Towards Effective Deployment of Unikernels

Gaulthier Gain

Faculty of Applied Sciences
Department of Electrical Engineering and Computer Science

Liège, July 2026

This page intentionally left blank.

PhD dissertation

Towards Effective Deployment of Unikernels

Gaulthier Gain

Supervisor: *Prof. Laurent Mathy*

PhD thesis submitted in fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science

Faculty of Applied Sciences
Department of Electrical Engineering and Computer Science
University of Liège

Liège, July 2026

This page intentionally left blank.

Towards Effective Deployment of Unikernels

This page intentionally left blank.

JURY MEMBERS

Prof. Guy Leduc, Université de Liège (President);
Prof. Laurent Mathy, Université de Liège (Advisor);
Prof. Bernard Boigelot, Université de Liège;
Prof. Jon Crowcroft, University of Cambridge;
Dr. Tom Goethals, Universiteit Gent;
Prof. Timo Hönig, Ruhr-Universität Bochum;
Dr. Felipe Huici, Unikraft GmbH;

This page intentionally left blank.

ACKNOWLEDGEMENTS

Like any complex operating system, this thesis took time to mature and evolved through iteration and adaptation. Papers were rejected, initial designs were refined, and understanding deepened through repeated cycles of implementation and revision. This work unfolded in parallel with teaching responsibilities, European research projects, and academic deadlines, each with its own timeline and challenges. Balancing research, courses, students, and collaborative projects often felt like running multiple critical processes without a scheduler generous enough to keep everything responsive.

Fortunately, I did not face these challenges alone. Beyond the difficulties, this thesis became a continuous learning process, shaped by encounters that were enriching both professionally and personally, and ultimately a deeply rewarding experience. The development of this work would not have been possible without the support of many people. I would therefore like to thank all those who contributed, from near and far, to the completion of my thesis.

First and foremost, I would like to express my deepest gratitude to my PhD supervisor, Laurent Mathy. Thank you for placing your trust in me and for guiding my development over all these years, from the operating systems course, through my master's thesis, and ultimately to this doctoral work, during which I learned a great deal. You consistently challenged my assumptions, often with a rigor that left little room for comfort, but always with the aim of strengthening the technical foundations of this research and encouraging me to step beyond my comfort zone. Our many discussions, often accompanied by cups of coffee, provided a space where ideas could be questioned, refined, and sometimes dismantled, before eventually finding their way into this thesis. This thesis would not exist without your excellent advice. For all of this, I am deeply grateful.

I would also like to sincerely thank my colleagues Cyril Soldani and Benoît Knott, with whom I had numerous insightful discussions, collaborations, and exchanges of valuable advice. I greatly enjoyed working and collaborating with you. I am also deeply grateful to all the collaborators who helped to explore and deepen my understanding of systems research, in particular Felipe Huici, Pierre Olivier, Benoit Donnet, and Costin Raiciu. I feel fortunate to have had the opportunity to work with such talented researchers and mentors. I would like to extend special thanks to my co-authors and friends Hugo Lefeuvre, Vlad-Andrei Bădoiu, and

Alexander Jung, with whom I had the pleasure of collaborating on several papers and sharing enriching research experiences. I am also thankful to the members of the Unikraft and UNICORE communities, particularly Simon Kuenzer and Răzvan Deaconescu, for their support, discussions, and collaboration.

I am grateful to my closest colleagues and friends, Emilien Wansart, Maxime Goffart, Benoît Mattheus, Florian Dekinder, Justin Iurman, and Elias Piette, for the many good times and engaging conversations we shared. I would also like to thank the many colleagues with whom I had the pleasure of working: Kenichi Yasukata, Emeline Maréchal, Jonathan Francart, Antoine Dubois, Jonathan Dumas, Anaïs Halin, Géraldine Brieven, Sami Ben Mariem, Vincent Rossetto, François Straet, Maxime Aerts, Vincent Jacquot, Loïc Champagne, Lev Malcev, Matthias Pirlet, Clément Moureau, and François Piron. I am further grateful to Bardhyl Miftari, my best student, as well as Jocelyn Mbenoun, Laurie Boveroux, and Guillaume Derval. I would also like to thank the students who worked with me during this thesis: Guillaume Lodrini, Terry Loslever, Léo Huteau, and Maxime Letemple. I am equally grateful to my colleagues from mathematics and chemistry: Safia Bennabi, Manon Stipulanti, Pierre Mathonet, Naïm Zenaïdi, Lucas Michel, Patricia Tossings, Marie-Noël Dumont, and Grégoire Leonard, for the enriching interdisciplinary discussions we shared.

Outside of work, I am thankful to the "Sunday sports enthusiasts" team, my friends with whom I shared running, workouts, and other activities: Julien Simar, Julien Gustin, Justin Smagghe, Melvin Gabrys, Hugo Di Giovanni, and Jeffrey De Santis. I am also grateful to Julien Ditroia, Romain Mathonet and Laura Fraineux.

I would also like to thank the members of the jury for agreeing to devote their time to evaluating my research work.

Finally, I would like to express my deepest gratitude to my family. Thank you, for everything you have given me, and especially for your constant encouragement, and unwavering support. I would also like to extend a special thank you to my sister, Gwen, who inspired me to pursue this PhD; I will always be grateful for that. I would also like to thank my in-laws for their encouragement, kindness, and support.

Last but not least, I would like to thank my partner, Aurore. Your unwavering support and encouragement have helped me navigate numerous challenges and motivated me to pursue my goals. I am deeply grateful to have you in my life. I love you.

Gaulthier

Funding

This work was partly funded by the European Union's Horizon 2020 program under grant agreement 825377 (UNICORE), and by the CyberExcellence project, funded by the Walloon Region under grant number 2110186.

"All problems in computer science can be solved by another level of indirection."

— David J. Wheeler

This page intentionally left blank.

RÉSUMÉ

Avec l'essor du *cloud computing*, la demande de déploiements logiciels performants, sécurisés et évolutifs ne cesse de croître. Les unikernels, des machines virtuelles hautement spécialisées dédiées à une seule application, offrent des avantages notables en termes de performance, de sécurité et d'efficacité des ressources. Cependant, leur adoption demeure limitée, notamment à cause de la complexité du portage des applications existantes, une difficulté bien identifiée liée au manque d'outils adaptés et à la compatibilité restreinte avec les interfaces POSIX. Cette spécialisation entraîne également une inefficacité du partage mémoire, réduisant la déduplication entre instances construites à partir de bases logicielles similaires.

Cette thèse analyse systématiquement ces limitations et propose des solutions concrètes pour améliorer l'adaptabilité à grande échelle et la portabilité des applications déployées sous forme d'unikernels.

Premièrement, nous abordons le problème d'inefficacité mémoire induite par la spécialisation, qui entrave le partage efficace des pages mémoire entre différents unikernels. Nous proposons une technique d'alignement de la disposition mémoire, augmentant la similarité des pages et permettant une déduplication plus efficace ainsi que des économies de mémoire significatives. Cette approche permet de supporter davantage d'instances simultanément, sans compromettre ni l'isolation ni les performances. Nous évaluons également les limites des mécanismes classiques de déduplication et explorons des alternatives plus adaptées. Nous analysons également l'impact de l'exécution simultanée de plusieurs unikernels utilisant différentes versions d'une même bibliothèque, et proposons une approche combinant alignement et analyse différentielle inter-versions.

Ensuite, nous abordons le défi du portage d'applications vers des unikernels, en analysant les dépendances et appels système requis par les applications existantes. Nous développons des outils capables d'identifier ces prérequis et de simplifier le portage.

Nos travaux s'appuient sur Unikraft, un *framework* modulaire pour le développement d'unikernels, constituant la base de la mise en oeuvre et de l'évaluation de nos contributions. En améliorant l'efficacité mémoire et en réduisant la barrière d'entrée au portage applicatif, cette thèse positionne les unikernels comme un modèle de déploiement viable dans des environnements modernes et contraints en ressources.

This page intentionally left blank.

ABSTRACT

In today's cloud-native computing ecosystem, the demand for efficient, secure, and scalable software deployment has intensified. Unikernels – highly specialized, single-purpose virtual machines – offer significant advantages in performance, security, and resource efficiency by compiling applications together with only the necessary system components. However, their adoption remains limited, notably due to the complexity of porting existing applications, a well-known challenge stemming from the lack of appropriate tooling and limited compatibility with Portable Operating System Interface (POSIX) interfaces. Beyond these portability challenges, a less explored limitation is the memory-sharing inefficiency caused by unikernel specialization, which prevents memory deduplication across instances that nonetheless share a largely similar software base.

This thesis systematically investigates these limitations and proposes practical solutions to enhance the scalability and usability of unikernel systems. First, we address the memory inefficiency caused by unikernel specialization, which prevents effective sharing of memory pages across instances. We propose a memory layout alignment technique that increases page sharing. This technique improves memory deduplication, yielding significant memory savings and allowing more instances to run concurrently while maintaining performance. We also evaluate the limitations of traditional deduplication mechanisms like Kernel Samepage Merging (KSM) in unikernel environments and explore tailored alternatives. Furthermore, we explore the impact of running multiple unikernels that include different versions of the same libraries and introduce a novel framework that combines alignment strategies with differential analysis across library versions. Second, we tackle the challenge of application portability by analyzing system call dependencies and OS abstractions required by legacy applications. We develop tooling that identifies these dependencies and simplifies the porting process, reducing manual effort and broadening compatibility with POSIX-compliant applications.

