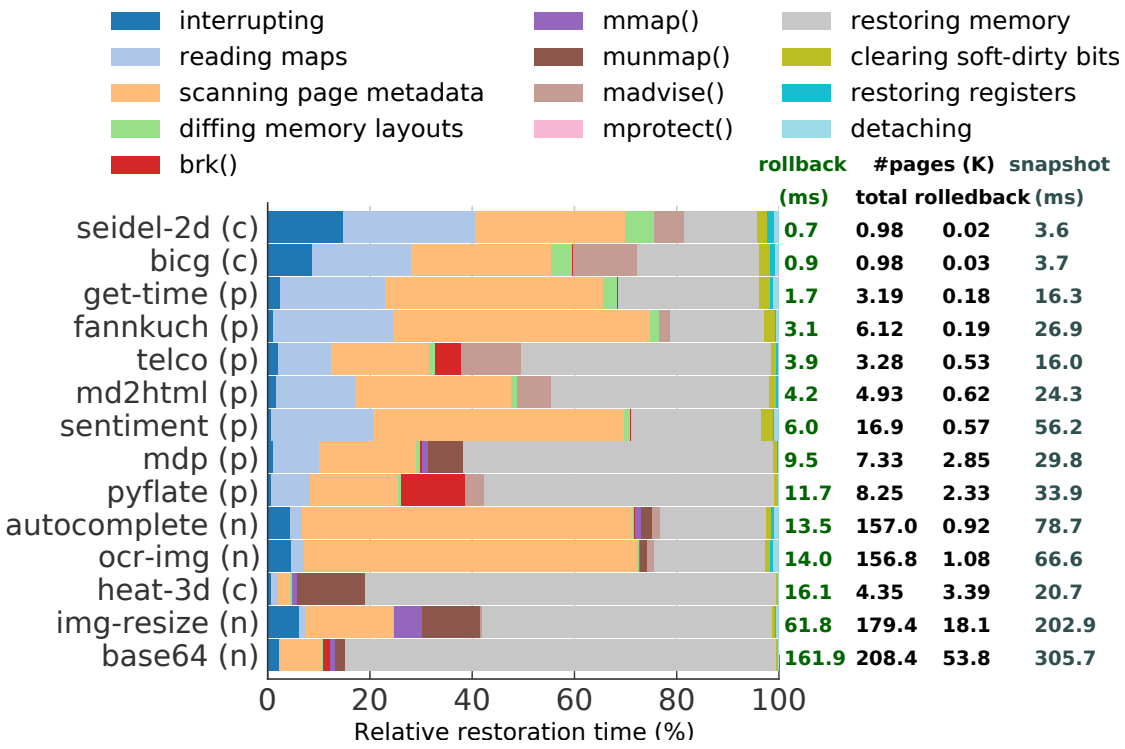


Figure 3.8.: *Restoration overhead (deconstructed) and the one-time snapshotting overhead for a subset of benchmarks.*

Groundhog restoration involves several steps that we outlined in §3.2.4. In this section we break down the cost of restoration for 14 representative benchmarks with varying durations, number of mapped pages, and number of dirtied pages. 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. Lastly, the cost of restoring the contents of pages depends on the number of pages dirtied or unmapped during an invocation.

Fig. 3.8 shows these costs normalized to the total restoration cost for our 14 representative functions shown in the throughput scaling experiment. 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 §3.3.5.) 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 §3.2.3, optimizations can make the costs correlate to the number of dirtied pages instead.)

3.3.5 Snapshotting overhead

The rightmost column of Fig. 3.8 outlines Groundhog's snapshotting latency overhead for the same 14 functions that we used in the throughput scaling experiment. 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 [37], REAP [118], FaaSnap [9], Replayable [124], Prebaking [108], Pagurus [67]) 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.

3.3.6 Dummy Requests

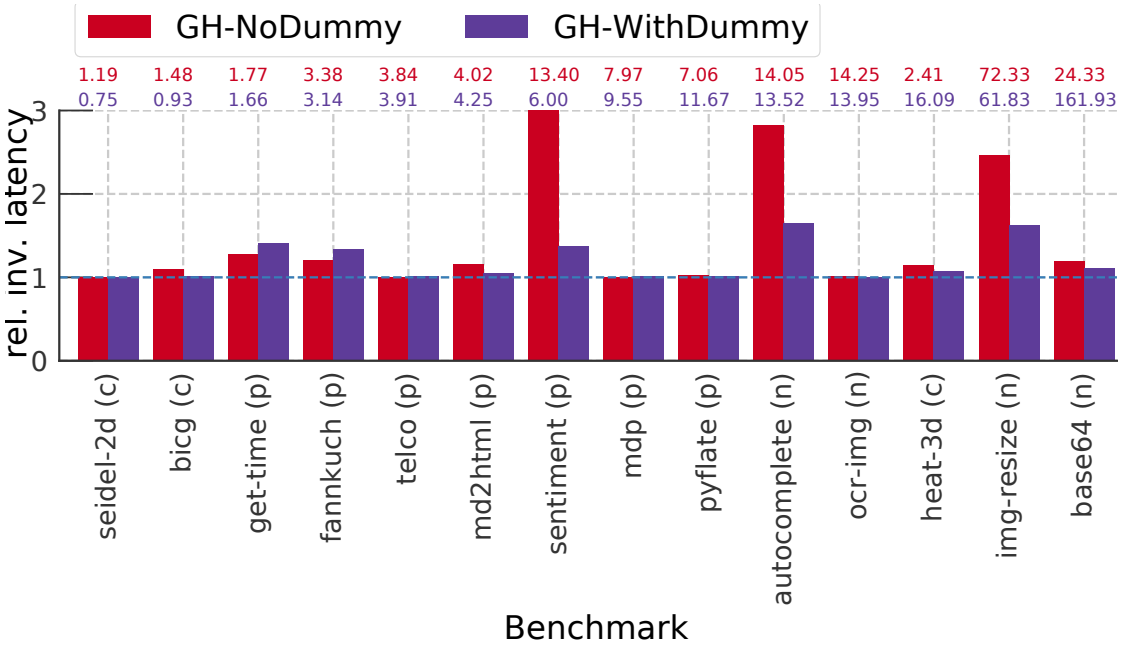


Figure 3.9.: *Latency of function invocations, normalized to the insecure warm baseline* BASE*, with and without executing a dummy request prior to taking the snapshot. The off-critical-path restore time (in ms) is shown over each benchmark. The symbols (p), (c), and (n) denote Python, C, and Node.js 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.***

Groundhog optimizes for the on-critical path latency and chooses to take a snapshot after a dummy request is processed 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 [73]. On a representative set of benchmarks, Fig. 3.9 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.

3.4 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 [37]. Even for single-threaded functions, a fork-based approach is less performant than Groundhog (see §3.3) 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) [74], 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 [8, 12], maintaining a pool of containers that can be repurposed [84, 111], maintaining partially initialized runtimes with loaded libraries as in SOCK [88], relaxing isolation between functions by allowing functions from the same app developer to share containers (SAND [4], Azure [82]), and starting from slim container images and adding non-essential functions only when needed (CNTR [115]). 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 [124], 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. However, 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 reuse an existing clean environment as Groundhog does.

Snapshotting techniques based on CRIU [31, 26, 120, 30, 98] 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 [121] 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 [23] 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 [37] trades function-start latency for in-function latency using a lazy restoration that incurs page faults. REAP [118] reduces the cost of these page faults by eagerly pre-fetching 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 $\sim 0.5\,\mathrm{ms}$ and an equivalent Python function in $\sim 1.7\,\mathrm{ms}$ off the critical path. Systems such as Catalyzer [37] 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 [9] 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 [29], Faastly [41, 117], and FAASM [106] solve the cold-start problem by relying on software-fault isolation (SFI) using V8 isolates and WebAssembly [127]. 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, on a copy-on-write basis. 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 Figure 3.3.3. Unlike Groundhog, this technique is limited to languages, runtimes, and threading models that can be compiled to WebAssembly.

3.5 Summary

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.

4

CtxTainter: Analyzing Statelessness

In FaaS, or any composable system, different components agree on and implement a contract to ensure that they can securely and correctly interact with one another. The FaaS paradigm, as currently implemented by commercial platform providers, expects functions to be *stateless*. As discussed earlier in §1.1, statelessness allows providers to safely implement performance optimizations without the need to take any special measures for ensuring the confidentiality and the correctness properties at the level of individual activations of a function, but rather at the level of function instances.

While developers may *try* to design and implement stateless functions, bugs in a function's implementation — or a third-party library/runtime it depends on — may cause a leak of information from one activation of the function to a subsequent one, violating the FaaS confidentiality contract. Similarly, the correctness contract is violated if functions rely on off-the-shelf third-party libraries/runtimes that were not developed with statelessness in mind because the state might be retained for correctness reasons. This retained state may be manipulated by common platform optimizations such as execution environment reuse and

rapid execution environment provisioning by cloning running or checkpointed instances.[1]

This chapter introduces CtxTainter, our proposed tool that assists developers in analyzing statelessness by detecting flows of data that cross a request context boundary. CtxTainter extends standard Dynamic Data Flow Analysis (DDFA) techniques[48, 49] with the ability to reason about request contexts and identify and report data flows that span multiple request contexts. As opposed to standard request-boundary-agnostic DDFA techniques, CtxTainter can be used by developers in scenarios where simultaneously reasoning about data flows and request context boundaries is needed, as in the case of FaaS.

CtxTainter's ability to track the interaction of data flows with request context boundaries can assist developers in writing functions that meet the contract's statelessness specifications by allowing developers to identify *correctness-critical*

[1] In addition to the traditional use of checkpoint and restoration techniques in debugging and fault-tolerance [110, 112, 51, 59, 96, 116, 130, 43, 63, 57, 107, 94, 123, 24], migrating applications [91, 77, 31] and intermittent computing [119, 79], C/R is now commonly used in modern cloud offerings, like FaaS, for accelerating function starts and rapid function out-scaling [7, 22, 37, 118, 9, 106]. Some FaaS providers checkpoint initialized function environments and reuse the checkpointed state to rapidly start new instances. If the functions are not stateless, such reuse of initialized state might break intended data flows or introduce undesired flows that cross a request context boundary, thus changing function semantics and potentially breaking correctness (and, in some cases, subsequently confidentiality and integrity [22]). For instance, pseudo-random number generators (PRNGs) need to maintain state to be able to generate seemingly fresh random numbers. If the checkpoint is taken after the PRNG is initialized (seeded), then the stream of random numbers will be the same after each restoration to the checkpoint. Correctness (and potentially security) problems arising from the interplay between PRNGs and C/R techniques have been thoroughly discussed in [22]. This correctness problem is an instance of a class of problems arising due to caching data from a source of per-context freshness (described in §2.2.1) and reusing the cached data in subsequent request contexts. Other manifestations of this problem can be seen in any function that relies on chained behavior (such as counters) where the chaining spans multiple request contexts. Similarly, the problem manifests with the use of successive timestamps to measure elapsed time (if a C/R operation separates the timestamps), or with the temporary caching of data that is otherwise assumed to be refreshed with every new environment initialization.

flows that cross the request context boundary and refactor the code to eliminate them at development time. Similarly, If the provider supports only instance-level isolation and does not enforce sequential request isolation, then CtxTainter can also be used to help identify *confidentiality-critical* flows that live beyond a request context and help the developer ensure data confidentiality. However, this is not CtxTainter's intended use-case as discussed in §2.4.

4.1 Design Preliminaries

As discussed in Chapter 2, we abstract FaaS functions to a set of data flows, each capturing the life cycle of a data item from the data source throughout the read, transform, and write operations it passes through until the data item reaches a sink. Some of these data flows arise and conclude within a single request context, thus having no bearing on the function's contract with the provider. Data flows that span multiple request contexts are the ones that can jeopardize meeting the contract. To help the developer meet the contract, flows that cross the request boundary must be detected and reported to the developer.

The goal of CtxTainter is to ① detect *as many data flows as possible* that cross the request context boundary and report them to the developer for detailed inspection. ② support developers in debugging violations by providing full data flow traces for developer-specified sources of interest. ③ Once the developer has indicated a source of interest, CtxTainter should be able to run the analysis without requiring the developer to further annotate or instrument their code.

4.1.1 Design Dimensions

Building a data flow analysis tool that fulfills the aforementioned goals requires making design decisions along three main dimensions. The first dimension is the broad data flow analysis approach, whether the analysis should be done statically or dynamically. Second, the abstraction level at which the analysis operates, whether at the source code, at an intermediate representation, or at the machine code level. Finally, how precise should the analysis be, as there is usually a trade-off between the precision and speed of the analysis. Next, we explore these dimensions and discuss our design decisions.

Data Flow Analysis Approaches

There are several standard approaches for analyzing data flows [36, 48, 49, 100, 50, 99, 16, 101]. At their core, they share a simple idea, *sources* of data are monitored. When data is introduced through one of these *sources* (c.f. §2.2.1), a label is attached to the data, and the label is propagated with the data through all read, write, and transform steps that involve the data item. Eventually, the data may reach outputs of interest (*sinks*, c.f. §2.2.2). A policy determines whether the flow of any given data item to a given sink is valid. The policy uses the label of the data item and reports invalid flows to the developer or to a system administrator. Data flow analysis can be done statically, dynamically, or through hybrid approaches. Here, we focus on the broad static and dynamic approaches.

Static approaches use static analysis tools such as the industry-standard CodeQL [42] and PySa [40] tools, which build data flow graphs of the program based on a

reachability analysis without the need to run the program. Being able to perform the analysis without the need to run the program makes static analysis-based approaches a low-cost option for scanning programs for bugs and vulnerabilities. However, static analysis tools cannot accurately detect dynamically resolved data flows without explicit hints/annotations from the developer, especially in dynamic languages like Python or JavaScript – two of the most common languages in the FaaS paradigm [32].

Dynamic data flow analysis tools [113, 62, 28, 25, 101, 85], on the other hand, track the flow of the data in the program as it executes, making it more capable of capturing flows that go through dynamically resolved execution paths. Dynamic data flow analysis typically involves instrumenting the code or the runtime to record the flow of data or using a debugger to step through the program execution steps and observe the values of variables and how they change over time. While dynamic data flow analysis is more precise, it can only detect problematic flows in paths that have been executed – If there is a bug/problem in an unexplored path, it will not be detected.

Both static and dynamic data flow analysis systems are commonly used to detect (and prevent) vulnerabilities. In such scenarios, they aim to ensure that sensitive data introduced by *unsafe* sources do not reach potentially vulnerable sinks. For example, raw user input may be considered an unsafe source of data; thus, should not be trusted to be directly passed to a database query engine. Once the unsafe raw user input is received, a label is attached to it (marking the variable holding the data as tainted) to prevent it from reaching a potentially vulnerable sink, such as a database query API. If the tainted raw user input reaches a function

that creates the code of a database query, then there is a risk of a code injection vulnerability. Data flow analysis systems also introduce functionality to drop the taint off of the data. For instance, if the raw user input goes through a function that *sanitizes* the input, then the resulting data is *endorsed* (*i.e*. its taint is dropped) and the sanitized input can be safely consumed by the sinks. To summarize, data flow analysis systems track data from *sources*, apply the propagation rules (one of which is *endorsement*) till the data reaches the *sinks* where the taint is evaluated.

CtxTainter checks a simple policy: A flow to a sink is valid if and only if it does not cross a request context boundary. No existing framework provides all the required machinery to allow accurate identification of data flows that *cross a request context boundary*. For FaaS, building a tool on top of dynamic data flow analysis is reasonable because most functions are written in dynamic languages which are more accurately analyzed by dynamic rather than static analysis techniques. Moreover, adding the reasoning about request contexts is simpler in DDFA tools as the request context boundary will be encountered during the execution, and information about this boundary can be easily extracted.

Execution Metadata Extraction

Performing data flow analysis requires identifying the relations between the different program constructs and variables. This step of extracting metadata can be done on the source-code (the abstract syntax tree (AST)), the intermediate representation (*e.g*. byte Code), or the machine code. Doing the data flow analysis at each abstraction level has its advantages and disadvantages. For development

and debugging support purposes, the analysis is typically done at the source-code level as the source-code is readily available and the data flow results can be directly mapped to locations in the source code.

One of the important data flow analysis techniques that allows for identifying the relationship between the program's variables is the construction of variable relation sets. In its simplest form, this technique requires (1) identifying the operands and destination (if any) of each statement, and (2) deciding whether the operation would result in data being propagated from each operand to the destination or not. (1) is generally known in the literature [48, 49, 101, 81, 66] as the `DEF/USE` (or *DEF/Ref)* sets of the statement, where `DEF` refers to the *def*initions of new or re-assigned variables and `USE` *(or Ref)* refers to the *usage (or referencing)* of data from a source or from previously defined variables; (2) is known as applying the propagation rules.

For compiled languages, it is possible to construct the `DEF/USE` sets of the variables statically at compile time. However, for dynamic languages, which are typically used in FaaS, the effectiveness of static analysis tools is limited due to the dynamic features of the languages [129].

When considering dynamic data flow analysis, there are two broad ways to extract the data from the execution: ① By instrumenting the interpreter and augmenting it with data structures to maintain additional metadata (like taint) about the data items processed [61, 80], or ② by injecting hooks into the runtime allowing data collection at vantage points within the execution. Such hooks can be set up through a debugging library [129, 66], or through code-rewriting to wrap operations of interest to call the hooks that collect the metadata [38].

CtxTainter collects data through a debugger interface hook. Relying on the debugger interface hooks allows the metadata extraction logic to be concise and well-contained in one module rather than scattered across the interpreter implementation, which facilitates extending and enhancing the logic or applying it to other programming languages. Moreover, the debugger interface should be stable against changes to the interpreter implementation, thus avoiding long-term maintenance costs. Additionally, the debugging interface typically provides rich enough information about the executed statement and provides a view into the interpreter state, which is sufficient to collect all required metadata.

Analysis Precision

There are several design aspects that influence the precision of data flow analyses. These include context-, flow-, object-, field-, and path-sensitivity, or any combination of them [109, 86, 16]. Context-sensitivity differentiates taints from different function calls of the same function definition, flow-sensitivity distinguishes the taints associated with the same variable at each assignment, object-sensitivity distinguishes different instances of the same class, field-sensitivity distinguishes between the fields of the same object, and path-sensitivity takes into considerations the variable(s) that decide the control flow when analyzing the different branches. There is a correlation between sensitivity and accuracy. If we give up on sensitivity along one or more dimensions, we get a simpler and faster tool, but we increase the possibility of reporting more false positives.

To minimize false positives, we choose to make CtxTainter's analysis context-, flow-, object-, and field-sensitive. Our analysis is not path-sensitive, which means that implicit information flows that arise from the evaluation of conditional branch predicates will not influence the labels/taints assigned to `DEF`s of different branches. Path-insensitivity in the case of CtxTainter is tolerable since the most important state (such as keys, passwords, user profile information, PRNG state, *etc*.) would be retained in variables that cross request boundaries through direct flows and not through implicit flows.

Because CtxTainter does not need to be path-sensitive, the taint of variables that are part of a conditional predicate does not need to be propagated to all `DEF`s that are assigned in the conditional branches. However, adding such support should be straightforward [16] if a use-case calls for it.

4.1.2 Assumptions

The FaaS platform, including the platform software, OS kernels, hypervisors, and platform services are trusted. Function implementations provided by tenants, including any libraries they link and the language runtimes that are not marked for analysis are assumed to be stateless.

Any source code that is marked for analysis is not trusted and may contain bugs that retain state, violating the FaaS statelessness assumption. CtxTainter's analysis reports individual execution steps that handle data flows from previous request contexts and optionally provides full traces of flows that stem from sources marked for full-trace reporting. For data flows that pass through functions that

are part of trusted libraries, CtxTainter assumes that the functions' returns only depend on the passed arguments.

When all of these assumptions hold, CtxTainter's design allows for a sound analysis of direct data flows on the executed paths. However, if one or more of the aforementioned assumptions do not hold, then CtxTainter becomes a best-effort aid for detecting data flows that cross a request context boundary.

4.2 Design and Implementation

This section provides an incremental description of the design of CtxTainter. Although CtxTainter's design is applicable to various dynamic programming languages such as Python, Ruby, and JavaScript, we adopt the terminology of the Python programming language in this thesis[2].

4.2.1 Overview

CtxTainter's design consists of three components, each responsible for one of the three phases of CtxTainter's operation. The three components/phases are ① **Preparation**: *Static* source-code normalization and AST-metadata extraction, ② **Online**: Execution of the code in a test environment and metadata collection, and ③ **Offline**: Data traces analysis.

[2]Python, being the most commonly used language for Function-as-a-Service (FaaS) [32], was chosen for implementing CtxTainter's proof of concept.

The **preparation phase** takes as an input the source code that will be subject to analysis. In this phase, the source code undergoes static transformations to simplify the subsequent phases. The source code transformation linearizes nested expressions, simplifies complex constructs, and transforms the code to its equivalent Single Static Assignment (SSA) form (whenever possible). Additionally, in this phase, the AST of each module is stored, and metadata about the source code structure is extracted and stored to assist in identifying the binding scope of variables referenced during the subsequent phases of CtxTainter.[3]

Next, the **online phase** invokes the FaaS function with developer-provided test inputs in a debugging environment that collects the relevant runtime metadata during the execution. This debugging environment relies on the language's debugging library (`sys.settrace()` in the case of Python), which is instructed to invoke a debugging hook *before* each statement is executed. The debugging interface provides the hook with information about the current statement as well as access to the interpreter's current frame object (which contains information on all visible variables). The hook then statically extracts the `DEF`s/`USE`s of the current statement by looking up the AST statement (available from the preparation phase) that corresponds to the statement to be executed. The hook assigns each `DEF` a *unique identifier*[4] based on its static and dynamic occurrence in the code and the execution, respectively. Similarly, the hook resolves the `USE`s to their earlier `DEF`s

[3]The preparation phase is mostly based on PolyCruise's [66] version of the PyPredictor [129] normalization module. CtxTainter extends the reused component to additionally extract information about the code structure and thereby allow precise variable scoping in the subsequent phases.

[4]Constructing a unique identifier for each `DEF` is crucial for context-sensitivity because at the time of resolving a `USE`, the tool must be able to precisely identify the `DEF` based on the variable-binding context and the language resolution rules.

by applying the language's variable resolution rules. Finally, the hook also labels each DEF with the request context it appeared in, attaches taint to all variables consuming data from the developer-specified sources of interest (and propagates the taint downstream to all dependent variables), outputs information on all USEs that depend on DEFs defined in previous request contexts, and stores traces of all statements that handle tainted data (coming from developer-specified sources of interest).[5]

Finally, the **offline analysis** performs a form of dynamic program slicing [27, 125] by consuming the output traces from the online phase, then building a dependency graph that captures the DEF-USE chains of tainted variables that stem from a developer-specified source and reach a developer-specified sink. This dependency graph is then analyzed to identify flows that span multiple request contexts, and finally, the tool outputs the execution trace to facilitate debugging.

Next, we describe the technical design details that enable CtxTainter's analysis.

4.2.2 Inputs and Outputs

The inputs to the analysis are:

- The source code to be analyzed. This includes the source code of the function and all untrusted libraries.

[5]Our implementation of the online phase is based on the dynamic data flow tracking framework introduced by PolyCruise[66] and PyPredictor[129]. CtxTainter extracts a lot of additional information to allow the analysis to be object- and field-sensitive and makes the analysis aware of request contexts.

- The function call that marks the beginning of a new request (*i.e.* the entry point of the FaaS function).
- Optionally, sources (and sinks) of interest.

The outputs of the analysis are:

- All incidents involving statements that depend on data stemming from a previous request context.[6]
- A full trace of all statements that appear in each flow that starts at a developer-marked source of interest, crosses a request context boundary, and ends at a sink of interest.

In addition to the main analysis outputs, the preparation phase outputs the normalized source code as well as the following mappings[7] (the relevance of which will become apparent in the next subsection):

- Normalized source code line number -> Original source code line number (to facilitate mapping the normalized code to the original source code)
- Normalized source code line number -> AST representation (to enable extracting the `DEF`s and `USE`s during the online phase)
- Normalized source code line number -> Enclosing function/class definition line number (to allow detecting variables scopes)

[6]The dependency on data introduced in a previous request context might cause confidentiality and/or correctness violations. All such incidents are reported by default, even if the developer does not specify sources/sinks of interest.

[7]The first two mappings are already emitted by PolyCruise[66], the third mapping, from the normalized source code line number to the enclosing function/class definition line number, is introduced by CtxTainter.

4.2.3 Metadata Extraction

At its core, CtxTainter's analysis constructs global DEF-USE chains for all variables encountered during the execution of the program. DEFs and USEs are data holders that are assigned (DEF) or used/referenced (USE) during the execution. Because FaaS platforms handle requests one at a time (§3.1.1), it is easy to identify where a request context change happens in the DEF-USE chains.

While conceptually very simple, to construct accurate[8] DEF-USE chains we need to accurately and uniquely identify each DEF and be able to reliably resolve its USEs to it.

Uniquely identifying DEFs CtxTainter's reliance on the debugging interface means that CtxTainter is allowed to inspect the statement *before* it is executed. This means that the DEF of the statement is not yet present in the interpreter's state. CtxTainter differentiates between two types of DEFs: scope-accessible and global-accessible.

Scope-accessible DEFs require information about the scope from which they are accessible. The accessibility scope of these DEFs is determined by their location in the code (static), and the call stack frame at which they are DEFined (dynamic).[9] Consequently, scope-accessible DEFs are uniquely identified by their static scope in the form "module.(class|function)*.NAME" along with a dynamic unique call stack frame ID.

[8] By accurate, we mean that the analysis is context-, flow-, object-, and field-sensitive.

[9] Context-sensitivity requires differentiating between the same variable DEFinitions that are introduced in different function calls.

Globally-accessible DEFs, on the other hand, are object attributes (e.g. module-bound variables, class (static) variables, instance variables, ...) that can be accessed from any code location in the program through an object reference and an attribute. Consequently, globally-accessible DEFs are uniquely identified by the object reference and the field name. In Python, a global DEF that is assigned within a non-global scope (*e.g.* a function) must be declared with keywords that indicate the scope binding [95]. DEFs that are bound globally at the module level must be declared “global". Similarly, DEFs that are to be bound to the nearest enclosing function scope, must be declared with the “nonlocal" keyword.

Mapping USEs to DEFs The unique identification rules for DEFs apply to USEs as well. Additionally, and in contrast to DEFs, when USEing variables that are not defined in the current scope, the Python interpreter does not require explicit variable qualification using the global or nonlocal keywords; instead, the Python interpreter automatically resolves the USE according to the LEGB rule.[10] CtxTainter follows the same scope resolution rules to be able to map USEs to DEFs. Specifically, CtxTainter’s online phase maintains information about the scope at which each DEF was defined. This is done by maintaining a set of globally DEFined variables as well as a set of DEFined variables for each frame on the call stack.

[10]The LEGB acronym stands for *Local, Enclosing, Global, and Built-in scopes* [64, 95].

4.2.4 Request Contexts and Taint Tracking

To detect data flows that cross a request context boundary, CtxTainter needs to maintain information on the context at which a variable was last written (DEFined). Accordingly, CtxTainter classifies each DEFined variable into one of three sets: The *initialization set*, the *previous requests contexts set*, and the *current request context set*. The initialization set contains all DEFs that are encountered before the first request's input is read. The previous requests contexts set contains all DEFs encountered after reading the first request's input and before reading the current request's input. Finally, the current request context set contains all DEFs encountered since the current request's input was read. These three sets are maintained for globally-scoped DEFs and for the locally-scoped per-stack-frame DEFs.

To additionally maintain information on data stemming from developer-specified sources of interest, the maintained sets are converted to hash maps that store a *taint* boolean to indicate that a particular variable stems from a *tainted source of interest*. Throughout the program execution, a DEF will be tainted if it directly reads data from a tainted source or a tainted USE. (See §4.2.5 for the detailed taint propagation rules.)

In summary, the final data structures maintained are:

- Three sets of global hashmaps for the initialization, previous, and current globally-accessible DEFs of the corresponding request contexts.

- Three sets of per-stack-frame hashmaps for the initialization, previous, and current scope-accessible DEFs of the corresponding request contexts.

Once the execution starts, encountered DEFs are always added to the “current" hashmap of the corresponding accessibility level (global/local). On the first encounter of the function that marks the beginning of a new request context, all DEFs in the current request context hashmap are moved to the initialization hashmap. Subsequent encounters of the function that marks the beginning of a new request context will move the DEFs from the current request context to the previous requests context hashmap.

4.2.5 Taint Propagation Rules

As described in 4.2.4, one of CtxTainter’s features that facilitates debugging is tracking and reporting full execution traces of correctness- and confidentiality-critical flows that stem from a developer-specified source of interest. To do that, CtxTainter attaches a taint to the returns of the input functions of interest specified by the developer and propagates that taint throughout the DEF-USE chains. Accordingly, newly defined DEFs will become tainted if they consume data held by a tainted USE. There are four cases for the forward propagation of taint from USEs to DEFs:

- A statement with a DEF and one or more USEs: If one of the USEs is tainted, then the DEF gets tainted.

- A call to an instrumented function with one or more passed arguments: If a passed argument is tainted, then the corresponding formal argument gets tainted.

- A return from an instrumented function: If the return is tainted, then the `DEF` at the call site gets tainted.

- A call to a non-instrumented function: The `DEF` at the call site is tainted if any of the passed arguments is tainted.

4.2.6 Detecting Boundary-Crossing Flows

A data flow that crosses a request context boundary is identified if a `USE`d variable is not found in the current request context's global or per-stack-frame hashmaps but is found in the previous request contexts' corresponding hashmaps.

Detected data flows that cross the request context boundary indicate that the function does not honor the *statelessness* requirement of the FaaS contract and is retaining data across requests. This retained data may be confidentiality- or correctness-critical, and could jeopardize the end-client's confidentiality if no request-isolation enforcement mechanism (such as Groundhog) is in place. Similarly, correctness may be in jeopardy if the FaaS platform performs optimizations that assume statelessness of functions.

4.2.7 Reconstructing Violating Flows

CtxTainter's offline analysis phase consumes the `DEF/USE` metadata of all instructions that handled tainted data during the online phase. From this, it constructs a dependency graph represented as an adjacency matrix. The graph nodes represent `DEF`s, and edges represent dependencies (if $graph[i][j]$ is set, it means that the variable represented by the node in row_i depends on the variable represented by the node in $column_j$). In a standard dependency graph, a variable occupies only one row; as the execution proceeds, the bits in a variable's row represent the variable's *current* dependencies.

CtxTainter builds the dependency graph differently, as a continuously expanding adjacency matrix – at every execution step, the assigned variable is placed in a new row, even if it has already been assigned earlier. As a result, the dependency graph captures the entire history and order of dependencies between variables (*flow-sensitivity*), not just the current status.