Our work is grounded in Unikraft, a modular unikernel development framework, which serves as the foundation for implementing and validating our contributions. By improving memory efficiency and lowering the entry barrier for application porting, this thesis contributes to making unikernels more practical for deployment in modern, multi-tenant, and resource-constrained environments.

This page intentionally left blank.

CONTENTS

This page intentionally left blank.

LIST OF FIGURES

This page intentionally left blank.

LIST OF TABLES

This page intentionally left blank.

ACRONYMS

ABI Application Binary Interface. (*p. 123*)
AI Artificial Intelligence. (*p. 141*)
API Application Programming Interface. (*p. 9*)
ASLR Address Space Layout Randomization. (*p. 22*)
AWS Amazon Web Services. (*p. 74*)

CI/CD Continuous Integration/Continuous Delivery. (*p. 57*)
CoW Copy-on-Write. (*p. 17*)
CXL Compute Express Link. (*p. 78*)

DCE Dead Code Elimination. (*p. 22*)
DKSM Daemonless Kernel Samepage Merging. (*p. 74*)

ELF Executable and Linkable Format. (*p. 3*)

FaaS Function as a Service. (*p. 21*)

GOT Global Offset Table. (*p. 14*)

IoT Internet of Things. (*p. 141*)

KPTI Kernel Page Table Isolation. (*p. 123*)
KSM Kernel Samepage Merging. (*p. vii*)
KVM Kernel-based Virtual Machine. (*p. 16*)

LLM Large Language Model. (*p. 133*)

ML Machine Learning. (*p. 139*)
MMIO Memory-mapped I/O. (*p. 37*)
MPK Memory Protection Keys. (*p. 77*)

NVMEM Non Volatile Memory. (*p. 78*)

OS Operating System. (*p. 1*)

PCRE Perl Compatible Regular Expressions. (*p. 96*)
PE Portable Executable. (*p. 75*)
PIC Position Independent Code. (*p. 27*)
PLT Procedure Linkage Table. (*p. 14*)
POSIX Portable Operating System Interface. (*p. vii*)

QEMU Quick EMUlator. (*p. 60*)

SaaS Software as a Service. (*p. 21*)

TCB Trusted Computing Base. (*p. 77*)
TLB Translation Lookaside Buffer. (*p. 45*)
TPS Transparent Page Sharing. (*p. 16*)

UKSM Ultra Kernel Samepage Merging. (*p. 18*)

VM Virtual Machine. (*p. 1*)
VMM Virtual Machine Manager. (*p. 15*)

WASM WebAssembly. (*p. 1*)

XLH Cross Layer I/O-based Hints. (*p. 18*)

This page intentionally left blank.

1 INTRODUCTION

Public and private cloud infrastructures underpin modern computing, enabling the deployment of scalable, continuously evolving services. At their core lies a fundamental tension: cloud platforms must simultaneously provide strong isolation between tenants while maximizing resource efficiency across large-scale deployments. These goals are inherently at odds. Mechanisms that strengthen isolation tend to fragment resources, while those that improve efficiency (for instance by sharing resources) often weaken isolation [3, 13, 28, 145].

Early cloud infrastructures relied on the Virtual Machine (VM) paradigm as the primary deployment model, leveraging hardware-assisted virtualization to ensure strong isolation [166, 233, 195, 218]. However, VMs are heavyweight: each requires a full Operating System (OS) image, consuming significant resources (e.g., memory, disk). This results in inefficiencies such as duplicated OS functionality, long boot times (often tens of seconds), and limited scalability. To overcome these issues, the industry has increasingly adopted containers.

Containers emerged as a more lightweight alternative, enabling near-native performance by sharing the host kernel across isolated user-space processes [62, 48, 135]. While containers improve efficiency, they weaken isolation by exposing a large, shared system call interface, which remains a frequent source of vulnerabilities [147, 187, 75, 191, 98, 95]. In practice, modern cloud platforms [82, 7] combine both approaches by deploying containers inside VMs, partially restoring isolation but reintroducing redundancy, leaving the fundamental efficiency problem unresolved.

These limitations have driven research and industry toward a broader spectrum of isolation primitives. More recent approaches, including microVMs [2], user-space kernel-based isolation mechanisms (such as gVisor [83]), and language-based sandboxes such as WebAssembly (WASM) [88], explore different trade-offs between isolation, performance, and compatibility. However, these systems largely retain or emulate general-purpose operating system abstractions, preserving a broad system call interface that contributes to both complexity and an enlarged attack sur-

face. As a result, they do not fundamentally resolve the tension between isolation and efficient resource utilization.

This observation motivates a more radical approach: rather than relying on a general-purpose OS, specialize it to the application. Built on the concept of library OSes [8, 60, 119, 156, 237], unikernels are specialized and lightweight OS images that include only the minimal components required to run a single application [128, 111, 97, 127, 105]. This design significantly reduces memory footprint, boot time, and attack surface, while preserving strong isolation through virtualization [155, 229]. Several mature implementations, including MirageOS [127], ClickOS [132], HermiTux [148], and IncludeOS [34], demonstrate the viability of this approach across diverse application domains and deployment contexts.

Despite their advantages, unikernels also come with certain trade-offs. First, their high degree of specialization – while beneficial for performance and security – inherently limits opportunities for memory deduplication [11, 138, 230, 27]. This mechanism, commonly employed by hypervisors and OSes, identifies identical memory pages across multiple virtual machines or processes and maps identical pages to a shared physical copy to reduce overall memory usage. While major public cloud providers may choose to disable cross-tenant memory deduplication in response to demonstrated side-channel attacks [189, 232, 32], such mechanisms remain available in private cloud and on-premises environments [104, 157], making memory efficiency a relevant concern for multi-tenant unikernel deployments. Because each unikernel is compiled and linked with only the exact libraries and system components it requires, it produces a compact, specialized binary. As a result, this significantly reduces overlap in memory pages between different unikernels, even when they share common code/libraries. This lack of sharing in unikernels can result in higher overall memory consumption than expected when deploying large numbers of similar services, which becomes a critical concern in multi-tenant environments [145]. This specialization also complicates library versioning. Indeed, even small changes (such as added, modified, or removed functions) in library versions can lead to divergent binary layouts, further reducing the chance of memory page similarity between unikernel instances. This version-specific compilation not only hampers deduplication but also increases storage requirements across deployments.

Another fundamental drawback of the unikernel paradigm is the need to manually port existing applications to the unikernel environment [148, 97]. For instance, converting a web server into a unikernel requires carefully selecting and extracting the necessary OS components (e.g., network stack, drivers) while conforming to a reduced system interface. Porting applications is challenging due to a variety of factors, including incompatible or missing libraries and features, proprietary code, complex codebase and build systems, limited developer tooling, and unimplemented or unsupported system calls. A specific difficulty arises from identifying which system calls an application truly requires. Overestimating this set can in-

troduce unnecessary engineering effort, forcing developers to include and manage features that the application does not actually need. Conversely, underestimating or overlooking required system calls can result in missing features, ultimately leading to runtime errors or application crashes. These tedious and error-prone manual steps make unikernel adoption resource-intensive and deter widespread use in the software industry.

In this thesis, we systematically analyze and address two limitations of current unikernel systems: (1) the lack of sharing opportunities that restrict the effectiveness of memory deduplication and thus density in cloud environments, and (2) the complexity of porting existing applications due to limited compatibility and tooling support. Through a detailed investigation of these challenges, we aim to propose practical solutions that make unikernels more scalable, resource-efficient, and accessible to developers.

To address the first challenge, we propose an alignment-based methodology for constructing memory-optimized unikernels. This technique strategically reorganizes the memory layout to maximize the number of identical pages, thereby enabling more effective sharing between instances and significantly improving overall memory efficiency – without introducing performance overhead or weakening isolation guarantees. Additionally, we examine the limitations of conventional deduplication mechanisms such as KSM [11], and explore alternative techniques specifically tailored to the unikernel execution model. These techniques aim to improve system performance while avoiding the drawbacks associated with memory deduplication scanners [11, 139, 138, 230], paving the way for its broader adoption in hyperscaler environments where cross-tenant deduplication has been largely disabled due to security concerns. Furthermore, we explore the impact of running multiple unikernels that include different versions of the same libraries and introduce a novel framework that combines alignment strategies with differential analysis across library versions to increase memory sharing.

To address the second limitation, we investigate system call analysis and dependency management, proposing methods to infer and extract the set of required OS abstractions from existing applications. This includes static and dynamic analysis of Executable and Linkable Format (ELF) binaries and object files to identify system-level dependencies. To this end, we develop and evaluate tooling that reduces manual intervention, lowers engineering overhead, and improves compatibility with widely-used POSIX applications.

To ground our work in a practical setting, we rely on Unikraft [105], an open-source unikernel development kit designed for modularity and specialization, and the central platform of the UNICORE Horizon 2020 project. Unikraft enables developers to assemble unikernels from fine-grained components, resulting in high-performance systems with minimal footprint that can be tailored to specific applications. Its modular architecture makes it a suitable foundation for exploring both memory deduplication and application portability. Most of our tools are available

on GitHub and are integrated into the Unikraft repository.

By tackling these challenges, this thesis contributes to making unikernels more practical for large-scale deployment. While application portability remains a primary barrier to adoption, improving resource sharing is essential to ensuring their efficiency in modern cloud-native platforms. Our goal is to address key limitations of unikernels – memory inefficiency and porting complexity – to bring them closer to the requirements of real-world systems.

1.1 Outline

This manuscript is organized into four parts. Part I provides the necessary background, offering key insights essential for understanding our contributions. Part II presents our work on unikernels and memory deduplication. Part III focuses on system call analysis and application porting to unikernels. Finally, Part IV summarizes our findings and outlines promising directions for future work.

Part I: Foundations

- Chapter 2 provides an overview of Unikraft (an open-source Unikernel Development Kit), ELF files, hypervisor, and memory deduplication, offering the necessary background for understanding this thesis.

Part II: Unikernels & Memory Deduplication

- Chapter 3 examines the memory impact of running multiple unikernels on the same server and introduces Spacer, a solution that aligns libraries to improve memory deduplication.
- Chapter 4 proposes a novel approach to improve memory efficiency and performance by merging identical code and read-only data of unikernels at load time, rather than relying on runtime memory scanners.
- Chapter 5 explores the implications of running multiple library versions of the same unikernel concurrently, highlighting both challenges and opportunities for memory deduplication, and proposes a method to minimize overall memory usage.

Part III: System Call Analysis & Unikernel Porting

- Chapter 6 focuses on application porting to Unikraft, presenting the design and implementation of analysis tools that infer system call usage and assist in configuring applications for unikernel environments.

Part IV: Conclusion & Future Work

- Chapter 7 concludes the thesis by summarizing our findings on unikernels, memory deduplication, system call analysis, and application porting to unikernels, and outlines directions for future research.

1.2 Selected Scientific Publications

This manuscript is based on existing research:

1. Chapter 3 is an adapted version of the publication: **Gaulthier Gain**, Cyril Soldani, Felipe Huici, and Laurent Mathy. *Want more unikernels? Inflate them!*. In Proceedings of the 13th Symposium on Cloud Computing (SoCC '22). Association for Computing Machinery, New York, NY, USA, 510–525.
2. Chapter 4 is an adapted version of the publication: **Gaulthier Gain**, Benoît Knott, Cyril Soldani, and Laurent Mathy. *Memory Matters: Load-Time Deduplication for Unikernels*. In Proceedings of the 2025 ACM Symposium on Cloud Computing (SoCC '25). Association for Computing Machinery, New York, NY, USA, 464–478. (*Best Paper Honorable Mention*).
3. Chapter 5 is an adapted version of the publication: **Gaulthier Gain**, Benoît Knott, and Laurent Mathy. *Efficient versioning for Unikernels*. In 2025 IEEE 18th International Conference on Cloud Computing (CLOUD), Helsinki, Finland, 2025, pp. 296-307.
4. Chapter 6 is based on the methodology and tools for application porting presented in the publication: Simon Kuenzer*, Vlad-Andrei Bădoiu*, Hugo Lefeuvre*, Sharan Santhanam*, Alexander Jung*, **Gaulthier Gain***, Cyril Soldani*, Costin Lupu, Ştefan Teodorescu, Costi Răducanu, Cristian Banu, Laurent Mathy, Răzvan Deaconescu, Costin Raiciu, and Felipe Huici. 2021. *Unikraft: fast, specialized unikernels the easy way*. In Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys '21). Association for Computing Machinery, New York, NY, USA, 376–394. (*Best Paper Award*).

1.3 Other Scientific Publications

Additional scientific publications arising from this research are listed below.

1. **Gaulthier Gain**, Cyril Soldani, and Laurent Mathy. *UNICORE: A toolkit to automatically build unikernels*. Grascomp Doctoral Day 2019.
2. Maxime Letemple, **Gaulthier Gain**, Sami Ben Mariem, Laurent Mathy, and Benoit Donnet. *vTNT: Unikernels for Efficient and Flexible Internet Probing*, 8th Network Traffic Measurement and Analysis Conference (TMA), Dresden, Germany, 2024, pp. 1-4.
3. Hugo Lefeuvre, **Gaulthier Gain**, Daniel Dinca, Alexander Jung, Simon Kuenzer, Vlad-Andrei Badoiu, Razvan Deaconescu, Laurent Mathy, Costin Raiciu, Pierre Olivier, and Felipe Huici. *Unikraft and the coming of age of unikernels*. USENIX; login, 2021.
4. Hugo Lefeuvre, **Gaulthier Gain**, Vlad-Andrei Bădoiu, Daniel Dinca, Vlad-Radu Schiller, Costin Raiciu, Felipe Huici, and Pierre Olivier. 2024. *Loupe:*

* These are main authors that contributed equally to the paper.

Driving the Development of OS Compatibility Layers. In Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '24), Vol. 1. Association for Computing Machinery, New York, NY, USA, 249–267.

1.4 Code and Artifacts

The code and artifacts for our publications are also publicly available online:

- Chapter 3: https://github.com/gaulthiergain/spacer
- Chapter 4: https://github.com/gaulthiergain/spacer-slt
- Chapter 5: https://github.com/gaulthiergain/spacer-delta
- Chapter 6: https://github.com/unikraft/eurosys21-artifacts

Part I

Foundations

This page intentionally left blank.

2
BACKGROUND

Overview

The objective of this chapter is to provide the necessary background on Unikraft, ELF files, hypervisor and memory deduplication, offering key insights to understand our contributions. It is not intended to be an exhaustive exploration, but rather an overview of the topics most relevant to this thesis. For readers interested in further deepening their understanding, we provide references to relevant literature.

2.1 Unikraft

Traditional operating system architectures typically fall into two categories. Monolithic kernels tightly integrate core components – such as device drivers, file systems, and memory management – into a single binary [94, 210]. In contrast, microkernels emphasize strong isolation between system components [89, 57].

Unikraft [105] introduces a modern alternative. It is an open-source codebase and development kit for building unikernels – specialized, single-purpose OS instances in which an application is compiled together with only the OS functionality it requires into a single binary. In this *one application per unikernel model*, traditional protection domains between user and kernel space become unnecessary, reducing context-switch overhead and enabling aggressive specialization.

Unikraft combines the performance benefits of monolithic kernels with the modularity and configurability typically found in microkernel architectures. Unikraft's key innovation lies in its fine-grained modularity, allowing developers to assemble highly specialized OS instances by selecting only the components required for their applications. These components interact strictly through well-defined Application Programming Interface (API) boundaries, maintaining isolation and clarity. Rather than compromising modularity for performance, Unikraft achieves high efficiency

through carefully designed APIs and static linking of components. Figure 2.1 illustrates how a traditional application stack can be transformed into a specialized Unikraft-based unikernel.

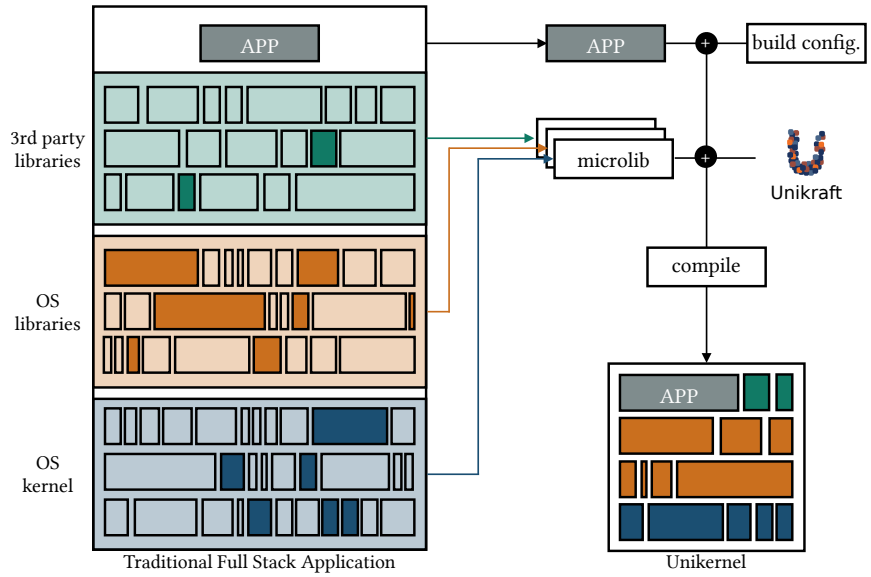


Figure 2.1: *From a traditional full stack application to a specialized Unikraft-based unikernel. The application to be ported is shipped with the essential libraries and OS components into a unikernel.*

To achieve the overarching principle of modularity, Unikraft consists of two main components:

1. Micro-libraries and pools: Micro-libraries (also called micro-libs or libraries) are modular software components that either implement specific Unikraft core subsystems (such as allocators or schedulers) or wrap external functionality (such as libc or network stacks) to make them compatible with the Unikraft ecosystem. Within a given pool, all libraries adhere to the same API standard, making them fully interchangeable. They can integrate functionality from external library projects such as OpenSSL [149], musl [61], Protobuf [81], and more. They can also embed applications like SQLite [6], Nginx [163], or even support different platforms such as Solo5 [182] or Firecracker [2] and CPU architectures (e.g., x86, ARM, etc.). The size of micro-libraries can vary widely, ranging from extremely compact ones, such as those containing basic boot code (e.g., ukboot [200]), to substantial ones, like those providing comprehensive libc (e.g., newlib [162]) or network support (e.g., lwip [170]). Unikraft maintains an official set of micro-libraries on GitHub [201], and users can create their own.
2. Build system: Unikraft provides a user-friendly menu based on Kconfig[1], allowing users to choose the micro-libraries they want to incorporate into their system during the build process. Users can also specify the platform(s) and

[1] Kconfig is a configuration system providing a hierarchical menu-driven interface for enabling/disabling features.