Once the dependency graph is built, request contexts are represented in adjacent rows and columns in the matrix. Similarly, developer-specified sources and sinks of interest are mapped to nodes on the dependency graph (*i.e.* matrix indexes). Identifying violations is done by simple bounds check on the dependencies of each `DEF`; if a `DEF` has a dependency in an index that belongs to a previous request context, then this `DEF` marks the point at which the data flow crosses the request boundary. For each such detected violation, a simple graph traversal starting at the violation finds the provenance of the violation and the downstream flows toward developer-specified sinks. The reconstructed data flow is then a candidate

for reporting to the developer as it stems from a source (and optionally reaches a sink) specified by the developer.

4.2.8 Limitations

The current design and implementation of CtxTainter suffers from some limitations. First, CtxTainter assumes that non-instrumented functions (that are part of non-instrumented third-party/native libraries) will only pass the taint from the arguments to the return. This assumption might lead to false negatives. One way to address this limitation is to mark all standard language libraries as part of the code-base that is subject to analysis and instrument native libraries to track taint.

Second, CtxTainter simplifies the tracking of collection data structures (lists, dictionaries, and sets) by maintaining a single taint for the whole data structure. If the data structures holds variables of primitive data types (int, char, *etc*.), then false positives may arise. If, however, the data structure holds objects, then there will be no false positives because the fields of the objects will be identified by the object reference and field name rather than the collection's object reference. Support for per-index taint for collection-like data structures could be added at the cost of increased space consumption.

Finally, analyzing compute-intensive loops would result in slowdowns that would increase the duration of the analysis, leading to developers waiting for more than a few seconds for the analysis to finish. Similarly, if these compute-intensive loops handled tainted variables, there would be a fast growth in the number

of emitted DEF/USE metadata chains. This limitation can be overcome through optimizations that detect the presence of such compute-intensive loops and enable coarse-grained tracking for them at the cost of reduced precision.

4.3 Implementation

We implement CtxTainter as an extension of standard dynamic data flow analysis. CtxTainter implements a novel analysis that utilizes the same information readily utilized by the taint-propagation components of DDFA tools. Concretely, CtxTainter builds on PolyCruise [66], a state-of-the-art cross-language data flow analysis tool.[11]

We chose PolyCruise because it has a modular design and decomposes the dynamic data flow taint analysis into two independent phases. The first phase is source-code translation, where the source code is transformed into a language-independent symbolic representation (LISR) that captures the DEF/USE sets of each relevant execution step. In the second phase, a language-agnostic data flow analysis engine analyzes the emitted DEF/USE sets. By separating these two phases, PolyCruise is able to perform data flow analysis on flows of data that span different programming languages by implementing the first phase for each language of interest. For the Python programming language, PolyCruise relies on PyPredictor [129] for transforming the Python source code into its equivalent

[11] Similar DDFA tools could have also been used as a base.

Single Static Assignment (SSA), which is then directly mapped to its LISR during runtime (by interrupting each instruction via `sys.settrace()`).

CtxTainter utilizes the Python source code translation component from PolyCruise and builds a new analysis engine. Although PolyCruise's engine is only execution-context- and flow-sensitive, CtxTainter's engine, in addition to being request context aware, is also execution-context-, flow-, object-, and field-sensitive.

4.4 Evaluation

In this section, we assess how well CtxTainter aids developers in reasoning about data flows that span multiple request contexts. We aim to answer the following questions:

1. How effective is CtxTainter in detecting data flows that cross a request context boundary? (Are there false negatives?)

2. How precise is CtxTainter's analysis? (Are there false positives?)

Throughout this section, we will highlight the capabilities (and limitations) of CtxTainter by showcasing various scenarios where CtxTainter is able (or unable) to detect data flows that cross a request context boundary. We also show the effort required from the developer to use CtxTainter on ready-to-deploy FaaS functions.

To understand CtxTainter's effectiveness in detecting data flows that cross the request context boundary, we handcrafted 10 test cases that use various features of the Python programming language to inject data flows that cross the request context boundary and can lead to confidentiality and/or correctness violations. To report on CtxTainter's precision, we analyze all Python functions that are part of Groundhog's evaluation along with the third-party libraries they rely on and report on the flows that cross the request context boundary as well as any false positives encountered.

4.4.1 Handcrafted examples

In high-level programming languages that do not generally allow direct memory access via mutable pointers, data flows that cross a request context boundary must happen through global state. This global state may be set and read through global variables, static/class variables, and object fields. To give an intuition of how such flows can manifest, we present a detailed walkthrough of one of the handcrafted examples (Example 1 below) that shows a data flow through a global variable. We then briefly discuss a second example in which data flows across a request context boundary through both static variables and object fields.

While detecting request-context-crossing data flows is conceptually simple, the challenge is in tracking flows through the language features that eventually lead to global state tainting. In our handcrafted examples (listed below), we cover a sizeable subset of the Python language's main features.

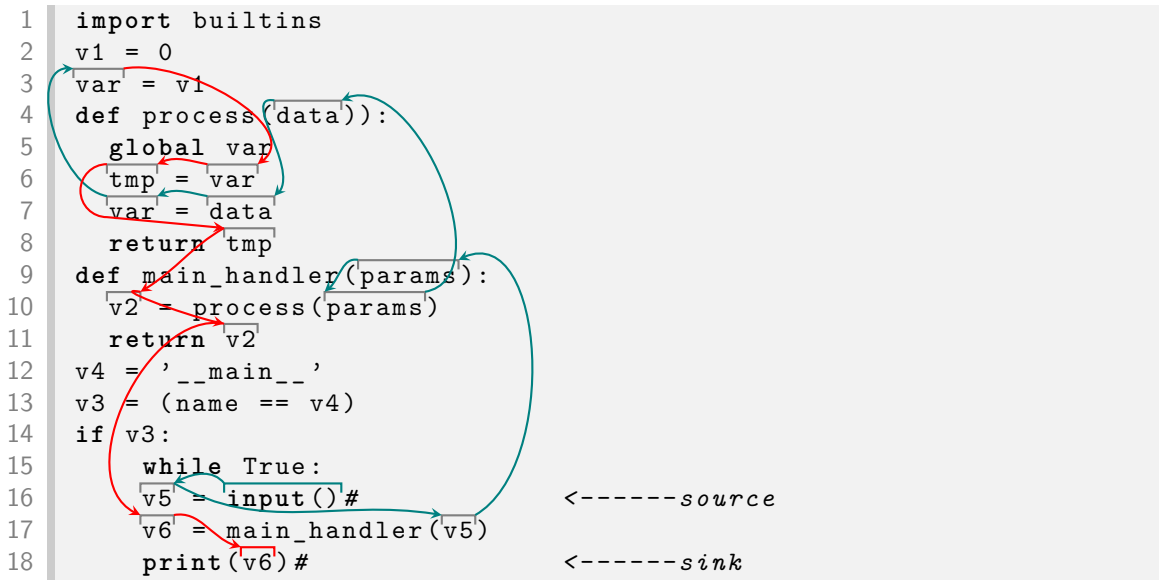
- P1: A global variable assigned in one request context and read in a subsequent one.
- P2: Time measurements, where an initial timestamp is taken in a request context and used for comparison in a subsequent one.
- P3: A variable assigned in a conditional branch and read in a subsequent request context that does not re-assign the variable first.
- P4: An internal state of a library is maintained and updated over multiple requests. (In this case, a PRNG python library holds the latest state to generate a new pseudo-random number.)

- P5: Accumulating end-client inputs (primitive data types) into lists/dictionaries/sets, but reading only the data item that was written in the current request context.
- P6: Accumulating end-client inputs (objects with fields) into lists/dictionaries/sets, but reading only the data item that was written in the current request context.
- P7: Passing confidential data through a non-instrumented library.
- P8: Passing data derived from tainted sources through recursive function calls.
- P9: Passing data derived from tainted sources through functions that rely on Python's polymorphism for resolving the target function.
- P10: Passing data derived from tainted sources through nested Python classes and nested functions.

Example 1

```
1  var = 0
2  def process(data):
3      global var
4      tmp = var
5      var = data
6      return tmp
7  def main_handler(params):
8      return process(params)
9  if __name__ == '__main__':
10     while (True):
11         print(main_handler(input()))
```

Listing 4.1: P1: Confidential inputs of each request is leaked to the subsequent one through a global variable.



```
1  import builtins
2  v1 = 0
3  var = v1
4  def process(data):
5    global var
6    tmp = var
7    var = data
8    return tmp
9  def main_handler(params):
10   v2 = process(params)
11   return v2
12 v4 = '__main__'
13 v3 = (name == v4)
14 if v3:
15     while True:
16     v5 = input()#          <------source
17     v6 = main_handler(v5)
18     print(v6)#             <------sink
```

Listing 4.2: A represention of the unwanted data flow happening in Listing 4.1 after the SSA transformation. In one request, confidential data from input() in line #16 (marked “source") flows along the green path to the variable “var". In a later request, that data flows along the red path from “var" to the output in line #18 (marked "sink").

In this sample program, `input()` is a source of interest that will return tainted data that must be fully tracked, and the sink is any data that is passed to the function `print()`. The main handler (Lines 7-8) calls a buggy function `process` that leaks the passed data to a global variable. This global variable is read during the subsequent call to the `process` function and returned. The function call `input()` defines the request boundary. To detect the flow that crosses the request boundary, we run CtxTainter on P1.

First, the example source code in Listing 4.1 is normalized during the preparation phase to be in the form shown in Listing 4.2. Next, the online phase runs the normalized P1 under the debugging interface where CtxTainter’s hook is called

	v1	var	v4	v3	v5 (source)	params	data	tmp	var	v2	v6	v5 (source)	params	data	tmp	var	v2	v6
v1	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
var	1	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
v4	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
v3	.	.	1	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
v5	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
params	.	.	.	.	1	.	.	.	.	.	.	.	.	.	.	.	.	.
data	.	.	.	.	.	1	.	.	.	.	.	.	.	.	.	.	.	.
tmp	.	1	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
var	.	.	.	.	.	.	1	.	.	.	.	.	.	.	.	.	.	.
v2	.	.	.	.	.	.	.	1	.	.	.	.	.	.	.	.	.	.
v6 (sink)	.	.	.	.	.	.	.	.	.	1	.	.	.	.	.	.	.	.
v5	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
params	.	.	.	.	.	.	.	.	.	.	.	1	.	.	.	.	.	.
data	.	.	.	.	.	.	.	.	.	.	.	.	1	.	.	.	.	.
tmp	.	.	.	.	.	.	.	.	1	.	.	.	.	.	.	.	.	.
var	.	.	.	.	.	.	.	.	.	.	.	.	.	1	.	.	.	.
v2	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1	.	.	.
v6 (sink)	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1	.

Table 4.1.: *CtxTainter's dependency graph (adjacency matrix) for Listing 4.2. Different shades of gray represent different request contexts.*

before each statement. CtxTainter extracts the DEFs/USEs from each statement and keeps track of the request context they appeared in as well as their accessibility scope. CtxTainter propagates the taint forward according to the propagation rules outlined in §4.2.5 and outputs a trace of all the statements (with the collected metadata) referencing tainted USEs. Finally, the offline phase consumes the output trace and constructs the flows that show how the leak happened, as we explain next.

For explanation purposes, Table 4.1 was generated using a special configuration that instructs CtxTainter to output and builds the dependency matrix for all executed statements instead of only outputting the statements that are part of data flows stemming from the developer-marked source of interest.

In Table 4.1, we see two boundary-crossing. A red colored cell represents the first read that causes data stemming from a source of interest to leak into a subsequent request context. The red-colored edges (set bits) collectively represent a flow starting from a specified source of interest and reaching a sink (with a marked border). Similarly, an Orange colored cell represents the first read of data in a flow that stems from non-sensitive initialization data that was assigned before the first request context, and all orange-colored edges collectively represent a flow starting from non-confidential data and reaching a sink (with a marked border). CtxTainter outputs the flows in the form of a chain of <variable:LineNumber> tuples, with the variable that crossed the request context marked. The red flow is outputted as: v5:16 -> params:9 -> data:4 -> var:7 =» tmp:6 =» v2:10 -> v6:17, while the orange flow is outputted[12] as: v1:2 -> var:3 =» tmp:6 =» v2:10 -> v6:17.

Example 2

In this example (Listing 4.4), we have a program that takes a student name as an input and generates a unique random student ID based on the population count and a randomly generated number. The PRNG implementation snippet shown in Listing 4.3 is taken from [68].

[12]Because the orange flow handles initialization data rather than data stemming from a previous request context, it is not emitted by default (Unless requested by the developer)

```
import numpy
class PRNG(object):
    """Base class for any Pseudo-Random Number Generator."""
    def __init__(self, X0=0):
        """Create a new PRNG with seed X0."""
        self.X0 = X0
        self.X = X0
        self.t = 0
        self.max = 0
....
class LCG(PRNG):
    """A simple linear congruential
    Pseudo-Random Number Generator."""
    def __init__(self, seed=0, m = 2**31-1, a = 7**4, c = 0):
        """Create a new PRNG with seed X0."""
        super().__init__(X0=seed)
        self.m = self.max = m
        self.a = a
        self.c = c

    def __next__(self):
        """Produce a next value and return it,
        following the recurrence equation:
        X_{t+1} = (a X_t + c) mod m."""
        self.t += 1
        x = self.X
        self.X = (self.a * self.X + self.c) % self.m
        return x
```

Listing 4.3: A snippet from the PRNG library [68].

```
from PRNG import LCG
class Person:
    population = get_population_from_db()
    prng = LCG(seed=12011993 + population)
    def __init__(self, name):
        self.name = name
        Person.population += 1
class Student(Person):
    def __init__(self, name):
        super().__init__(name)
        self.id = str(self.population + self.prng.randint())
if __name__ == '__main__':
    while (True):
        s = Student(input())
        print(s.name, s.id)
```

Listing 4.4: P4: Correctness-critical state is retained (by object fields in the PRNG library and by a class/static variable of the Person class) and used accross request contexts.

Upon Initialization, the function refreshes the population counts from a remote server and seeds a pseudo-random number generator before handling any request. In the shown snippet, each request updates the population count and generates a new student ID based on the updated population count and a pseudo-random number.

For this example, CtxTainter reports multiple statements that read data written in a previous request context. Namely,

- $Person.population$ at Person.init:7
- t at LCG.__next__:25
- X at LCG.__next__:26
- X at LCG.__next__:27

None of the statements stem from a developer-specified source that should be fully tracked. Accordingly, CtxTainter reports only the statements that reads data from a previous request context rather than the full trace. Note that $self.a$, $self.c$, and $self.m$ in LCG.__next__:27 are detected as flows that stem from the initialization and thus are not reported by default.

Results

We ran CtxTainter on all hand-crafted examples (P1-P10). We confirmed that all relevant flows were reported (*i.e.* there were no false negatives). However, a false positive was reported in the case of P5 because CtxTainter currently accumulates the taint of all data inserted into a collection data structure, which is a known

limitation when primitive data types are added to collections (see §4.2.8). Note that this does not affect P6, because CtxTainter is field-sensitive, and attaches taints to the object fields and not the object references.

4.4.2 Benchmark examples

While we do not claim that CtxTainter supports the full Python Programming language syntax, it does support a relatively large set of it as demonstrated by the analysis of generic Python benchmarks.[13] We ran CtxTainter on all the Python functions that are part of the Groundhog evaluation, including all the third-party libraries they rely on. Out of 28 benchmarks, two benchmarks were not fully analyzed. The sentiment-analysis benchmark was not analyzed due to a bug in the source-code translation logic in one of the PolyCruise components. Additionally, the pyperf-mdp benchmark has compute-intensive loops, thereby slowing the processing of the benchmark when run under CtxTainter's taint tracking. The majority of the 26 successfully analyzed benchmarks were analyzed by CtxTainter in less than a minute (median: 19s, max: 40h:27m). We leave runtime optimizations to future work (discussed in §5.1.2).

In all benchmarks, we mark the function that reads the request inputs as the request context boundary and as a source of interest.

[13] CtxTainter supports the subset of the python programming language that is generated by the SSA normalization step of PolyCruise/PyPredictor

Results

On the 26/28 successfully analyzed pyperformance and FaaSProfiler benchmarks, no tainted data leaks across request context boundaries were detected. This was expected because the majority of the benchmarks only take an input to control how compute-intensive the benchmark should be; this input is then eventually used in a loop guard, which does not taint `DEF`s introduced in the loop body because, as discussed in §4.1.1, our FaaS-focused analysis does not need path-senstivity.

In two benchmarks (pyperf-go and pyperf-richards), CtxTainter reported 11 unique data flow violations where a data flow crossed the request context boundary. All occurrences of the 11 violations were true positives, and the code was manually inspected to verify that there was a variable update in one request context and that the updated state was accessed in the subsequent request context. CtxTainter did not report any false positives.

The 11 unique data flow patterns that CtxTainter reported were as follow: In pyperf-go, 6 execution paths involved reading one of two global variables. Both global variables were updated in a request context and later read in a subsequent one. On the other hand, pyperf-richards maintains a global object that has multiple fields, some of which are nested objects. CtxTainter reported 5 execution paths that involved reading one of the attributes of the global object or of one of its nested objects that had an attribute updated in one request context and later read in a subsequent one.

While the loss of the state maintained via the reported data flows might not be detrimental to the correctness of the benchmarks, similar patterns in other FaaS functions might result in impaired functionality or data loss if the FaaS provider decides to employ optimizations that assume that functions are stateless. Such optimizations include directly terminating the function after each request, bootstrapping multiple function instances by cloning existing functions, or request-isolation that involves function-state-rollback as in Groundhog.

4.5 Related Work

While Data Flow Analysis (DFA) can be used to analyze the source code of functions and assist developers in tracking data flows and understanding them, existing tools focus on vulnerability detection and are geared towards finding flows from sources (usually confidential or untrusted input) to sinks (usually public outputs or critical APIs) without any notion of request boundaries. CtxTainter extends DFA techniques to make them request-aware. In the rest of this chapter, we discuss the different general analysis techniques and why they are insufficient for analyzing statlessness. Finally, we discuss the information flow analysis trends in FaaS and how CtxTainter contrasts with them.

4.5.1 Static Analysis

Static analysis tools analyze the source code of an application and build data flow graphs without running the application, providing a cheap way for scanning

the application for bugs/vulnerabilities. Industry-standard static analysis tools such as [42, 40] are optimized for security vulnerability detection and must make approximations that lead to false positives, false negatives, or both. Indeed, it is not possible for a static analyzer to accurately detect all flows of information that will be dynamically resolved (at runtime) without explicit hints from the developer, more so in dynamic languages that are the norm for FaaS applications [32].

4.5.2 Dynamic Analysis

Dynamic analysis tools [113, 62, 28, 25, 101, 85, 66, 61] are able to precisely capture data flows, even for dynamically resolved variables. On the other hand, they are limited to the execution paths that are exercised during testing. For CtxTainter, as a tool that aids developers, relying on dynamic analysis is an acceptable compromise, as most functions are (1) written in dynamic languages, and are (2) designed to perform one functionality, so the code path exercised is likely similar for all requests.

In contrast to static analysis, dynamic analysis makes it easier to reason about different instances of (calls to) the same data source, which can be easily expanded by CtxTainter to reason about flows spanning multiple request contexts.

4.5.3 Hybrid approaches

Hybrid approaches such as concolic execution [44, 103, 102] rely on symbolic execution [58], which analyzes the program to identify the dependencies between program symbols. These dependencies can be used to build the program's control flow graph (CFG). The CFG can then be used to derive expressions that can be used to generate inputs that exercise various execution paths and then run the program under dynamic analysis with fresh inputs generated to cover the execution paths. This approach augments dynamic analysis by improving its path coverage. Extending CtxTainter with a symbolic execution pass to enhance coverage is left as future work.

4.5.4 Verification

A verifiably stateless function trivially does not have boundary-crossing flows and will, therefore, meet its contract. Additionally, it will not leak data even if the provider does not enforce sequential request isolation (*e.g.* through Groundhog). However, current methods of formal verification require expert knowledge of special theorem proving tools [20, 87], a huge investment of time [65, 60], and more research is required to enable robust support for dynamic languages. Hence, relying on assistive best-effort analysis tools like CtxTainter is a much more viable alternative.

4.5.5 Information flow in FaaS

Most existing work on information flow analysis in FaaS has been done at the granularity of the FaaS function (the functionality and deployment unit), where the goal is to prevent control flow hijacking by passing information between functions in an unintended order, as in SecLambda [55] and Alastor [34]. Systems such as Valve [33] propagate taint over the network across workflows to capture inter-function security violations. SCIFFS [93] enables information flow tracking of confidential FaaS when the data propagates to third-party security analytics platforms. These systems all focus on leaks that happen through channels outside the execution environment. CtxTainter, on the other hand, focuses on the orthogonal problem of intra-function data flows that arise due to the re-use of the execution environment among mutually distrusting end-clients. A secure FaaS offering should have mechanisms to prevent both intra- and inter-function data leaks, and CtxTainter (and/or Groundhog) can be augmented with inter-function data flow tracking tools to prevent both kinds of leaks.

Distributed tracing aims to track data flows that belong to a request within a distributed system to analyze correlated events across different components. Distributed tracing systems propagate request-critical information along the request execution path within the system's components (as in baggage contexts [78]). Like standard data flow tracking, distributed tracing propagates relevant information within an execution environment and additionally propagates selected information across environments, so a request context encapsulates a single request across multiple components. CtxTainter, on the other hand, is limited

to a single execution environment but analyzes data flows across sequential requests.

4.6 Conclusion

This chapter presented CtxTainter, a system that assists developers in identifying flows of data that cross the request context boundary, thus helping a function developer meet their side of the FaaS contract.

CtxTainter can also be used as a standalone sequential request isolation tool that assists developers in identifying confidentiality-critical flows of data that cross a request context boundary.

5 Conclusion and Future Work

The Function-as-a-Service (FaaS) model provides a simplified programming model for developing inherently scalable event-driven applications. The FaaS model allows tenants to state their application logic in the form of functions without managing the underlying infrastructure that runs and scales their applications. FaaS providers are responsible for deploying and executing the tenants' functions, provisioning and replicating functions as workload demand fluctuates, and maintaining and multiplexing the hardware and software infrastructure across different tenants and functions.

Current FaaS providers ensure the confidentiality of tenants' data by isolating function instances from one another. While such isolation suffices for ensuring confidentially and correctness properties for stateless functions, a function's statelessness cannot be trivially guaranteed or verified, especially for functions that rely on third-party libraries and runtimes that may have bugs or may have been developed originally for stateful paradigms. As a result, sequential requests that activate the same function instance may be at risk of confidentiality breaches because bugs in the function's code or in a third-party library it relies on could

expose data from one request to a subsequent one. Moreover, optimizations that reuse an instance's state jeopardize the function's correctness because such optimizations can introduce unintended data flows between sequential requests.

This thesis presented two complementary systems: Groundhog and CtxTainter. Groundhog is a black-box and programming-language agnostic solution that enforces confidentiality by design. Groundhog efficiently rolls back changes to a function's state after each function activation, effectively breaking all data flows at the request boundary. CtxTainter is a development-phase dynamic data flow analysis tool that detects data flows that cross a request context boundary and reports them to the developer for auditing.

5.1 Future Work

5.1.1 Groundhog

The current Groundhog prototype is built using standard Linux Kernel facilities. Groundhog allows retrofitting existing production systems with request isolation without the need for kernel patching. That said, Groundhog can benefit from local optimizations that can improve the efficiency of the memory snapshotting, tracking, and restoration operations. Additionally, global optimizations at the platform/host level would allow Groundhog to operate with lower resource consumption.

Local Optimizations

A custom dirty-page tracking facility Groundhog's prototype relies on the standard Linux Kernel soft-dirty bits (SD) tracking [71]. SD bits provide a simple and clean interface for memory modification tracking but suffers two main limitations. First, the absence of tracking granularity control: SD does not provide knobs for setting the tracking granularity: either all process pages are tracked or none. Second, there is no interface to retrieve the set of pages that have been modified. Instead, the whole mapped address space must be scanned to identify the modified pages.

One way to address this limitation would be to use an approach similar to Linux's user-space fault-tracking file descriptor (UFFD) [72] that does not suffer from these problems. UFFD allows fine-grained tracking control and sends fault notifications of modified pages to the userspace for handling. However, the IOCTL-based notification mechanism has significantly higher overhead compared to SD-bits due to the frequent context switches to user space for fault handling.

Relying on a custom in-kernel facility that allows the application to specify the regions to be tracked (as in UFFD) and additionally request a list of modified pages asynchronously would be probably more efficient. A similar facility was implemented by James Litton [75] for anonymous memory mappings and showed promising throughput improvements over standard SD-bits.

Hot memory regions With better control over the memory tracking, hot memory regions that are modified as part of each request can be detected and excluded from the tracking and be *always* restored after each request.

Global Optimizations

A production FaaS platform that retrofits Groundhog can enable scheduling optimizations that further reduce the customer-perceived impact of restoring containers to a clean state. For instance, sequential requests from the same end-client trust domain can be forwarded to the same function instance without the need to roll back the state of the previous invocation. Similarly, Groundhog-aware scheduling optimizations (similar to the optimizations described in [2]) can be applied for applications that invoke multiple functions in a workflow/sequence such that resources can be directed towards function instances that will be invoked first.

Another optimization would be to run Groundhog as part of the container/VM management system, thereby allowing Groundhog to consolidate and deduplicate the memory snapshots of all function instances on the same host. However, this optimization carries the risk of introducing new performance bottlenecks that are not present in the current design.

5.1.2 CtxTainter

CtxTainter tracks data flows that cross a request context boundary by means of dynamic taint tracking. The nature of dynamic taint tracking, as part of a forward execution of the source code, makes forward tracing of data flows easy. To that end, CtxTainter allows developers to specify sources of data that will be fully traced. Complete flow traces that show the position of a violation on the trace are easy to audit and fix. Additionally, CtxTainter reports all instances of boundary-crossing data movements, even if the upstream source of data is not marked for tracking by the developer. These reported boundary-crossing data movements serve as a hint for the developer to investigate the data flow and decide whether to deem such flow safe or not. In some cases, the developer may need support in the form of tracing the data movement operation backward to the data source that introduced it. Backward tracing is not trivial, as it requires information that was not collected during the execution of the program.

To support backward tracing, one can record all execution traces, but that is not feasible from a computational and storage point of view. Programs can perform millions of memory reads/writes per second, and the overhead of tracking the dependencies and flow provenance in such scenarios is very high. To combat the explosion in the storage requirement, one can rely on coarse-grained tracking that records high-level dependencies for debugging purposes. For instance, detailed dependencies captured within a function call can be discarded on the function's return, and only the dependencies between the outputs and the inputs get maintained. Similarly, recording the dependencies within a loop can be skipped, and a compact dependency summary can be added once the loop concludes. Such

optimizations can enable full tracing of all data flows at a lower computational cost.

Another approach to support backward tracing would be to re-execute the source code, instructing the dynamic tracing to record all traces for the N execution steps preceding the execution step that resulted in a boundary-crossing data movement. This can be easily achieved by means of execution step counting and the use of deterministic program inputs during testing.