CPU architecture(s) to target and configure individual micro-libs if desired (i.e., memory allocators, network protocol support, etc). The build system efficiently handles the compilation and linking of all chosen micro-libs, resulting in the creation of one binary per selected platform and architecture. Unlike conventional applications that rely on dynamically linked shared libraries, Unikraft embeds all required code and libraries directly into the executable at build time, producing a self-contained ELF binary.

2.2 Executable and Linkable Format (ELF)

Unikernels are typically stored as binaries [105, 34, 97] in ELF [207], a widely used format for executable files in UNIX-like systems. The ELF format has a consistent structure across different architectures, though certain elements, such as endianness and symbol types, may vary depending on the platform. Additionally, the content of the ELF file, particularly the code, is architecture-specific. ELF files are processed by two different tools: the linker and the loader, each interacting with the file from different perspectives.

From the linker's perspective, the ELF format divides the binary into distinct sections, such as `.text`, `.data`, `.bss`, and `.rodata`, each serving a specific role – storing executable code, initialized and uninitialized writable data, and read-only data. The linker's primary task is to combine multiple object files (ELF format) into a single executable by resolving symbols and addresses and arranging sections to create an executable binary. Additionally, ELF files contain metadata, such as the ELF header and program header table, which provide crucial information about the binary's structure and layout.

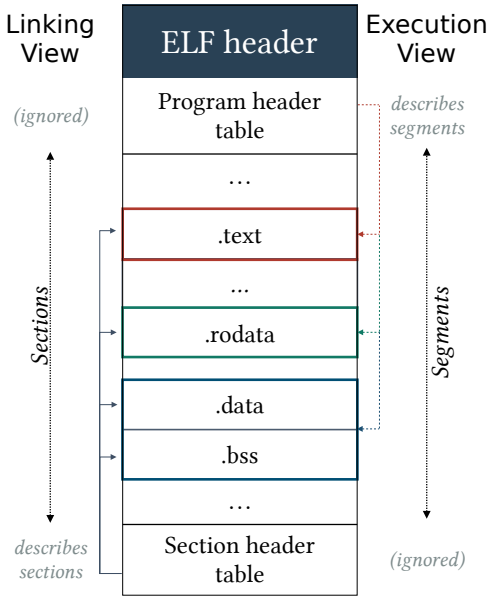


Figure 2.2: *Two views of an ELF file.*

From the loader's perspective, the ELF file serves as an instruction set for load-

ing the executable into memory. The loader interprets the program header table to map the segments defined in the ELF file to the appropriate memory regions. The loader is responsible for managing memory allocation, applying any necessary relocations, and setting up the execution environment, ensuring that the binary can run correctly on the target system. While the linker prepares the ELF file by creating a fully linked executable, the loader is tasked with making that executable ready for execution in memory. Figure 2.2 shows the dual nature of ELF files.

2.2.1 Statically vs. Dynamically linked ELF

ELF binaries can be produced through two primary linking strategies: *static* and *dynamic* linking. Although both approaches generate ELF files, they differ significantly in structure, dependency management, and runtime behavior.

Statically linked ELF. In a statically linked binary, all required code and libraries are incorporated directly into the executable at link time. Consequently, the resulting ELF file is self-contained and does not rely on any external shared libraries at runtime. The linker resolves all symbol references during linking, embedding both code and data sections of the dependent libraries into the final binary. This design simplifies deployment, as the executable can run on any compatible system without requiring external dependencies. However, static linking increases binary size and can lead to redundancy when multiple binaries include identical library code. Moreover, maintenance becomes more complex, as updating a library requires relinking and redistributing every statically linked executable that uses it. Unikraft exclusively produces statically linked ELF binaries. This design choice aligns with the unikernel philosophy of creating specialized, self-contained images that include only the exact components required for a specific application.

Dynamically linked ELF. In contrast, a dynamically linked ELF binary defers the resolution of external library symbols to runtime. Instead of embedding all library code, it contains references to shared libraries (typically with the `.so` extension) that are loaded by the dynamic linker (`ld-linux.so` in Linux) when the program starts. These binaries include additional sections such as `.dynamic`, `.dynsym`, and `.dynstr`, which store metadata describing the external dependencies and exported symbols. During program initialization, the dynamic linker maps the necessary shared libraries into memory, resolves symbol addresses, and updates the relocation entries to redirect function calls appropriately.

2.2.2 Symbolic Information

Some sections within an ELF binary are dedicated to storing symbolic information. For instance, the `.symtab` section contains symbol table entries that associate the

names of functions, variables, or other entities – stored as strings in the `.strtab` section – with their corresponding addresses and attributes in the binary.

In some cases, however, binaries may be stripped to reduce size or conceal implementation details. Stripping removes non-essential symbols, such as entries in the `.symtab` section, which are primarily useful for debugging. Nevertheless, stripping does not affect the symbols required for dynamic linking. These are maintained in separate, dedicated sections: `.dynsym` for the symbol entries and `.dynstr` for the corresponding strings. The dynamic symbol tables must be preserved to allow the dynamic linker to resolve external references during program execution.

2.2.3 Relocations

Relocation is the process of connecting symbolic references with symbolic definitions. For example, when a program calls a function, the associated call instruction must transfer control to the proper destination address at execution. In essence, relocations bridge the gap between symbolic references produced during compilation and concrete memory addresses computed at link time or load time.

From the linker's perspective, relocations arise as a consequence of code generation and assembly. Object files contain references to symbols whose final addresses are not yet known, such as functions or global variables defined in other compilation units. To represent these unresolved references, the compiler and assembler emit relocation entries that record where address fix-ups must occur and how they should be applied. Traditionally, during static linking, the linker resolves all symbol references, computes final addresses, and applies the corresponding relocations, producing an executable with no remaining unresolved relocations.

In the ELF format, relocations are represented using dedicated sections, most commonly `.rela.*`, which are associated with specific sections such as `.text` or `.data`. Each relocation entry specifies the location to be patched, the referenced symbol, and a relocation type that defines how the symbol's final address must be computed.

Relocations also play a crucial role in dynamically linked ELF binaries. Unlike statically linked executables, dynamically linked binaries defer some relocations until program load time or even until the first invocation of a function. These dynamic relocations are processed by the dynamic linker after the executable and its shared libraries have been mapped into memory. At runtime, the dynamic linker applies relocations to adjust references to shared libraries, global variables, and function entry points according to the actual memory layout. This mechanism relies on cooperation between relocation entries and runtime indirection tables, which are discussed in the following section.

2.2.4 Procedure Linkage Table, Global Offset Table and Lazy Binding

While Unikraft produces statically linked binaries and does not depend on the mechanisms described in this section, we include them to facilitate comparison with the dynamic linking paradigm in later chapters and to better support the analysis of traditional applications prior to porting them to Unikraft.

In dynamically linked ELF binaries, dynamic linking is essential for enabling efficient code reuse and modularity through shared libraries. To support dynamic linking at runtime, ELF uses several specialized sections, notably the Global Offset Table (GOT) and the Procedure Linkage Table (PLT).

The Global Offset Table (.got). The `.got` section contains addresses used for dynamic symbol resolution. It acts as an indirect jump table for both functions and global variables defined in shared libraries. The `.got` is divided into two parts: the `.got` itself handles data symbols (e.g., global variables), while `.got.plt` stores function information used with the `.plt`. Although technically separate, the term `.got` is often used loosely to refer to both sections.

The Procedure Linkage Table (.plt). The `.plt` section is a sequence of trampoline stubs – small blocks of code inserted into the binary to handle indirect function calls to external (shared library) functions. Rather than calling a shared library function directly, the compiled code calls the corresponding entry in the `.plt`. Each `.plt` entry typically performs an indirect jump via the corresponding `.got.plt` entry. If the resolution has not yet been performed, this indirection allows the dynamic linker to resolve the actual memory address of the target function at runtime.

Lazy binding. Lazy binding is a performance optimization strategy where the actual resolution of a shared library function's address is deferred until the first time that function is called.

Function resolution example. When calling a library function, the resolution, illustrated in Figure 2.3, works as follows:

① In the executable code, the function `puts` is called. The compiler translates it to a call to the corresponding PLT stub, `puts@plt`.
② The PLT stub begins with an indirect jump instruction, which jumps to an address stored in the `.got.plt` section.
③ Before lazy binding occurs, this address initially points to the next instruction in the function stub, which is a `push` instruction.
④ After pushing an integer that identifies the PLT stub, the execution jumps to a shared default stub common to all PLT function stubs.
⑤ The default stub first pushes an identifier – retrieved from the `.got.plt` – that represents the main executable, then performs an indirect jump (also via the

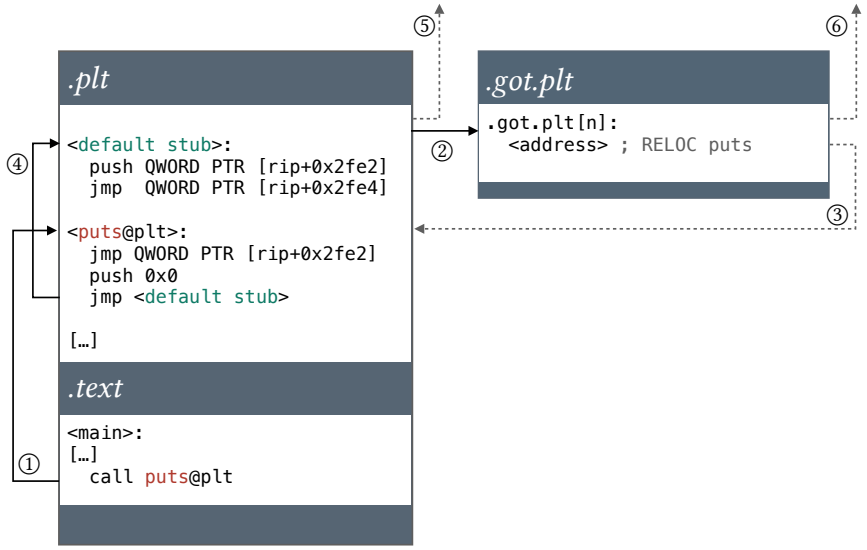


Figure 2.3: *Using the PLT and GOT to call external functions. The dynamic linker resolves the address of* `puts` *the first time it is called.*

`.got.plt`) to the dynamic linker. The linker uses this information to determine that it must resolve the address of `puts` on behalf of the main executable currently loaded into the process.

⑥ On subsequent calls, the `.plt` entry performs the same indirect jump (step ②), but this time the `.got.plt` entry contains the correct address of the library function, eliminating the need for further resolution.

Figure 2.3 illustrates this behavior using the example of the `puts` library function. Note that this figure represents the contents of the ELF file prior to relocation, meaning the relocations have not yet been resolved. As a result, the `.got.plt` entry does not yet contain the actual address of the `puts` function.

2.3 Hypervisor

The hypervisor, also known as the Virtual Machine Manager (VMM), is a software layer that virtualizes physical resources, enabling multiple VMs to coexist and share a single physical host. As the core component of virtualization, it ensures efficient resource allocation, strong isolation, and flexible management of virtualized environments. In the context of unikernels, each unikernel instance runs as a single VM, relying on the hypervisor to provide isolation from other workloads.

Hypervisors are classified into two categories:

1. *Type-1 Hypervisor* (*Bare-Metal*): A Type-1 hypervisor, also known as a bare-metal hypervisor, runs directly on the host hardware without requiring an underlying operating system. By providing direct access to physical resources, it ensures high efficiency and strong security. Notable examples include Microsoft Hyper-V [215], and Xen [26].

2. *Type-2 Hypervisor* (*Hosted*): A type-2 hypervisor (also called a hosted hypervisor) runs as an application on top of a conventional OS. Unlike bare-metal hypervisors, it manages guest virtual machines as processes within the host OS. However, this architecture introduces additional overhead, as performance depends on the underlying OS. One representative example is VMware Workstation [216].

2.3.1 Firecracker

Firecracker [2] is a lightweight VMM implemented in Rust. It leverages the Kernel-based Virtual Machine (KVM) [1], a Linux kernel module that enables the kernel to function as a type-1 (bare-metal) hypervisor. Designed with minimalism in mind, Firecracker provides only the essential features required to run microVMs – VMs typically designed to provide fast startup, strong isolation, and efficient resource usage.

In Firecracker, launching a microVM involves configuring it with the required virtual hardware components, such as a virtual CPU, memory size, and boot parameters. The microVM, typically provided in executable format, is parsed by Firecracker, which maps its segments into guest memory with the appropriate protections. Firecracker then uses the KVM API to manage low-level hardware operations and launch the microVM.

2.4 Memory Deduplication

Memory deduplication is a memory optimization technique employed in operating systems and hypervisors to reduce redundant memory usage by identifying and merging identical memory pages. This optimization is particularly relevant in environments where multiple processes or VMs have common components (e.g., copies of the same kernel), leading to multiple identical pages occupying physical RAM. By detecting these duplicates and merging them into a single shared physical page (frame), memory deduplication reduces overall memory pressure and improves system scalability. Memory deduplication strategies are typically categorized into two main approaches: memory scanning-based deduplication and content-aware deduplication. The first approach involves a memory scanner that periodically scans the memory of multiple VMs and records fingerprints (hashes) of each page. When a match is found, the system compares the content of the corresponding pages and merges them if they are identical [139, 230, 218, 138]. This class of techniques is further discussed in Section 2.4.2, with a particular focus on KSM, the primary memory deduplication scanner integrated into the Linux kernel.

Content-aware deduplication, on the other hand, relies on semantic knowledge of the data or application to identify redundancy. Content-aware deduplication is employed in systems like Disco's Transparent Page Sharing (TPS) [37] and Satori

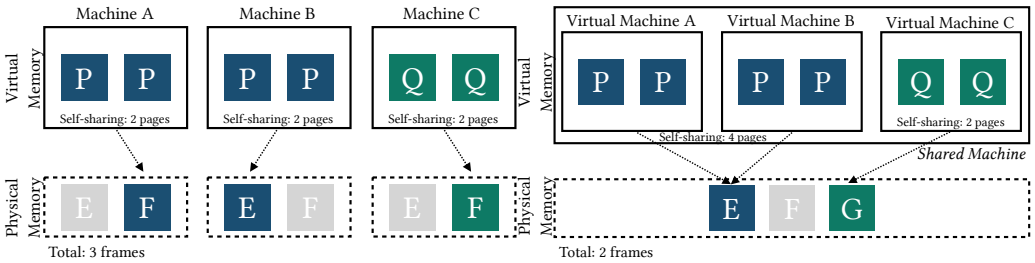
on Xen [140]. For instance, in Disco, when a VM reads from a *Copy-on-Write (CoW)* disk, the VMM checks whether the corresponding block already resides in memory. If so, it maps the VM's pages to those existing memory pages, avoiding duplication and enabling efficient memory sharing.

Although memory deduplication reduces memory consumption, it entails CPU overhead [11, 138, 230] and opens the door to side-channel attacks [231, 189, 32, 22].

2.4.1 Types of Sharing

We separate sharing into two essential categories: self-sharing (also known as intra-sharing) and inter-sharing. Self-sharing refers to deduplication within a single (virtual) machine or process, where redundant memory pages – such as repeated data structures or code – are merged. Inter-sharing, on the other hand, occurs across multiple VMs or processes, enabling them to share identical pages, such as common code/data. We illustrate the difference between self-sharing and inter-VM sharing with a simple example, illustrated in Figure 2.4.

Consider three machines A, B, and C, each containing shareable memory content as illustrated in Figure 2.4a. Machines A and B each hold two copies of the same page (P), while machine C holds two identical copies of another page (Q). When evaluated individually, each machine can reduce its identical pages to a single physical frame thanks to self-sharing, saving one frame per machine – for a total of three frames used (one frame per machine).



(**a**) *Self-sharing within single machines* (**b**) *Inter-VM sharing between VMs*

Figure 2.4: *Self-sharing vs. inter-VM sharing.*

When A, B, and C are instead co-located as virtual machines on the same physical host (Figure 2.4b), the hypervisor can additionally perform inter-VM sharing. Since VMs A and B contain identical page content (P), their pages can be merged and backed by a single physical frame. In contrast, VM C's pages (Q) cannot be shared with the others due to differing content. As a result, inter-VM sharing yields one additional frame of memory savings compared to intra-machine sharing alone, reducing the total number of physical frames from three to two.

2.4.2 Kernel Samepage Merging (KSM)

Originally developed for KVM, the KSM [11] daemon allows a Linux system to share identical memory pages between several processes and/or virtualized guests. KSM relies on a background service (ksmd), which periodically scans areas of user memory which have been registered through the `madvise` system call [129], looking for pages pointing to different physical memory frames with identical content. When such pages are found, they are *merged*, i.e., they are made to point to a single frame, and the other frame(s) is/are reclaimed. The merged pages are marked CoW, so that they will automatically be duplicated if a process updates its content later.

KSM manages memory pages using two distinct red-black trees. Pages tracked by KSM are initially placed in the "unstable tree", which represents pages whose contents may still change. This tree is rebuilt from scratch after each scan cycle to reflect the latest page states. Pages that have been successfully merged are placed in the "stable tree", which contains write-protected, shared pages whose contents are assumed to be immutable. Because these pages should not change, the stable tree does not require regular rebuilding. A page remains in the stable tree until any referencing process modifies it or all processes unmap it, at which point it is removed and returned to the unstable tree for rescanning.

For each page scanned, KSM first searches for a match in the stable tree. If a match is found, the current page is merged with the corresponding page. If not, KSM checks whether the page has changed since the last scan of this page using a checksum. If the checksum has changed, it is updated and the page is skipped to avoid tracking unstable content. If the checksum is unchanged, KSM searches the unstable tree for a match. A successful match leads to merging and promotion of the resulting shared page to the stable tree, while the matched page is removed from the unstable tree. If no match is found, the page is added to the unstable tree for future comparisons.