Appendix

A

Detailed Groundhog results

		E2E Latency (ms)	Invoker Latency (ms)	T'put (req/s)
base	chaos (p)	688 ±143	649 ±86	6.03
GH		691 ±111	652 ±39	5.94
GH_{NOP}		689 ±116	649 ±54	5.98
fork		690 ±105	651 ±64	5.81
faasm		1235 ±18	1201 ±13	2.99
base	deltablue (p)	48.4 ±56	20.4 ±1.6	158
GH		50.3 ±52	21.3 ±2.0	140
GH_{NOP}		47.0 ±34	20.7 ±2.0	155
fork		53.0 ±55	22.1 ±1.9	84.7
faasm		150 ±5.8	129 ±2.1	24.4
base	telco (p)	191 ±98	156 ±3.8	25.0
GH		191 ±73	158 ±3.0	23.8
GH_{NOP}		192 ±93	158 ±3.1	24.5
fork		189 ±71	157 ±3.6	23.4
faasm		332 ±9.9	315 ±0.7	11.3
base	unpack_seq (p)	28.3 ±21	3.33 ±1.2	802
GH		29.6 ±17	5.03 ±2.1	398
GH_{NOP}		28.4 ±22	3.44 ±1.4	836
fork		34.1 ±16	7.71 ±5.7	136
faasm		123 ±4.8	103 ±0.4	29.6
base	richards (p)	387 ±77	353 ±4.6	10.7
GH		385 ±87	351 ±4.4	10.8
GH_{NOP}		391 ±112	353 ±6.2	10.6
fork		396 ±81	360 ±6.2	10.2
faasm		636 ±11	607 ±19	5.86
base	pidigits (p)	2380 ±71	2348 ±5.8	1.64
GH		2380 ±56	2349 ±6.5	1.63
GH_{NOP}		2376 ±44	2347 ±5.6	1.64
fork		2385 ±59	2353 ±6.9	1.62
faasm		7224 ±44	6994 ±16	0.47
base	adi (c)	28470 ±1117	28311 ±923	0.12
GH		28941 ±1226	28858 ±1216	0.12
GH_{NOP}		29088 ±1156	29011 ±1152	0.11
fork		28257 ±1621	28173 ±1602	0.12
faasm		24456 ±2525	19504 ±1178	0.15
base	correlatio (c)	32509 ±694	32430 ±693	0.10
GH		34406 ±1122	34329 ±1118	0.09
GH_{NOP}		34142 ±1038	34056 ±1029	0.10
fork		32964 ±987	32869 ±957	0.10
faasm		25082 ±3381	19377 ±4334	0.14
base	durbin (c)	33.1 ±27	7.64 ±1.4	315
GH		33.5 ±41	8.03 ±1.5	296
GH_{NOP}		32.0 ±19	7.65 ±1.1	325
fork		34.0 ±34	7.98 ±1.4	318
faasm		9.57 ±1.1	5.43 ±0.0	326
base	heat-3d (c)	3088 ±95	3060 ±82	1.02
GH		3305 ±66	3272 ±28	0.98
GH_{NOP}		3090 ±93	3060 ±73	1.02
fork		3319 ±48	3289 ±19	0.98
faasm		8780 ±49	8645 ±19	0.33
base	ludcmp (c)	193637 ±6461	193546 ±6456	0.02
GH		199649 ±8790	199550 ±8783	0.02
GH_{NOP}		199407 ±8662	199297 ±8660	0.02
fork		193763 ±6494	193653 ±6490	0.02
faasm		161293 ±3763	138991 ±10860	0.02
base	trisolv (c)	57.6 ±75	23.1 ±1.5	138
GH		55.8 ±65	23.2 ±2.2	135
GH_{NOP}		54.4 ±71	23.0 ±1.7	137
fork		58.9 ±88	23.2 ±1.7	134
faasm		19.3 ±1.5	11.4 ±0.5	175
base	md2html (pandoc) (n)	69.4 ±52	31.1 ±1.9	93.9
GH		73.8 ±75	32.7 ±2.3	88.5
GH_{NOP}		71.2 ±70	31.3 ±1.8	93.5
fork		77.2 ±62	35.1 ±2.4	68.8
base	base64 (n)	42.7 ±111	3.82 ±1.4	923
GH		39.9 ±80	6.29 ±3.4	122
GH_{NOP}		37.7 ±82	4.47 ±1.3	677
base	ocr-img (n)	686 ±107	644 ±20	5.62
GH		758 ±112	715 ±21	4.34
GH_{NOP}		664 ±118	620 ±29	5.85
base	logging (p)	1288 ±657	1249 ±653	0.00
GH		259 ±68	228 ±5.2	16.3
GH_{NOP}		1319 ±648	1282 ±646	0.00
fork		260 ±73	224 ±4.8	15.6
faasm		383 ±12	345 ±2.6	9.69
base	go (p)	631 ±92	593 ±6.6	6.48
GH		637 ±110	597 ±5.7	6.42
GH_{NOP}		633 ±80	598 ±5.8	6.44
fork		634 ±84	596 ±6.4	6.24
faasm		1014 ±14	982 ±4.9	3.51
base	hexiom (p)	254 ±85	218 ±4.2	17.4
GH		254 ±84	219 ±4.0	17.3
GH_{NOP}		251 ±79	218 ±2.9	17.4
fork		255 ±72	220 ±3.7	16.0
faasm		495 ±10	467 ±1.6	7.60
base	fannkuch (p)	29.7 ±25	4.59 ±1.2	572
GH		31.8 ±26	6.13 ±2.0	350
GH_{NOP}		29.0 ±16	4.70 ±1.5	557
fork		38.4 ±67	8.56 ±5.5	132
faasm		125 ±5.2	105 ±1.1	29.1
base	version (p)	28.2 ±23	3.07 ±1.6	990
GH		29.8 ±28	4.05 ±1.5	563
GH_{NOP}		27.9 ±27	3.03 ±1.2	925
fork		32.3 ±22	6.81 ±3.8	265
faasm		11.0 ±0.8	3.89 ±0.0	254
base	scimark (p)	1848 ±79	1813 ±31	2.12
GH		1841 ±75	1807 ±28	2.12
GH_{NOP}		1838 ±85	1801 ±28	2.14
fork		1868 ±77	1833 ±23	2.10
faasm		3513 ±25	3482 ±5.9	0.97
base	atax (c)	68.7 ±64	36.5 ±1.6	93.5
GH		68.4 ±65	36.8 ±2.0	92.0
GH_{NOP}		70.6 ±75	36.5 ±1.7	93.1
fork		70.4 ±75	36.7 ±1.6	91.5
faasm		30.3 ±2.0	22.2 ±0.8	118
base	covariance (c)	33092 ±496	33021 ±495	0.10
GH		35055 ±1081	34971 ±1084	0.10
GH_{NOP}		34858 ±942	34776 ±948	0.10
fork		33715 ±1191	33641 ±1189	0.10
faasm		24674 ±3531	17964 ±4700	0.15
base	fdtd-2d (c)	2210 ±62	2179 ±24	0.89
GH		2213 ±58	2183 ±20	0.89
GH_{NOP}		2216 ±69	2185 ±22	0.89
fork		2209 ±60	2176 ±21	0.89
faasm		2856 ±24	2695 ±12	0.87
base	jacobi-1d (c)	27.9 ±18	3.81 ±1.2	671
GH		30.6 ±49	4.15 ±1.6	579
GH_{NOP}		28.9 ±42	3.87 ±1.2	652
fork		33.6 ±63	4.10 ±1.4	604
faasm		8.27 ±1.1	4.01 ±0.0	359
base	mvt (c)	176 ±87	140 ±3.1	28.8
GH		178 ±89	144 ±3.2	28.3
GH_{NOP}		179 ±81	145 ±3.6	28.0
fork		178 ±92	143 ±3.8	28.9
faasm		108 ±5.3	76.7 ±7.1	36.1
base	gset-time (p)	29.6 ±34	2.94 ±1.2	1039
GH		30.4 ±35	4.15 ±1.7	552
GH_{NOP}		27.9 ±20	3.03 ±1.4	1004
fork		35.0 ±40	7.10 ±4.1	257
base	base64 (p)	785 ±110	743 ±7.1	5.18
GH		806 ±107	761 ±10	5.10
GH_{NOP}		786 ±102	745 ±7.0	5.19
fork		807 ±100	764 ±9.3	5.05
base	json (n)	71.1 ±101	9.39 ±3.5	159
GH		72.7 ±79	16.1 ±4.9	86.6
GH_{NOP}		72.5 ±89	13.6 ±3.6	147
base	pyaes (p)	4707 ±99	4672 ±64	0.82
GH		4788 ±93	4751 ±61	0.80
GH_{NOP}		4734 ±100	4699 ±65	0.81
fork		4715 ±98	4678 ±53	0.81
faasm		8721 ±85	8559 ±73	0.40
base	mdp (p)	6377 ±88	6346 ±64	0.59
GH		6444 ±99	6412 ±82	0.58
GH_{NOP}		6395 ±103	6362 ±82	0.58
fork		6419 ±98	6387 ±81	0.58
faasm		12422 ±105	12295 ±82	0.24
base	nbody (p)	2859 ±96	2824 ±69	1.34
GH		2881 ±103	2845 ±53	1.34
GH_{NOP}		2869 ±90	2835 ±55	1.34
fork		2882 ±98	2847 ±60	1.33
faasm		5471 ±38	5361 ±26	0.63
base	json_dumps (p)	567 ±76	533 ±6.0	7.19
GH		586 ±86	551 ±9.9	6.95
GH_{NOP}		586 ±101	549 ±8.2	7.11
fork		580 ±105	541 ±6.8	6.86
faasm		939 ±14	900 ±5.1	3.94
base	float (p)	57.3 ±65	27.1 ±1.9	126
GH		57.9 ±56	27.9 ±1.9	109
GH_{NOP}		56.2 ±51	27.1 ±1.9	125
fork		62.9 ±69	29.3 ±2.0	73.9
faasm		162 ±5.7	141 ±1.3	22.5
base	2mm (c)	27390 ±1548	27236 ±1544	0.12
GH		28963 ±1715	28887 ±1712	0.10
GH_{NOP}		28147 ±1908	28065 ±1906	0.13
fork		28274 ±2212	28183 ±2193	0.11
faasm		24181 ±2903	20590 ±2777	0.14
base	bicg (c)	75.9 ±75	42.8 ±1.9	81.0
GH		77.1 ±80	43.2 ±2.0	79.9
GH_{NOP}		75.7 ±75	42.7 ±1.8	80.6
fork		75.5 ±71	43.3 ±2.0	79.3
faasm		34.4 ±2.2	25.9 ±0.7	105
base	deriche (c)	1148 ±108	1115 ±86	4.47
GH		1148 ±104	1115 ±77	4.43
GH_{NOP}		1119 ±112	1086 ±90	4.34
fork		1146 ±114	1114 ±86	4.50
faasm		919 ±76	674 ±18	4.26
base	floyd-wars (c)	21225 ±42	21151 ±39	0.17
GH		21243 ±39	21171 ±37	0.17
GH_{NOP}		21248 ±110	21158 ±44	0.17
fork		21234 ±66	21158 ±57	0.17
faasm		23356 ±165	21840 ±35	0.11
base	jacobi-2d (c)	2357 ±40	2329 ±17	1.05
GH		2378 ±76	2343 ±15	1.05
GH_{NOP}		2371 ±71	2338 ±17	1.04
fork		2368 ±63	2338 ±19	1.05
faasm		5077 ±57	4971 ±25	0.71
base	nussinov (c)	39327 ±4277	39123 ±4053	0.09
GH		38398 ±830	38324 ±827	0.09
GH_{NOP}		38163 ±1059	38086 ±1058	0.08
fork		36208 ±1035	36134 ±1033	0.09
faasm		38477 ±3146	30232 ±1793	0.09
base	sentiment (p)	32.7 ±25	6.47 ±1.8	385
GH		34.9 ±24	8.88 ±3.2	230
GH_{NOP}		32.6 ±27	6.64 ±1.8	385
fork		43.0 ±40	11.7 ±3.7	44.1
base	primes (p)	1867 ±89	1830 ±53	2.04
GH		1870 ±119	1831 ±75	1.99
GH_{NOP}		1860 ±105	1820 ±55	1.99
fork		1896 ±91	1857 ±35	1.99
base	primes (n)	317 ±109	275 ±20	11.8
GH		327 ±111	287 ±23	8.16
GH_{NOP}		319 ±101	279 ±20	11.8
base	spectral (p)	631 ±84	593 ±9.9	6.45
GH		642 ±87	605 ±14	6.40
GH_{NOP}		636 ±92	599 ±16	6.52
fork		638 ±83	602 ±13	6.24
faasm		1367 ±18	1323 ±5.1	2.62
base	pyflate (p)	1636 ±76	1600 ±16	2.39
GH		1660 ±74	1623 ±13	2.34
GH_{NOP}		1645 ±69	1611 ±16	2.38
fork		1663 ±92	1623 ±11	2.32
faasm		2780 ±30	2644 ±5.6	1.26
base	raytrace (p)	2496 ±107	2459 ±67	1.58
GH		2500 ±96	2464 ±51	1.57
GH_{NOP}		2494 ±80	2460 ±47	1.57
fork		2490 ±73	2458 ±44	1.56
faasm		4070 ±38	4001 ±40	0.83
base	pickle (p)	139 ±80	106 ±1.9	35.5
GH		140 ±109	106 ±2.1	35.0
GH_{NOP}		139 ±69	105 ±1.8	35.4
fork		139 ±61	107 ±2.2	32.7
faasm		210 ±6.8	184 ±1.2	17.6
base	json_loads (p)	135 ±82	102 ±1.9	36.5
GH		136 ±70	103 ±2.1	35.3
GH_{NOP}		136 ±73	103 ±2.1	36.3
fork		139 ±80	104 ±2.2	30.7
faasm		286 ±8.5	252 ±2.2	13.2
base	3mm (c)	45948 ±1849	45729 ±1717	0.07
GH		46901 ±2476	46824 ±2473	0.06
GH_{NOP}		47007 ±2714	46929 ±2716	0.07
fork		45736 ±2720	45661 ±2720	0.06
faasm		38270 ±3312	31627 ±2653	0.09
base	cholesky (c)	166285 ±9224	166183 ±9209	0.02
GH		175803 ±8275	175692 ±8256	0.02
GH_{NOP}		177333 ±8313	177223 ±8310	0.02
fork		170728 ±11479	170629 ±11483	0.02
faasm		140259 ±21552	112430 ±8753	0.02
base	doitgen (c)	691 ±101	651 ±15	5.98
GH		691 ±94	650 ±15	5.96
GH_{NOP}		688 ±118	646 ±13	6.01
fork		694 ±102	652 ±15	5.95
faasm		677 ±10	662 ±1.6	5.55
base	gramschmid (c)	61227 ±6114	60900 ±6020	0.06
GH		65076 ±2157	64980 ±2151	0.05
GH_{NOP}		62882 ±3330	62799 ±3331	0.06
fork		62591 ±1577	62507 ±1570	0.05
faasm		45304 ±2867	44627 ±2702	0.07
base	lu (c)	196660 ±11451	196556 ±11445	0.02
GH		207712 ±13029	207604 ±13014	0.02
GH_{NOP}		206484 ±10894	206382 ±10885	0.02
fork		200061 ±11097	199953 ±11092	0.02
faasm		160516 ±4122	138303 ±6926	0.02
base	seidel-2d (c)	23186 ±194	23140 ±22	0.16
GH		23173 ±46	23139 ±21	0.16
GH_{NOP}		23177 ±46	23142 ±20	0.16
fork		23180 ±51	23145 ±20	0.17
faasm		19062 ±88	18836 ±32	0.18
base	json (p)	71.0 ±87	9.85 ±3.4	150
GH		72.1 ±73	13.0 ±4.0	135
GH_{NOP}		67.6 ±77	10.0 ±3.4	152
fork		78.8 ±89	15.8 ±4.6	120
base	get-time (n)	36.8 ±78	3.70 ±1.3	942
GH		39.3 ±87	6.37 ±3.6	133
GH_{NOP}		35.1 ±65	4.55 ±1.2	779
base	img-resize (n)	506 ±132	445 ±74	6.57
GH		780 ±149	722 ±111	4.10
GH_{NOP}		556 ±148	491 ±68	6.18

Table A.1.: Latency and throughput measurements comparing Groundhog to several other systems: **base** baseline OpenWhisk; **GH** Groundhog on OpenWhisk; **faasm** faasm; **fork** fork-based implementation on OpenWhisk; and **GH_{NOP}** Groundhog with no restoration. We run 58 benchmarks across three languages indicated by **(p)** Python, **(c)** C, and **(n)** Node.js. We highlight cells of interest if there is more than a 5% difference as follows: Green indicates results faorable to Groundhog Red indicates results unfaorable to Groundhog Blue indicates the unexpected result that Groundhog is outperforming the baseline. ***Disclaimer***: Faasm throughput measurements are provided for the curious reader, but no conclusions should be drawn based on them as they entangle many variables such as the difference in the platforms, their internal components and deployment, runtimes (native vs WebAssembly), as well as the isolation mechanism.

Benchmark	Base End-to-End Latency (ms) / rel. overhead (%) ±CoV(%)					Throughput (r/s) / rel. overhead (%)					Inv. lat (ms)/rel. (%) ± CoV (%)		Restoration	
	baseline	GH_{NOP}	GH	fork	faasm	baseline	GH_{NOP}	GH	fork	faasm	baseline	GH	time (ms)	#Kpages (%))
chaos (p)	688.18±20.8	+0.12 ±16.8	+0.36 ±16.1	+0.22 ±15.3	+79.45 ±1.4	6.03	-0.90	-1.40	-3.70	-50.40	648.52±13.3	+0.54 ±6.0	4.9	0.47 (7.4)
logging (p)	1287.57±51.0	+2.43 ±49.1	-79.89 ±26.1	-79.82 ±28.3	-70.26 ±3.1	0	0	+inf	+inf	+inf	1249.39±52.2	-81.76 ±2.3	4.8	0.41 (6.7)
pyaes (p)	4707.26±2.1	+0.57 ±2.1	+1.71 ±1.9	+0.16 ±2.1	+85.26 ±1.0	0.82	-0.40	-1.30	-1.00	-50.80	4671.98±1.4	+1.70 ±1.3	6	0.84 (13.5)
spectral (p)	630.84±13.4	+0.87 ±14.5	+1.84 ±13.6	+1.20 ±13.0	+116.72 ±1.3	6.45	+1.1	-0.80	-3.20	-59.30	592.76±1.7	+2.09 ±2.3	4.3	0.21 (3.4)
deltablue (p)	48.44±116.6	-2.97 ±71.6	+3.88 ±102.6	+9.51 ±103.8	+210.47 ±3.8	157.63	-1.70	-11.00	-46.30	-84.50	20.43±7.9	+4.33 ±9.5	4.6	0.33 (5.3)
go (p)	631.16±14.5	+0.32 ±12.7	+0.91 ±17.3	+0.41 ±13.2	+60.67 ±1.4	6.48	-0.60	-0.90	-3.60	-45.80	592.99±1.1	+0.61 ±1.0	6.9	0.95 (15.2)
mdp (p)	6377.46±1.4	+0.27 ±1.6	+1.05 ±1.5	+0.65 ±1.5	+94.77 ±0.8	0.59	-0.60	-1.00	-1.40	-58.80	6345.53±1.0	+1.05 ±1.3	9.6	2.85 (38.9)
pyflate (p)	1635.93±4.7	+0.53 ±4.2	+1.45 ±4.5	+1.65 ±5.5	+69.92 ±1.1	2.39	-0.60	-2.10	-3.20	-47.30	1599.84±1.0	+1.42 ±0.8	11.7	2.33 (28.2)
telco (p)	190.81±51.5	+0.80 ±48.2	+0.30 ±38.2	-0.83 ±37.3	+74.13 ±3.0	25.01	-2.00	-4.90	-6.50	-54.70	155.64±2.4	+1.54 ±1.9	3.9	0.53 (16.2)
hexiom (p)	253.92±33.5	-1.18 ±31.6	+0.13 ±33.2	+0.58 ±28.0	+94.80 ±2.1	17.45	-0.10	-1.00	-8.30	-56.50	218.21±1.9	+0.45 ±1.8	4.3	0.28 (4.5)
nbody (p)	2858.53±3.4	+0.38 ±3.1	+0.78 ±3.6	+0.83 ±3.4	+91.39 ±0.7	1.34	+0.0	-0.10	-0.70	-53.20	2823.65±2.4	+0.76 ±1.9	4.1	0.21 (3.4)
raytrace (p)	2495.66±4.3	-0.06 ±3.2	+0.17 ±3.8	-0.22 ±2.9	+63.10 ±0.9	1.58	-0.30	-0.60	-1.10	-47.60	2459.2±2.7	+0.19 ±2.1	4.4	0.35 (5.6)
unpack_seq (p)	28.34±74.3	+0.27 ±75.8	+4.45 ±56.3	+20.19 ±47.5	+333.81 ±3.9	801.86	+4.3	-50.40	-83.00	-96.30	3.32±36.8	+51.21 ±40.9	3.2	0.2 (3.3)
fannkuch (p)	29.7±83.6	-2.46 ±56.3	+7.04 ±83.3	+29.26 ±173.9	+319.97 ±4.2	572.32	-2.80	-38.80	-77.00	-94.90	4.59±27.1	+33.77 ±32.7	3.1	0.19 (3.1)
json_dumps (p)	567.41±13.3	+3.26 ±17.3	+3.19 ±14.6	+2.13 ±18.2	+65.41 ±1.5	7.19	-1.10	-3.40	-4.70	-45.30	533.09±1.1	+3.45 ±1.8	4.9	0.51 (8.0)
pickle (p)	139.26±57.7	+0.01 ±49.5	+0.64 ±77.5	-0.07 ±43.6	+50.49 ±3.2	35.49	-0.20	-1.40	-7.80	-50.60	105.64±1.8	+0.01 ±2.0	2.9	0.23 (6.7)
richards (p)	387.47±19.9	+1.01 ±28.7	-0.66 ±22.7	+2.22 ±20.4	+64.22 ±1.7	10.68	-0.50	+1.6	-4.10	-45.20	353.13±1.3	-0.56 ±1.3	4.2	0.23 (3.7)
version (p)	28.24±82.1	-1.35 ±98.6	+5.33 ±92.7	+14.34 ±66.8	-61.13 ±7.1	990.38	-6.60	-43.20	-73.30	-74.30	3.07±50.5	+31.82 ±36.0	1.7	0.17 (5.4)
float (p)	57.28±112.7	-1.80 ±90.7	+1.07 ±96.4	+9.73 ±110.0	+182.82 ±3.5	125.98	-0.80	-13.40	-41.30	-82.10	27.06±7.1	+2.91 ±6.7	5	0.65 (10.4)
json_loads (p)	135.04±60.7	+0.87 ±53.4	+1.04 ±51.2	+3.03 ±57.3	+111.67 ±3.0	36.46	-0.50	-3.20	-15.70	-63.70	101.98±1.9	+1.33 ±2.1	4	0.22 (3.6)
pidigits (p)	2380.0±3.0	-0.19 ±1.8	-0.01 ±2.4	+0.20 ±2.5	+203.53 ±0.6	1.64	-0.20	-0.50	-1.00	-71.50	2347.55±0.2	+0.07 ±0.3	5.4	0.81 (13.2)
scimark (p)	1848.12±4.3	-0.54 ±4.6	-0.37 ±4.1	+1.09 ±4.1	+90.08 ±0.7	2.12	+0.7	+0.0	-0.90	-54.40	1812.64±1.7	-0.34 ±1.6	3.8	0.52 (16.0)
2mm (c)	27390.28±5.7	+2.76 ±6.8	+5.74 ±5.9	+3.22 ±7.8	-11.72 ±12.0	0.12	+0.6	-16.10	-12.50	+10.7	27236.21±5.7	+6.06 ±5.9	3.1	0.02 (2.0)
3mm (c)	45947.69±4.0	+2.30 ±5.8	+2.07 ±5.3	-0.46 ±5.9	-16.71 ±8.7	0.07	-1.80	-8.80	-8.80	+23.4	45729.02±3.8	+2.40 ±5.3	2.3	0.02 (2.0)
adi (c)	28470.33±3.9	+2.17 ±4.0	+1.65 ±4.2	-0.75 ±5.7	-14.10 ±10.3	0.12	-5.80	-2.60	-2.80	+24.6	28311.08±3.3	+1.93 ±4.2	0.8	0.02 (2.0)
atax (c)	68.72±92.5	+2.78 ±105.9	-0.42 ±94.3	+2.52 ±106.4	-55.96 ±6.5	93.55	-0.50	-1.70	-2.20	+25.8	36.45±4.4	+1.01 ±5.4	1	0.03 (3.1)
bicg (c)	75.89±98.7	-0.20 ±98.5	+1.59 ±103.1	-0.45 ±93.9	-54.72 ±6.3	81.05	-0.50	-1.40	-2.10	+29.1	42.78±4.4	+0.89 ±4.7	0.9	0.03 (3.1)
cholesky (c)	166284.84±5.5	+6.64 ±4.7	+5.72 ±4.7	+2.67 ±6.7	-15.65 ±15.4	0.02	-8.60	-1.00	-2.60	+0.4	166182.8±5.5	+5.72 ±4.7	0.6	0.02 (2.0)
correlation (c)	32508.82±2.1	+5.02 ±3.0	+5.84 ±3.3	+1.40 ±3.0	-22.85 ±13.5	0.1	+2.4	-6.20	-5.90	+43.1	32429.64±2.1	+5.86 ±3.3	2	0.02 (2.0)
covariance (c)	33092.13±1.5	+5.34 ±2.7	+5.93 ±3.1	+1.88 ±3.5	-25.44 ±14.3	0.1	-3.80	+0.8	+0.4	+48.3	33020.56±1.5	+5.91 ±3.1	2	0.02 (2.0)
deriche (c)	1148.32±9.4	-2.56 ±10.1	-0.01 ±9.0	-0.16 ±9.9	-19.94 ±8.3	4.47	-3.00	-1.00	+0.6	-4.70	1114.99±7.7	+0.00 ±6.9	0.8	0.02 (2.0)
doitgen (c)	691.08±14.7	-0.44 ±17.1	-0.01 ±13.7	+0.45 ±14.8	-2.00 ±1.5	5.98	+0.6	-0.30	-0.40	-7.00	650.53±2.2	-0.08 ±2.3	1.3	0.02 (2.0)
durbin (c)	33.1±82.8	-3.25 ±59.1	+1.31 ±122.9	+2.64 ±100.6	-71.09 ±11.8	314.68	+3.2	-6.00	+1.0	+3.8	7.64±17.6	+5.05 ±18.4	0.6	0.02 (2.0)
fdtd-2d (c)	2209.61±2.8	+0.29 ±3.1	+0.15 ±2.6	-0.05 ±2.7	+29.23 ±0.9	0.89	-0.50	-0.30	+0.1	-2.30	2179.15±1.1	+0.16 ±0.9	1	0.02 (2.0)
floyd-warshall (c)	21224.8±0.2	+0.11 ±0.5	+0.09 ±0.2	+0.05 ±0.3	+10.04 ±0.7	0.17	-0.30	-1.80	-1.20	-35.20	21151.44±0.2	+0.09 ±0.2	0.8	0.02 (2.0)
gramschmidt (c)	61226.6±10.0	+2.70 ±5.3	+6.29 ±3.3	+2.23 ±2.5	-26.01 ±6.3	0.06	-1.10	-11.10	-6.20	+18.5	60899.77±9.9	+6.70 ±3.3	2.5	0.02 (2.0)
heat-3d (c)	3088.12±3.1	+0.07 ±3.0	+7.02 ±2.0	+7.47 ±1.5	+184.32 ±0.6	1.02	-0.40	-4.00	-4.10	-67.60	3059.55±2.7	+6.94 ±0.9	16.1	3.39 (77.9)
jacobi-1d (c)	27.92±63.9	+3.34 ±144.5	+9.65 ±159.4	+20.44 ±188.8	-70.39 ±12.8	671.34	-2.80	-13.80	-10.00	-46.60	3.81±32.7	+9.01 ±39.4	0.6	0.02 (2.0)
jacobi-2d (c)	2356.66±1.7	+0.62 ±3.0	+0.90 ±3.2	+0.49 ±2.7	+115.44 ±1.1	1.05	-0.20	0.00	+0.3	-32.10	2329.32±0.7	+0.60 ±0.6	0.7	0.02 (2.0)
lu (c)	196660.22±5.8	+5.00 ±5.3	+5.62 ±6.3	+1.73 ±5.5	-18.38 ±2.6	0.02	+5.8	+2.2	+3.3	+11.6	196555.78±5.8	+5.62 ±6.3	0.7	0.02 (2.0)
ludcmp (c)	193637.44±3.3	+2.98 ±4.3	+3.10 ±4.4	+0.06 ±3.4	-16.70 ±2.3	0.02	+0.1	+7.3	+7.0	+6.9	193545.91±3.3	+3.10 ±4.4	1	0.02 (2.0)
mvt (c)	176.37±49.5	+1.40 ±45.2	+0.89 ±49.9	+0.75 ±51.9	-38.62 ±4.9	28.78	-2.60	-1.70	+0.4	+25.3	140.33±2.2	+2.86 ±2.2	1.2	0.03 (3.1)
nussinov (c)	39326.91±10.9	-2.96 ±2.8	-2.36 ±2.2	-7.93 ±2.9	-2.16 ±8.2	0.09	-6.40	-3.00	-0.90	+2.0	39122.65±10.4	-2.04 ±2.2	1	0.02 (2.0)
seidel-2d (c)	23186.15±0.8	-0.04 ±0.2	-0.06 ±0.2	-0.02 ±0.2	-17.79 ±0.5	0.16	-0.50	+0.0	+0.6	+9.7	23140.14±0.1	-0.01 ±0.1	0.8	0.02 (2.0)
trisolv (c)	57.62±130.2	-5.65 ±130.4	-3.14 ±115.6	+2.28 ±149.3	-66.54 ±7.7	138.18	-0.80	-2.40	-2.80	+26.6	23.07±6.6	+0.42 ±9.3	1	0.02 (2.0)
get-time (p)	29.6±113.6	-5.68 ±72.6	+2.71 ±114.8	+18.13 ±114.1		1038.74	-3.30	-46.90	-75.30		2.94±40.5	+41.08 ±41.0	1.7	0.18 (5.6)
sentiment (p)	32.67±76.3	-0.10 ±82.0	+6.66 ±68.5	+31.69 ±92.4		385.07	0.00	-40.20	-88.50		6.47±27.2	+37.26 ±35.8	6	0.57 (3.4)
json (p)	70.97±122.9	-4.81 ±113.7	+1.56 ±101.8	+11.04 ±113.0		150	+1.5	-9.80	-19.80		9.85±34.2	+31.68 ±30.6	3.7	0.87 (26.1)
md2html (p)	69.36±75.3	+2.66 ±98.6	+6.36 ±101.6	+11.38 ±80.0		93.94	-0.50	-5.80	-26.80		31.04±6.3	+5.46 ±7.1	4.2	0.62 (12.6)
base64 (p)	785.33±14.0	+0.03 ±13.0	+2.63 ±13.3	+2.81 ±12.4		5.18	+0.2	-1.50	-2.50		743.23±1.0	+2.45 ±1.4	7.7	1.66 (32.4)
primes (p)	1866.58±4.8	-0.34 ±5.6	+0.17 ±6.3	+1.56 ±4.8		2.04	-2.40	-2.30	-2.40		1829.74±2.9	+0.05 ±4.1	3.2	0.53 (16.5)
get-time (n)	36.84±211.2	-4.69 ±184.6	+6.68 ±220.4			942.07	-17.30	-85.80			3.7±34.8	+72.22 ±56.2	12.6	0.64 (0.4)
autocomplete (n)	42.74±260.6	-11.90 ±217.9	-6.71 ±201.2			922.59	-26.60	-86.80			3.82±36.8	+64.58 ±54.2	13.5	0.92 (0.6)
json (n)	71.1±142.4	+1.99 ±122.7	+2.22 ±108.5			159.09	-7.50	-45.60			9.4±37.8	+71.32 ±30.7	13	0.85 (0.5)
primes (n)	316.85±34.3	+0.65 ±31.7	+3.27 ±33.9			11.79	+0.1	-30.70			274.63±7.3	+4.56 ±8.0	84.7	34.2 (17.0)
img-resize (n)	505.76±26.0	+9.87 ±26.6	+54.20 ±19.2			6.57	-6.00	-37.60			445.27±16.7	+62.09 ±15.3	61.8	18.05 (10.1)
base64 (n)	686.33±15.6	-3.19 ±17.8	+10.48 ±14.7			5.62	+4.1	-22.80			644.02±3.1	+11.04 ±2.9	161.9	53.83 (25.8)
ocr-img (n)	2539.62±3.9	+0.43 ±4.2	+0.60 ±3.8			1.53	-0.40	-1.00			2491.66±0.4	+0.68 ±0.5	13.9	1.08 (0.7)

Table A.2.: Latency and throughput measurements showing the overheads of GH_{NOP} (Groundhog with no restoration), GH, FORK, and FAASM relative to an unsecure baseline. We run 58 benchmarks across three languages indicated by **(p)** Python, **(c)** C, and **(n)** Node.js. We highlight cells of interest if there is more than a 5% difference as follows: Green indicates results faorable to Groundhog Red indicates results unfaorable to Groundhog Blue indicates the unexpected result that Groundhog is outperforming the baseline. GH's impact on throughput can be approximated by the relative restoration time compared to the invoker's latency (inv. lat.). ***Disclaimer***: Faasm throughput measurements are provided for the curious reader, but no conclusions should be drawn based on them as they entangle many variables such as the difference in the platforms, their internal components and deployment, runtimes (native vs WebAssembly), as well as the isolation mechanism.