KSM is not the only memory scanner available. Indeed, there exist others such as Cross Layer I/O-based Hints (XLH) [139] and Ultra Kernel Samepage Merging (UKSM) [230], but KSM is the one currently integrated into the Linux kernel codebase [12].

Part II

Unikernels & Memory Deduplication

This page intentionally left blank.

3

UNIKERNELS AND MEMORY DEDUPLICATION

Overview

In this chapter, we address the challenges of running multiple unikernels on the same server. While unikernels are designed to have a small memory footprint per instance, their inherent specialization can significantly limit the effectiveness of memory deduplication. As a result, deploying multiple Software as a Service (SaaS) or Function as a Service (FaaS) unikernels on the same machine can lead to unexpectedly high memory consumption, reducing their scalability advantages. To address this limitation, we introduce Spacer, an alignment-based methodology for building memory-efficient unikernels. Spacer reorganizes the unikernel memory layout to enhance page sharing across deployed instances, significantly improving memory efficiency while maintaining performance and isolation.

3.1 Introduction

Due to their specialization, unikernels offer high performance with a minimal footprint, allowing the deployment of hundreds or even thousands of instances on a single physical server [131]. However, as the number of unikernels running on a single server increases, new challenges arise, particularly concerning memory consumption and resource management. This leads to a critical question for cloud providers and system designers: *Are unikernels truly suited for serverless computing, and can they efficiently support a multitude of SaaS or FaaS instances on a single machine?* A key consideration when answering that question is *memory deduplication*, a technique that can reduce the memory footprint of multiple VMs by identifying memory pages with identical content and *merging* them, i.e., making them point to a single shared

memory frame. This approach has proven highly effective in traditional VMs, which often share the same guest kernel, system libraries, and other resources [11, 139, 138, 27, 159, 218].

Nevertheless, the specialized and compact design of unikernels may limit the effectiveness of memory deduplication mechanisms. Unikernels are typically statically linked and engineered to use as little memory as possible, embedding only the strictly necessary libraries directly into a highly compact binary. Although this specialization makes them efficient individually, it also limits opportunities for memory deduplication when multiple instances run on the same server. Since many memory pages in unikernel images are unique, very few can be merged. As a result, deploying and running many unikernels on the same host can lead to substantial memory consumption. Randomizing library placement to support Address Space Layout Randomization (ASLR) worsens this issue by further reducing the potential for memory page sharing.

To address this challenge and realize the promise of running very large numbers of unikernels on a single server with minimal memory overhead, we introduce Spacer, a tool that analyzes large sets of unikernel images prior to deployment and automatically arranges their memory layouts by aligning and placing libraries at absolute addresses. This reorganization significantly increases the number of identical memory pages across unikernel instances, enhancing memory deduplication and overall efficiency. A natural choice for our prototype is Unikraft [105], an open-source unikernel development framework that emphasizes statically linked, highly specialized images. Its open-source nature and active community further make Unikraft an ideal platform for our experiments.

However, placing libraries at fixed absolute addresses introduces security vulnerabilities [176, 74]. To mitigate this, we developed Spacer (ASLR), an enhanced version which performs ASLR during link-time. We evaluate our approach using KSM [11], the default memory deduplication scanner in Linux and demonstrate that both Spacer and Spacer (ASLR) can reduce memory consumption by up to 3× compared with unikernels built with Dead Code Elimination (DCE) [196].

Our main contributions are: (1) We highlight and quantify the inefficient memory consumption exhibited by multiple running unikernels. (2) We introduce a new methodology based on page alignment which reduces memory consumption (in number of frames) when several unikernels run on the same machine. (3) From this approach, we derive Spacer and its ASLR version, two proof-of-concept prototypes which rearrange a unikernel image's memory layout so that it is more likely to have identical pages with other unikernels deployed on the system. (4) We provide an evaluation for FaaS and SaaS by comparing our approach with other configurations including DCE-optimized images. (5) We discuss Spacer's limitations and the scenarios in which it may be less relevant, especially for heap-intensive applications.

3.2 Problem Statement and Possible Solutions

Unikernels are built on the concept of a library OS [60, 213], where operating system functionality is decomposed into libraries with well-defined behavior. Each unikernel is fully statically linked, producing a single binary image that contains both the application and the set of libraries on which it depends. Consequently, one might expect effective memory deduplication, since all unikernels rely on a largely similar set of libraries. However, not all unikernels include the same components, and even when they do, their memory layouts may still differ due to specialization and compactness.

Consider the two unikernels in Figure 3.1. We initially assume that all libraries are independent (i.e., they do not interact with each other). When the unikernels are loaded into memory, their `.text` section – containing the binary instructions – is spread across several memory pages. Recall that modern OSes manage memory in fixed-size pages, typically 4 KiB each. These pages (and their content) are depicted as dark grey rectangles in Figure 3.1.

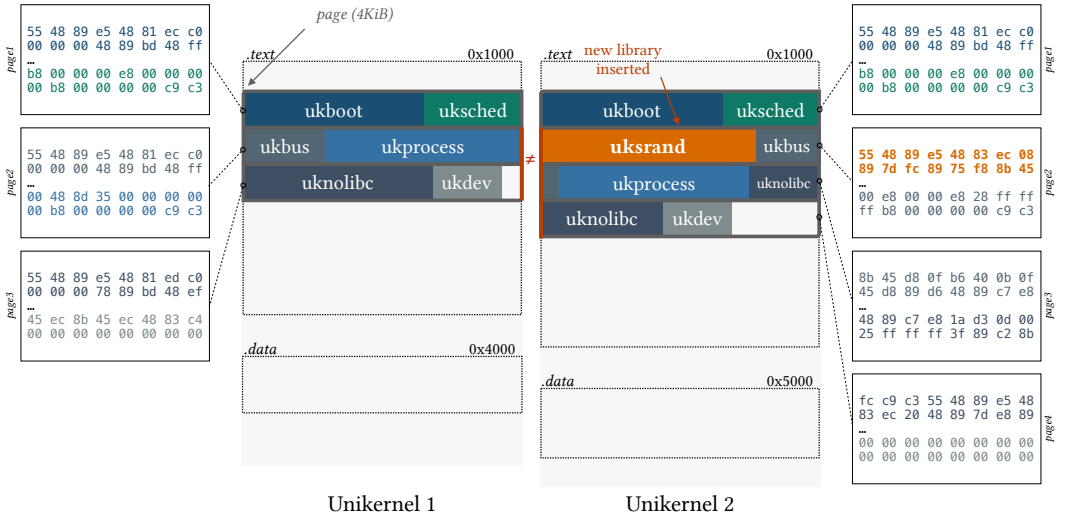


Figure 3.1: *The insertion of the uksrand library into the second unikernel shifts all subsequent libraries in memory. Only the first page of both instances can be shared.*

Depending on their size and offset, each library can either be contained within a single page or be divided into several pages. For instance, *uknolibc* is contained within a single page in the first instance and is spread over two pages in the second instance. The insertion of an additional library (*uksrand*) in the second instance modifies the order of all subsequent libraries in memory and their offset to page boundaries (i.e., the start of a page). In this example, the initial pages containing *ukboot* and *uksched* can be shared, but the remaining memory pages differ, even though both instances share most libraries.

Reordering the libraries can increase sharing, but the effect is limited, as some pages remain unshareable. This is the case in Figure 3.2, where *uksrand* is inserted at the end of the second unikernel.

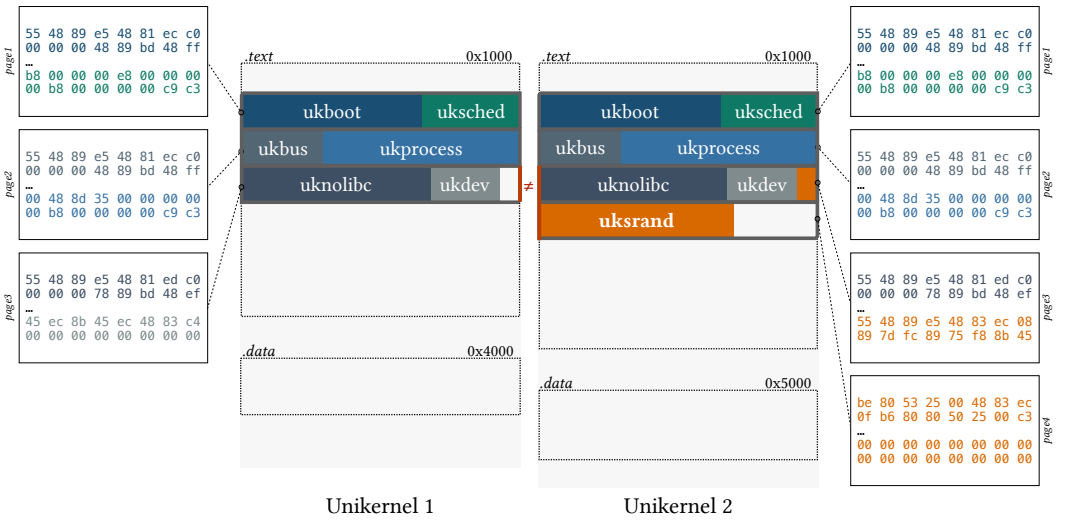


Figure 3.2: *Reordering the libraries can increase sharing, but the effect is limited, as some pages remain unshareable. The first two pages can be shared; others cannot.*

In this configuration, the first two pages of each instance can be merged in two frames. However, the page containing *uknolibc* and *ukdev* cannot be shared because a fraction of *uksrand* is present on the second instance. Moreover, the placement of libraries is determined by the underlying build system, which considers unikernels individually and does not have a global view of all the libraries used in multiple unikernels.