Benchmark	*baseline*		*Groundhog*		*Restoration*			
	Invoker lat (ms)	T'put	Invoker lat (ms)	T'put	time (ms)	#pages (K)	#faults (K)	#restored (K)
cholesky (c)	166182.8±9208.73	0.02	175691.9±8256.49	0.02	0.57	0.98	0.02	0.01
jacobi-1d (c)	3.8±1.25	671.34	4.2±1.64	578.99	0.62	0.98	0.03	0.02
durbin (c)	7.6±1.35	314.68	8.0±1.48	295.98	0.62	0.98	0.03	0.02
jacobi-2d (c)	2329.3±17.03	1.05	2343.4±14.98	1.05	0.69	0.98	0.02	0.01
lu (c)	196555.8±11445.0	0.02	207603.5±13014.02	0.02	0.74	0.98	0.02	0.01
seidel-2d (c)	23140.1±22.03	0.16	23139.0±21.4	0.16	0.75	0.98	0.02	0.02
deriche (c)	1115.0±86.2	4.47	1115.0±76.95	4.43	0.75	0.98	0.02	0.01
adi (c)	28311.1±923.24	0.12	28857.6±1215.98	0.12	0.77	0.98	0.02	0.02
floyd-warshall (c)	21151.4±39.35	0.17	21171.3±37.12	0.17	0.78	0.98	0.02	0.01
bicg (c)	42.8±1.88	81.05	43.2±2.03	79.87	0.93	0.98	0.03	0.03
fdtd-2d (c)	2179.1±23.85	0.89	2182.6±19.73	0.89	0.97	0.98	0.02	0.02
trisolv (c)	23.1±1.51	138.18	23.2±2.16	134.92	0.97	0.98	0.03	0.02
atax (c)	36.4±1.6	93.55	36.8±1.99	91.99	0.99	0.98	0.03	0.03
nussinov (c)	39122.6±4053.11	0.09	38323.5±827.3	0.09	1.02	0.98	0.02	0.02
ludcmp (c)	193545.9±6455.96	0.02	199550.2±8782.81	0.02	1.02	0.98	0.03	0.02
mvt (c)	140.3±3.06	28.78	144.3±3.2	28.28	1.16	0.98	0.04	0.03
doitgen (c)	650.5±14.61	5.98	650.0±14.79	5.96	1.31	0.98	0.04	0.02
version (p)	3.1±1.55	990.38	4.0±1.46	562.89	1.66	3.14	0.17	0.17
get-time (p)	2.9±1.19	1038.74	4.1±1.7	552.09	1.66	3.19	0.18	0.18
covariance (c)	33020.6±494.9	0.10	34971.3±1084.18	0.10	1.97	0.98	0.04	0.02
correlation (c)	32429.6±692.85	0.10	34328.9±1118.18	0.09	2.00	0.98	0.04	0.02
3mm (c)	45729.0±1717.42	0.07	46824.4±2473.21	0.06	2.32	0.98	0.04	0.02
gramschmidt (c)	60899.8±6020.33	0.06	64980.4±2150.99	0.05	2.53	0.98	0.04	0.02
pickle (p)	105.6±1.89	35.49	105.7±2.11	34.98	2.90	3.45	0.23	0.23
2mm (c)	27236.2±1544.4	0.12	28887.4±1712.35	0.10	3.12	0.98	0.04	0.02
fannkuch (p)	4.6±1.24	572.32	6.1±2.0	350.22	3.14	6.12	0.19	0.19
unpack_seq (p)	3.3±1.22	801.86	5.0±2.06	398.15	3.17	6.12	0.20	0.20
primes (p)	1829.7±53.45	2.04	1830.7±75.43	1.99	3.24	3.22	0.51	0.53
json (p)	9.9±3.37	150.00	13.0±3.97	135.34	3.71	3.33	0.64	0.87
scimark (p)	1812.6±30.71	2.12	1806.6±28.47	2.12	3.77	3.26	0.51	0.52
telco (p)	155.6±3.8	25.01	158.0±2.95	23.77	3.91	3.29	0.53	0.53
json_loads (p)	102.0±1.95	36.46	103.3±2.14	35.29	4.04	6.12	0.22	0.22
nbody (p)	2823.7±69.0	1.34	2845.0±53.46	1.34	4.08	6.12	0.21	0.21
richards (p)	353.1±4.64	10.68	351.1±4.41	10.85	4.16	6.18	0.23	0.23
md2html (p)	31.0±1.95	93.94	32.7±2.31	88.50	4.25	4.93	0.63	0.62
spectral (p)	592.8±9.92	6.45	605.2±14.14	6.40	4.29	6.12	0.22	0.21
hexiom (p)	218.2±4.21	17.45	219.2±3.98	17.28	4.35	6.18	0.28	0.28
raytrace (p)	2459.2±67.26	1.58	2463.9±51.19	1.57	4.42	6.25	0.36	0.35
deltablue (p)	20.4±1.61	157.63	21.3±2.02	140.26	4.64	6.18	0.23	0.33
logging (p)	1249.4±652.55	0.00	227.9±5.2	16.34	4.77	6.12	0.42	0.41
json_dumps (p)	533.1±6.0	7.19	551.5±9.92	6.95	4.92	6.37	0.51	0.51
chaos (p)	648.5±86.06	6.03	652.0±39.21	5.94	4.93	6.32	0.47	0.47
float (p)	27.1±1.92	125.98	27.8±1.87	109.09	4.99	6.26	0.65	0.65
pidigits (p)	2347.6±5.76	1.64	2349.1±6.51	1.63	5.40	6.14	0.81	0.81
sentiment (p)	6.5±1.76	385.07	8.9±3.18	230.39	6.00	16.86	0.57	0.57
pyaes (p)	4672.0±63.68	0.82	4751.3±61.36	0.80	6.02	6.21	0.83	0.84
go (p)	593.0±6.64	6.48	596.6±5.69	6.42	6.90	6.25	0.84	0.95
base64 (p)	743.2±7.11	5.18	761.5±10.48	5.10	7.67	5.13	1.86	1.66
mdp (p)	6345.5±63.96	0.59	6412.3±82.13	0.58	9.55	7.33	2.22	2.85
pyflate (p)	1599.8±16.39	2.39	1622.5±12.58	2.34	11.67	8.25	3.01	2.33
get-time (n)	3.7±1.29	942.07	6.4±3.58	133.45	12.58	156.76	0.59	0.64
json (n)	9.4±3.55	159.09	16.1±4.94	86.58	13.02	156.78	0.67	0.85
autocomplete (n)	3.8±1.41	922.59	6.3±3.41	121.98	13.52	156.98	0.69	0.92
ocr-img (n)	2491.7±10.63	1.53	2508.5±12.24	1.52	13.95	156.80	0.89	1.08
heat-3d (c)	3059.5±81.59	1.02	3272.0±28.01	0.98	16.09	4.35	0.02	3.39
img-resize (n)	445.3±74.34	6.57	721.7±110.76	4.10	61.83	179.43	9.58	18.05
primes (n)	274.6±20.11	11.79	287.1±23.1	8.16	84.74	201.35	1.27	34.20
base64 (n)	644.0±20.22	5.62	715.1±20.89	4.34	161.93	208.42	47.98	53.83

Table A.3.: Groundhog's overhead on the throughput is a function of Groundhog's added overhead both on the critical path (#faults) which has negligible overhead on most functions, as well as the overhead off the critical path (Restoration time) which is mostly a function of the address space size (#pages (K)) and the number of restored pages (#restored (K)) as well as restoring the memory layout as demonstrated earlier in Fig. 3.8. Data is sorted by the restoration time.

Bibliography

[1] 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`. Ret. 12.02.2023 (cit. on p. 8).

[2] Mania Abdi, Samuel Ginzburg, Xiayue Charles Lin, et al. "Palette Load Balancing: Locality Hints for Serverless Functions". In: *Proceedings of the Eighteenth European Conference on Computer Systems (EuroSys'23)*. New York, NY, USA: Association for Computing Machinery, 2023, 365–380 (cit. on p. 110).

[3] Alexandru Agache, Marc Brooker, Alexandra Iordache, et al. "Firecracker: Lightweight Virtualization for Serverless Applications". In: *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 2020, pp. 419–434 (cit. on pp. 2, 16).

[4] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, et al. "SAND: Towards High-Performance Serverless Computing". In: *2018 Usenix Annual Technical Conference (USENIX ATC 18)*. 2018, pp. 923–935 (cit. on pp. 2, 16, 65).

[5] Mohamed Alzayat, Jonathan Mace, Peter Druschel, and Deepak Garg. "Groundhog: Efficient Request Isolation in FaaS". In: *Proceedings of the Eighteenth European Conference on Computer Systems (EuroSys'23)*. EuroSys '23. Rome, Italy: Association for Computing Machinery, 2023, 398–415 (cit. on p. 13).

[6] Alzayat, Mohamed and Mace, Jonathan and Druschel, Peter and Garg, Deepak. *Groundhog Project Website*. `https://groundhog.mpi-sws.org/`. 2023 (cit. on pp. 27, 31).

[7] Amazon AWS. *AWS Lambda*. `https://aws.amazon.com/lambda/`. Ret. 04.02.2023 (cit. on pp. 2, 6, 16, 70).

[8] Amazon AWS. *AWS Lambda: Provisioned concurrency*. `https://aws.amazon.com/blogs/aws/new-provisioned-concurrency-for-lambda-functions/`. Accessed 28.11.2023 (cit. on p. 65).

[9] Lixiang Ao, George Porter, and Geoffrey M Voelker. "FaaSnap: FaaS made fast using snapshot-based VMs". In: *Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys'22)*. 2022, pp. 730–746 (cit. on pp. 8, 32, 62, 66, 70).

[10] Apache. *Apache MPM prefork*. `https://httpd.apache.org/docs/2.4/mod/prefork.html`. Ret. 12.02.2023 (cit. on pp. 9, 64).

[11] Apache. *OpenWhisk*. `https://openwhisk.apache.org/`. Ret. 05.02.2023 (cit. on pp. 2, 16).

[12] Apache. *Pre-Warmed actions in Openwhisk*. https://github.com/apache/openwhisk/blob/master/docs/actions.md/. Accessed 28.11.2023 (cit. on p. 65).

[13] 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. Ret. 12.02.2023 (cit. on p. 8).

[14] Apache Tomcat Security team. *CVE-2022-25762: Apache Tomcat request mix-up*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-25762. Ret. 12.02.2023 (cit. on p. 8).

[15] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. 1.00. 2018 (cit. on p. 15).

[16] Thomas H Austin and Cormac Flanagan. "Permissive dynamic information flow analysis". In: *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*. 2010, pp. 1–12 (cit. on pp. 72, 76, 77).

[17] AWS Lambda. *Implementing statelessness in functions*. https://docs.aws.amazon.com/lambda/latest/operatorguide/statelessness-functions.html. Accessed 28.11.2023 (cit. on p. 30).

[18] 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/. Ret. 12.02.2023 (cit. on p. 8).

[19] AWS Lambda. *Security Overview of AWS Lambda*. `https://docs.aws.amazon.com/whitepapers/latest/security-overview-aws-lambda/lambda-executions.html`. Accessed 28.11.2023 (cit. on p. 8).

[20] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013 (cit. on p. 103).

[21] Sol Boucher, Anuj Kalia, David G Andersen, and Michael Kaminsky. "Putting the" micro" back in microservice". In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 2018, pp. 645–650 (cit. on p. 32).

[22] Marc Brooker, Adrian Costin Catangiu, Mike Danilov, et al. "Restoring Uniqueness in MicroVM Snapshots". In: *arXiv preprint arXiv:2102.12892* (2021) (cit. on p. 70).

[23] James Cadden, Thomas Unger, Yara Awad, et al. "SEUSS: skip redundant paths to make serverless fast". In: *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys'20)*. 2020, pp. 1–15 (cit. on p. 66).

[24] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. "Microreboot—A Technique for Cheap Recovery". In: *6th Symposium on Operating Systems Design & Implementation (OSDI'04)*. San Francisco, CA: USENIX Association, Dec. 2004 (cit. on p. 70).

[25] Walter Chang, Brandon Streiff, and Calvin Lin. “Efficient and extensible security enforcement using dynamic data flow analysis”. In: *Proceedings of the 15th ACM conference on Computer and communications security*. 2008, pp. 39–50 (cit. on pp. 73, 102).

[26] Yang Chen. “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 (cit. on p. 65).

[27] Zhifei Chen, Lin Chen, Yuming Zhou, et al. “Dynamic Slicing of Python Programs”. In: *2014 IEEE 38th Annual Computer Software and Applications Conference*. 2014, pp. 219–228 (cit. on p. 80).

[28] James Clause, Wanchun Li, and Alessandro Orso. “Dytan: a generic dynamic taint analysis framework”. In: *Proceedings of the 2007 international symposium on Software testing and analysis*. 2007, pp. 196–206 (cit. on pp. 73, 102).

[29] Cloudflare. *Cloudflare Workers*. `https://workers.cloudflare.com/` (cit. on pp. 2, 67).

[30] Gene Cooperman, Jason Ansel, and Xiaoqin Ma. “Transparent adaptive library-based checkpointing for master-worker style parallelism”. In: *Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID’06)*. Vol. 1. IEEE. 2006, 9–pp (cit. on p. 65).

[31] CRIU. *Checkpoint/Restore In Userspace)*. `https://www.criu.org/` (cit. on pp. 65, 70).

[32] Datadog. *The state of serverless*. https://www.datadoghq.com/state-of-serverless. Ret. on 04.02.2023 (cit. on pp. 1, 9, 73, 78, 102).

[33] Pubali Datta, Prabuddha Kumar, Tristan Morris, et al. "Valve: Securing Function Workflows on Serverless Computing Platforms". In: *Proceedings of The Web Conference 2020*. WWW '20. 2020, 939–950 (cit. on p. 104).

[34] Pubali Datta, Isaac Polinsky, Muhammad Adil Inam, Adam Bates, and William Enck. "ALASTOR: Reconstructing the Provenance of Serverless Intrusions". In: *31st USENIX Security Symposium (USENIX Security 22)*. Aug. 2022 (cit. on p. 104).

[35] Dave Gamble, Max Bruckner, and Alan Wang and contributors. *cJSON)*. https://github.com/DaveGamble/cJSON (cit. on p. 43).

[36] Dorothy E Denning and Peter J Denning. "Certification of programs for secure information flow". In: *Communications of the ACM* 20.7 (1977), pp. 504–513 (cit. on p. 72).

[37] Dong Du, Tianyi Yu, Yubin Xia, et al. "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 (cit. on pp. 8, 32, 62, 64, 66, 70).

[38] Aryaz Eghbali and Michael Pradel. “DynaPyt: a dynamic analysis framework for Python”. In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2022, pp. 760–771 (cit. on p. 75).

[39] Emptymonkey. *A ptrace library designed to simplify syscall injection in Linux*. `https://github.com/emptymonkey/ptrace_do` (cit. on p. 38).

[40] Facebook. *Pysa Tool*. `https://developers.facebook.com/blog/post/2021/04/29/eli5-pysa-security-focused-analysis-tool-python/`. Ret. 13.02.2023 (cit. on pp. 72, 102).

[41] Fastly. *https://www.fastly.com/)*. `https://www.fastly.com/` (cit. on pp. 2, 67).

[42] Github. *CodeQL Tool*. `https://securitylab.github.com/tools/codeql`. Ret. 13.02.2023 (cit. on pp. 72, 102).

[43] Cristiano Giuffrida, Lorenzo Cavallaro, and Andrew S Tanenbaum. “We crashed, now what?” In: *Sixth Workshop on Hot Topics in System Dependability (HotDep 10)*. 2010 (cit. on p. 70).

[44] Patrice Godefroid, Nils Klarlund, and Koushik Sen. “DART: Directed automated random testing”. In: *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. 2005, pp. 213–223 (cit. on p. 103).

[45] Google. *Google Cloud Functions*. `https://cloud.google.com/functions`. Ret. 04.03.2023 (cit. on pp. 2, 6, 16).

[46] Google Cloud Functions. *Tips & Tricks for Cold Start)*. https://cloud.google.com/functions/docs/bestpractices/tips. Accessed 28.11.2023 (cit. on p. 8).

[47] Andreas Haas, Andreas Rossberg, Derek L Schuff, et al. "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 (cit. on p. 58).

[48] JC Huang. "Program instrumentation and software testing". In: *Computer* 11.4 (1978), pp. 25–32 (cit. on pp. 12, 70, 72, 75).

[49] Jung-Chang Huang. "Detection of data flow anomaly through program instrumentation". In: *IEEE Transactions on Software Engineering* 3 (1979), pp. 226–236 (cit. on pp. 12, 70, 72, 75).

[50] Sebastian Hunt and David Sands. "On flow-sensitive security types". In: *ACM SIGPLAN Notices* 41.1 (2006), pp. 79–90 (cit. on p. 72).