A potential solution to prevent libraries from shifting across page boundaries is to align each library to a page boundary and insert zero padding between them, ensuring each library starts at a page boundary. This approach is shown in Figure 3.3.

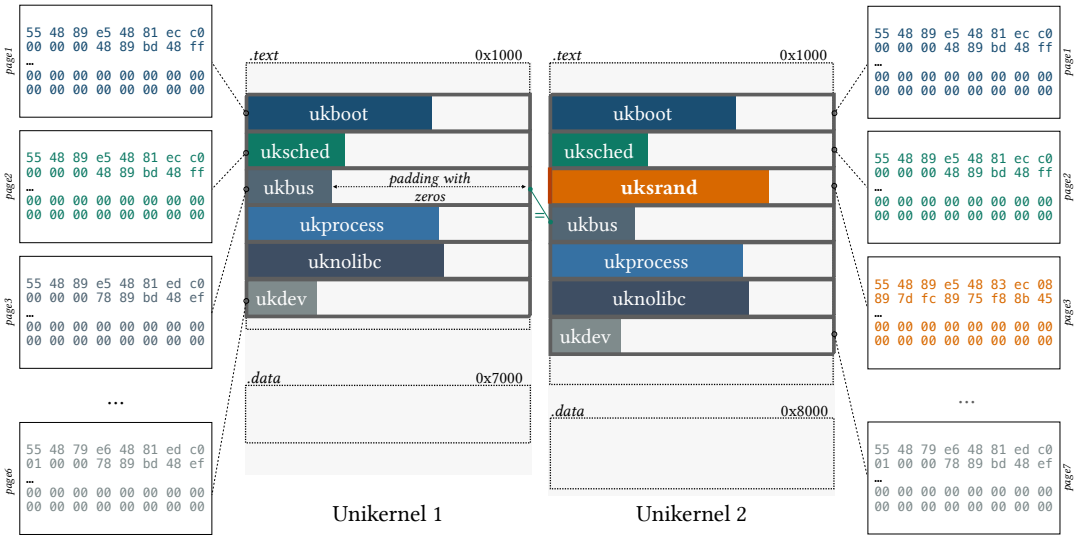


Figure 3.3: *Aligning libraries on page boundaries in the virtual address space increases the sharing. Only uksrand cannot be shared, as it is unique.*

All libraries present in multiple instances can then be shared, leaving only *uksrand* unshareable in our example, as it is unique to the second instance. This ap-

proach guarantees that the set of pages corresponding to a library remains independent of the libraries preceding or following it.

For simplicity, we previously assumed that all libraries were independent, but this is rarely the case in practice. Libraries can interact with each other through specific binary instructions, such as `lea` for data manipulation and `call` for function invocation, both of which use RIP-relative addressing[1]. Consequently, instructions that reference addresses from other sections or libraries can differ across instances. Only aligning libraries to page boundaries is therefore insufficient when unikernels have differing library ordering, as variations in cross-reference addresses still lead to divergence in page content. This is illustrated in Figure 3.4, where instructions that reference addresses from other sections (e.g., `.data`, which stores variables) or other libraries (e.g., *uksched* calling a function in *ukbus*) differ across instances. Even minimal differences – such as a single byte – are sufficient to render pages distinct, thereby preventing memory deduplication. As a result, pages that would otherwise be identical cannot be shared when running multiple unikernels concurrently on the same host.

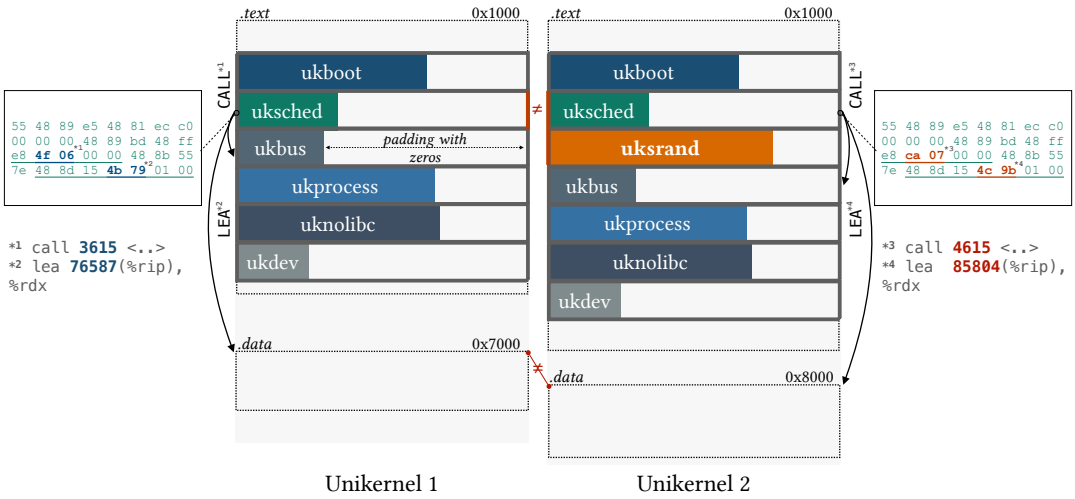


Figure 3.4: *Aligning libraries to page boundaries still results in cross-references in instructions such as `call` or `lea` which use different addresses and therefore reduce sharing opportunities.*

To ensure the binary instructions manipulating memory addresses are the same in all instances, the addresses must be identical. Each section and each common library should be aligned to the same absolute addresses across all instances. This solution is illustrated in Figure 3.5. In this figure, each library and section starts at identical addresses. As a result, instructions manipulating addresses use the same values, leading to identical binary code resulting in shareable pages.

Note that some libraries introduce global variables in the `.data`, `.rodata`, and `.bss` sections. To ensure that a given variable always resides at the same address across different instances, we use a similar approach – padding within the section

[1] In x86-64, RIP-relative addressing allows instructions to compute memory addresses as offsets from the instruction pointer (RIP) [93].

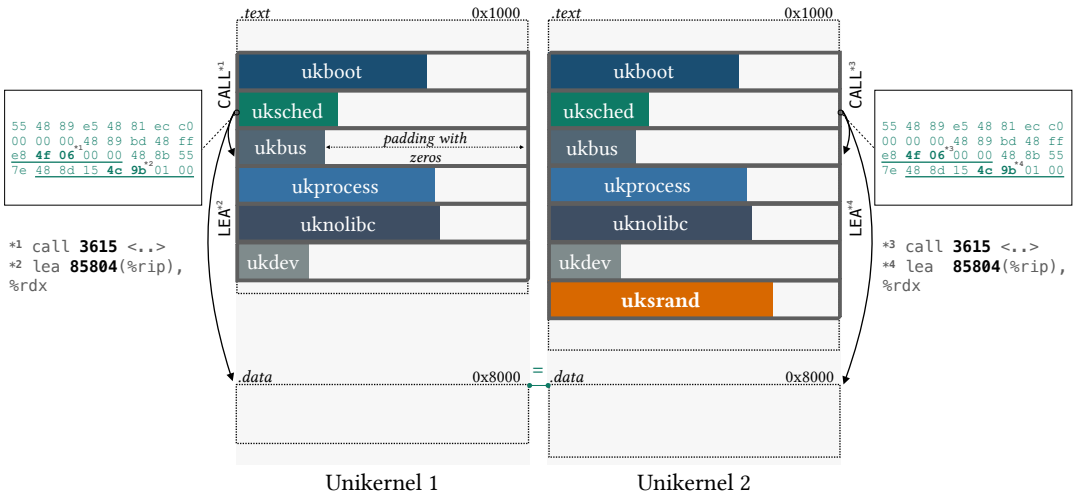


Figure 3.5: *With sections and libraries at the same absolute addresses in the virtual address space of all instances, we can obtain identical pages and maximize memory sharing.*

to guarantee a unique mapping between variables and memory addresses.

To illustrate the problem and the proposed solution more clearly, we have considered only two instances so far. When more than two unikernel instances use different sets of libraries, the solution becomes more complex. If some libraries are common only to a subset of instances, they must be aligned to specific addresses, creating *gaps* in the virtual memory space of certain instances. Figure 3.6 shows a scenario where the *uksrand* library is aligned by leaving gaps in the second instance, ensuring the same addresses in the binary instructions across all other instances. Similarly, the *ukring* library is also aligned using padding in certain instances to preserve address consistency.