[51] Joshua Hursey, Chris January, Mark O'Connor, et al. "Checkpoint/restart-enabled parallel debugging". In: *European MPI Users' Group Meeting*. Springer. 2010, pp. 219–228 (cit. on p. 70).

[52] IBM. *IBM Cloud functions*. https://cloud.ibm.com/functions/. Ret. 04.02.2023 (cit. on p. 6).

[53] 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. Ret. 12.02.2023 (cit. on p. 8).

[54] Abhinav Jangda, Bobby Powers, Emery D Berger, and Arjun Guha. “Not so fast: Analyzing the performance of webassembly vs. native code”. In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 2019, pp. 107–120 (cit. on p. 58).

[55] Deepak Sirone Jegan, Liang Wang, Siddhant Bhagat, Thomas Ristenpart, and Michael M. Swift. “Guarding Serverless Applications with SecLambda”. In: *CoRR* abs/2011.05322 (2020). arXiv: 2011.05322 (cit. on p. 104).

[56] 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/. Ret. 12.02.2023 (cit. on p. 8).

[57] Asim Kadav, Matthew J Renzelmann, and Michael M Swift. “Fine-grained fault tolerance using device checkpoints”. In: *ACM SIGPLAN Notices* 48.4 (2013), pp. 473–484 (cit. on p. 70).

[58] James C King. “Symbolic execution and program testing”. In: *Communications of the ACM* 19.7 (1976), pp. 385–394 (cit. on p. 103).

[59] Samuel T King, George W Dunlap, and Peter M Chen. “Debugging operating systems with time-traveling virtual machines”. In: *Proceedings of the 2005 USENIX Technical Conference*. 2005, pp. 1–15 (cit. on p. 70).

[60] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, et al. “seL4: Formal verification of an OS kernel”. In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP'09)*. 2009, pp. 207–220 (cit. on p. 103).

[61] Jacob Kreindl, Daniele Bonetta, Lukas Stadler, David Leopoldseder, and Hanspeter Mössenböck. “Multi-language dynamic taint analysis in a polyglot virtual machine”. In: *Proceedings of the 17th International Conference on Managed Programming Languages and Runtimes*. 2020, pp. 15–29 (cit. on pp. 75, 102).

[62] Lap Chung Lam and Tzi-cker Chiueh. “A general dynamic information flow tracking framework for security applications”. In: *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*. IEEE. 2006, pp. 463–472 (cit. on pp. 73, 102).

[63] Andrew Lenharth, Vikram S Adve, and Samuel T King. “Recovery domains: an organizing principle for recoverable operating systems”. In: *ACM Sigplan Notices* 44.3 (2009), pp. 49–60 (cit. on p. 70).

[64] Leodanis Pozo Ramos. *https://realpython.com/python-scope-legb-rule/*. `https://realpython.com/python-scope-legb-rule/`. Ret. 12.04.2023 (cit. on p. 83).

[65] Xavier Leroy. “Formal verification of a realistic compiler”. In: *Communications of the ACM* 52.7 (2009), pp. 107–115 (cit. on p. 103).

[66] Wen Li, Ming Jiang, Xiapu Luo, and Haipeng Cai. “PolyCruise: A Cross-Language Dynamic Information Flow Analysis”. In: *31st USENIX Security Symposium (USENIX Security 22)*. Aug. 2022 (cit. on pp. 75, 79–81, 89, 102).

[67] Zijun Li, Quan Chen, and Minyi Guo. “Pagurus: Eliminating Cold Startup in Serverless Computing with Inter-Action Container Sharing”. In: *arXiv preprint arXiv:2108.11240* (2021) (cit. on pp. 32, 62).

[68] Lilian Besson. *Naereen Notebooks: Manual implementation of the Mersenne twister PseudoRandom Number Generator (PRNG)*. `https://github.com/Naereen/notebooks/blob/master/Manual_implementation_of_the_Mersenne_twister_PseudoRandom_Number_Generator__PRNG_.py`. Ret. 28.03.2023 (cit. on pp. 96, 97).

[69] Linux. *Fork system call*. `https://man7.org/linux/man-pages/man2/fork.2.html/` (cit. on p. 9).

[70] Linux. *ptrace – process trace interface*. `https://man7.org/linux/man-pages/man2/ptrace.2.html/` (cit. on pp. 37, 38).

[71] Linux. *SOFT-DIRTY PTEs*. `https://www.kernel.org/doc/Documentation/vm/soft-dirty.txt/` (cit. on pp. 37, 109).

[72] Linux. *Userfaultfd*. `https://www.kernel.org/doc/html/latest/admin-guide/mm/userfaultfd.html` (cit. on pp. 38, 109).

[73] David Lion, Adrian Chiu, Hailong Sun, et al. “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 (cit. on pp. 37, 64).

[74] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, et al. “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 (cit. on p. 64).

[75] James Benjamin Litton. “Composing and Decomposing OS Abstractions”. PhD thesis. University of Maryland, College Park, 2020 (cit. on p. 109).

[76] Louis-Noel Pouchet. *Polybench/C)*. `https://web.cs.ucla.edu/~pouchet/software/polybench/` (cit. on pp. 27, 50).

[77] Uri Lublin, Anthony Liguori, et al. “Kvm live migration”. In: *KVM Forum*. 2007 (cit. on p. 70).

[78] Jonathan Mace and Rodrigo Fonseca. “Universal context propagation for distributed system instrumentation”. In: *Proceedings of the thirteenth EuroSys conference*. 2018, pp. 1–18 (cit. on p. 104).

[79] Kiwan Maeng and Brandon Lucia. “Adaptive dynamic checkpointing for safe efficient intermittent computing”. In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI’18)*. 2018, pp. 129–144 (cit. on p. 70).

[80] Marcin Fatyga, Felix Groebert. *pytaint*. https://github.com/felixgr/pytaint. Ret. 09.04.2023 (cit. on p. 75).

[81] Wes Masri and Andy Podgurski. “Algorithms and Tool Support for Dynamic Information Flow Analysis”. In: *Inf. Softw. Technol.* 51.2 (2009), 385–404 (cit. on p. 75).

[82] Microsoft. *Azure Functions*. https://azure.microsoft.com/en-us/services/functions/. Ret. 04.02.2023 (cit. on pp. 2, 6, 16, 65).

[83] 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/ (cit. on p. 37).

[84] Anup Mohan, Harshad Sane, Kshitij Doshi, et al. “Agile cold starts for scalable serverless”. In: *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*. 2019 (cit. on p. 65).

[85] James Newsome and Dawn Xiaodong Song. “Dynamic Taint Analysis for Automatic Detection, Analysis, and SignatureGeneration of Exploits on Commodity Software.” In: *NDSS*. Vol. 5. Citeseer. 2005, pp. 3–4 (cit. on pp. 73, 102).

[86] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. “Data Flow Analysis”. In: *Principles of Program Analysis*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 35–139 (cit. on p. 76).

[87] Tobias Nipkow, Markus Wenzel, and Lawrence C Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer, 2002 (cit. on p. 103).

[88] Edward Oakes, Leon Yang, Dennis Zhou, et al. “SOCK: Rapid task provisioning with serverless-optimized containers”. In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 2018, pp. 57–70 (cit. on pp. 8, 65).

[89] OpenFaaS. *OpenFaaS*. `https://www.openfaas.com/`. Ret. 05.02.2023 (cit. on pp. 2, 16).

[90] OpenWhisk. *OpenWhisk commit*. `https://github.com/apache/openwhisk/commit/ed3f76e38d89468d11e862ee0539e74f02ac7f8e` (cit. on p. 42).

[91] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. “The design and implementation of Zap: A system for migrating computing environments”. In: *ACM SIGOPS Operating Systems Review* 36.SI (2002), pp. 361–376 (cit. on p. 70).

[92] OWASP. *OWASP Serverless Top 10*. `https://owasp.org/www-project-serverless-top-10/`. Ret. 12.02.2023 (cit. on p. 8).

[93] Isaac Polinsky, Pubali Datta, Adam Bates, and William Enck. “SCIFFS: Enabling Secure Third-Party Security Analytics Using Serverless Computing”. In: *Proceedings of the 26th ACM Symposium on Access Control Models and Technologies*. SACMAT ’21. 2021, 175–186 (cit. on p. 104).

[94] Georgios Portokalidis and Angelos D Keromytis. “REASSURE: A self-contained mechanism for healing software using rescue points”. In: *International Workshop on Security*. Springer. 2011, pp. 16–32 (cit. on p. 70).

[95] Python. *Python: Execution model*. https://docs.python.org/3/reference/executionmodel.html/. Ret. 12.04.2023 (cit. on p. 83).

[96] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. "Rx: Treating Bugs as Allergies—a Safe Method to Survive Software Failures". In: *20th ACM Symposium on Operating Systems Principles (SOSP '05)*. 2005, 235–248 (cit. on p. 70).

[97] Rich Jones. *Gone in 60 Milliseconds: Intrusion and Exfiltration in Serverless Architectures*. https://media.ccc.de/v/33c3-7865-gone_in_60_milliseconds. Ret. 12.02.2023 (cit. on p. 8).

[98] Michael Rieker, Jason Ansel, and Gene Cooperman. "Transparent User-Level Checkpointing for the Native Posix Thread Library for Linux." In: *PDPTA*. Vol. 6. 2006, pp. 492–498 (cit. on p. 65).

[99] Alejandro Russo and Andrei Sabelfeld. "Dynamic vs. Static Flow-Sensitive Security Analysis". In: *2010 23rd IEEE Computer Security Foundations Symposium*. 2010, pp. 186–199 (cit. on p. 72).

[100] Andrei Sabelfeld and Andrew C Myers. "Language-based information-flow security". In: *IEEE Journal on selected areas in communications* 21.1 (2003), pp. 5–19 (cit. on p. 72).

[101] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. "All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask)". In: *2010 IEEE Symposium on Security and Privacy*. 2010, pp. 317–331 (cit. on pp. 72, 73, 75, 102).

[102] Koushik Sen and Gul Agha. “CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools”. In: *International Conference on Computer Aided Verification*. 2006, pp. 419–423 (cit. on p. 103).

[103] Koushik Sen, Darko Marinov, and Gul Agha. “CUTE: A concolic unit testing engine for C”. In: *ACM SIGSOFT Software Engineering Notes* 30.5 (2005), pp. 263–272 (cit. on p. 103).

[104] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. “Architectural Implications of Function-as-a-Service Computing”. In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO ’52. 2019, 1063–1075 (cit. on pp. 27, 50).

[105] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, et al. “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 (cit. on pp. 8, 16).

[106] Simon Shillaker and Peter Pietzuch. “Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing”. In: *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. July 2020, pp. 419–433 (cit. on pp. 2, 56, 58, 67, 70).

[107] Stelios Sidiroglou, Oren Laadan, Carlos Perez, et al. “Assure: automatic software self-healing using rescue points”. In: *ACM SIGARCH Computer Architecture News* 37.1 (2009), pp. 37–48 (cit. on p. 70).

[108] Paulo Silva, Daniel Fireman, and Thiago Emmanuel Pereira. “Prebaking functions to warm the serverless cold start”. In: *Proceedings of the 21st International Middleware Conference*. 2020, pp. 1–13 (cit. on pp. 32, 62).

[109] Johannes Späth, Karim Ali, and Eric Bodden. “Context-, flow-, and field-sensitive data-flow analysis using synchronized pushdown systems”. In: *Proceedings of the ACM on Programming Languages* 3.POPL (2019), pp. 1–29 (cit. on p. 76).

[110] Sudarshan M. Srinivasan, Srikanth Kandula, Christopher R. Andrews, and Yuanyuan Zhou. “Flashback: A Lightweight Extension for Rollback and Deterministic Replay for Software Debugging”. In: *2004 USENIX Annual Technical Conference (USENIX ATC 04)*. 2004, p. 3 (cit. on p. 70).

[111] Oliver Stenbom. “Refunction: Eliminating Serverless Cold Starts Through Container Reuse”. MA thesis. Imperial College London, 2019 (cit. on pp. 8, 32, 38, 65).

[112] Dinesh Subhraveti and Jason Nieh. “Record and transplay: partial checkpointing for replay debugging across heterogeneous systems”. In: *Proceedings of the ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*. 2011, pp. 109–120 (cit. on p. 70).

[113] G Edward Suh, Jae W Lee, David Zhang, and Srinivas Devadas. “Secure program execution via dynamic information flow tracking”. In: *ACM Sigplan Notices* 39.11 (2004), pp. 85–96 (cit. on pp. 73, 102).

[114] Synopsys. *CVE-2014-0160: The Heartbleed Bug*. https://heartbleed.com/. Ret. 12.02.2023 (cit. on p. 8).

[115] Jörg Thalheim, Pramod Bhatotia, Pedro Fonseca, and Baris Kasikci. "Cntr: Lightweight OS Containers". In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 2018, pp. 199–212 (cit. on p. 65).

[116] Joseph Tucek, Shan Lu, Chengdu Huang, Spiros Xanthos, and Yuanyuan Zhou. "Triage: diagnosing production run failures at the user's site". In: *ACM SIGOPS Operating Systems Review* 41.6 (2007), pp. 131–144 (cit. on p. 70).

[117] 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 (cit. on p. 67).

[118] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. "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 (cit. on pp. 8, 31, 32, 62, 66, 70).

[119] Joel Van Der Woude and Matthew Hicks. "Intermittent computation without hardware support or programmer intervention". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. 2016, pp. 17–32 (cit. on p. 70).

[120] Manav Vasavada, Frank Mueller, Paul H Hargrove, and Eric Roman. “Comparing different approaches for incremental checkpointing: The showdown”. In: *Linux Symposium*. Vol. 69. 2011 (cit. on p. 65).

[121] Ranjan Sarpangala Venkatesh, Till Smejkal, Dejan S Milojicic, and Ada Gavrilovska. “Fast in-memory CRIU for docker containers”. In: *Proceedings of the International Symposium on Memory Systems*. 2019, pp. 53–65 (cit. on p. 66).

[122] Victor Stinner. *The Python Performance Benchmark Suite)*. `https://pyperformance.readthedocs.io` (cit. on pp. 27, 49).

[123] Dirk Vogt, Cristiano Giuffrida, Herbert Bos, and Andrew S Tanenbaum. “Lightweight memory checkpointing”. In: *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE. 2015, pp. 474–484 (cit. on pp. 32, 70).

[124] Kai-Ting Amy Wang, Rayson Ho, and Peng Wu. “Replayable execution optimized for page sharing for a managed runtime environment”. In: *Proceedings of the Fourteenth EuroSys Conference 2019*. 2019, pp. 1–16 (cit. on pp. 8, 62, 65).

[125] Tao Wang and Abhik Roychoudhury. “Dynamic Slicing on Java Bytecode Traces”. In: *ACM Trans. Program. Lang. Syst.* 30.2 (2008) (cit. on p. 80).

[126] 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`. Ret. 12.02.2023 (cit. on p. 8).

[127] WebAssembly. *WebAssembly)*. https://webassembly.org/ (cit. on pp. 32, 67).

[128] Wesley Beary. *CVE-2019-16779: RubyGem excon leak previous response*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-16779. Ret. 12.02.2023 (cit. on p. 8).

[129] Zhaogui Xu, Peng Liu, Xiangyu Zhang, and Baowen Xu. "Python predictive analysis for bug detection". In: *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*. 2016, pp. 121–132 (cit. on pp. 75, 79, 80, 89).

[130] Angeliki Zavou, Georgios Portokalidis, and Angelos D Keromytis. "Self-healing multitier architectures using cascading rescue points". In: *Proceedings of the 28th Annual Computer Security Applications Conference*. 2012, pp. 379–388 (cit. on p. 70).