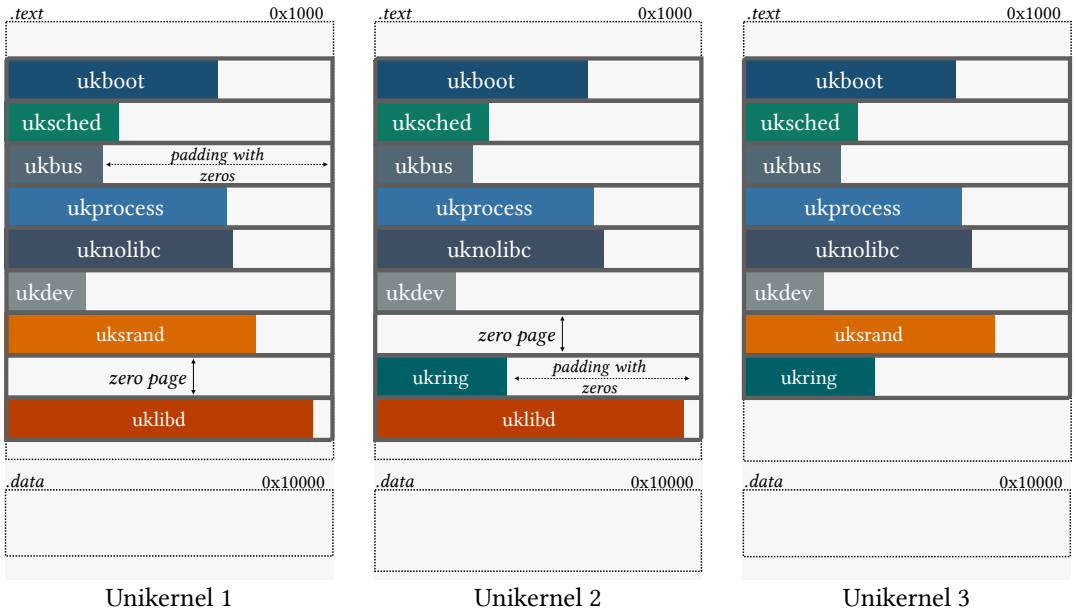


Figure 3.6: *Aligning different sets of libraries across multiple instances creates gaps in the virtual memory space of some instances.*

From a security standpoint, the main limitation of the previous approach is the fixed placement of libraries at absolute virtual addresses, which bypasses ASLR and introduces a security risk. We propose a simple approach to overcome this problem and enable ASLR and memory deduplication to coexist by varying the locations of libraries and sections across different unikernel instances. As for the vanilla version, we begin by aligning all libraries on page boundaries again. Any instruction that references an address from another section or library is problematic, as it creates cross-library dependencies that prevent identical page content. We relocate these problematic instructions to a dedicated trampoline table, placed at the next page boundary. Each such instruction is replaced with a relative call to its corresponding trampoline entry, ensuring that offsets remain identical across all randomized layouts. Libraries that are used in multiple unikernels have a corresponding trampoline table. Figure 3.7 illustrates this approach where ASLR is applied on different unikernels. In this case, all problematic instructions of the *uksched* library will be moved to a trampoline table called *tpl.uksched*.

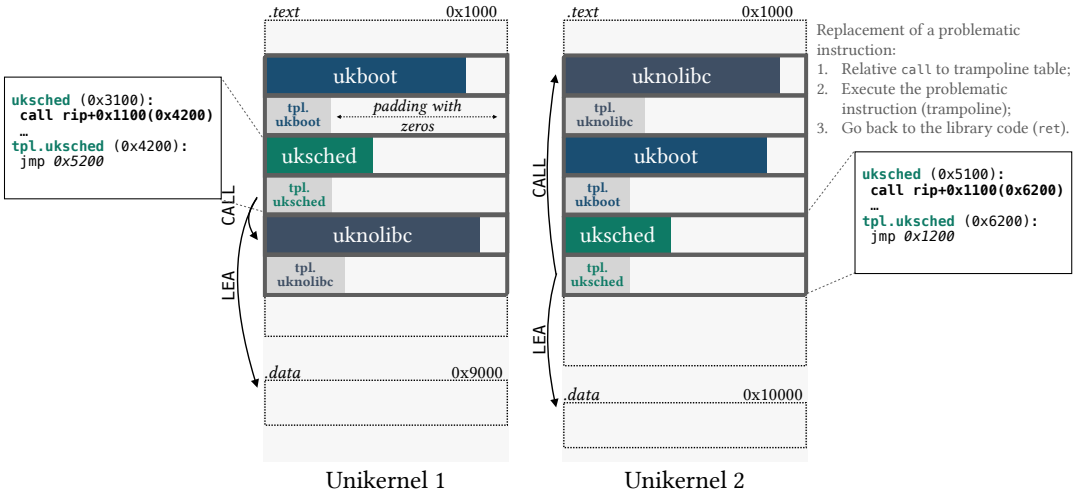


Figure 3.7: *To support ASLR while sharing code pages, we add one instance-specific trampoline table per library. These tables contain problematic instructions (e.g., those referencing addresses in other sections or libraries).*

The location of the trampoline table is not irrelevant. In our approach, each trampoline table is placed immediately after its corresponding library, allowing the library's instructions to make relative calls to it. A single global trampoline table at a fixed absolute address would not work, as the offset to problematic instructions would vary between instances. Likewise, using existing approaches such as Position Independent Code (PIC) will also impact memory sharing. This will be further discussed in Section 3.5.4.

3.2.1 Optimizing the Virtual Memory Layout

We identified two additional optimizations that can be applied to the virtual memory layout of unikernels to enhance memory sharing when running multiple in-

stances.

Compacting Core Libraries

In Unikraft, libraries are first-class components, and all core libraries – from the scheduler to the memory allocator – are interchangeable. However, when all unikernels are built using the same set of core libraries, some memory optimizations become possible. Specifically, the number of required pages and frames can be reduced through compaction. Since core libraries are always shared across instances, they do not need to be aligned on page boundaries. Instead, only libraries that are unique or shared by a subset of instances require such alignment, leading to more efficient memory usage.

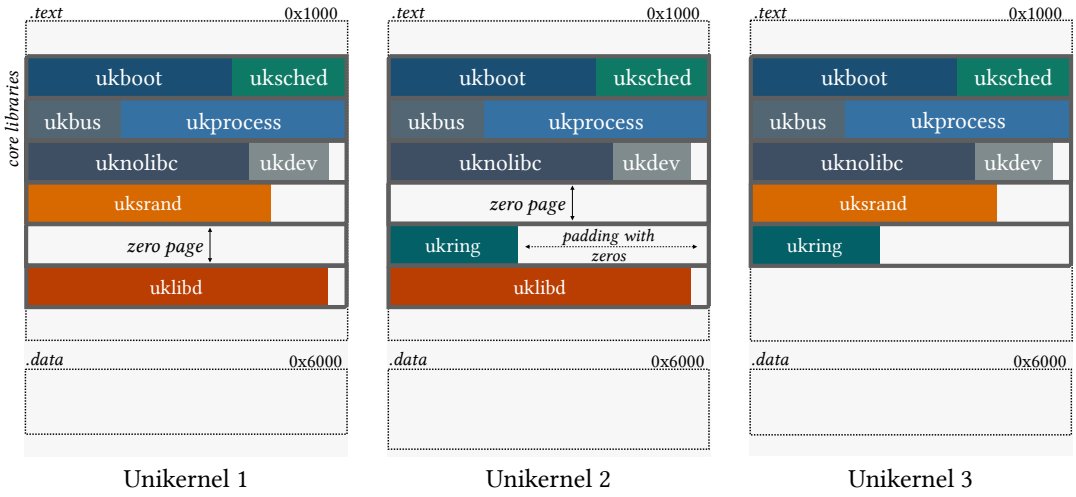


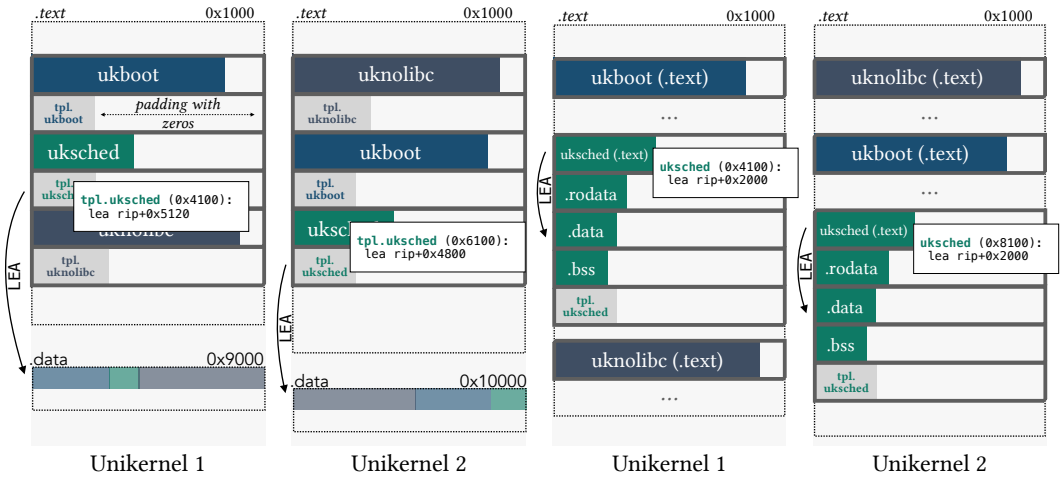
Figure 3.8: *Core libraries that are always integrated in all unikernels can be compacted on the same pages to reduce the number of frames. This optimization reduces the requirement from 9 frames to just 6 frames.*

In Figure 3.8, *uksrand*, *uklibd*, and *ukring* are aligned on a page boundary since they are not common to all instances. The remaining libraries are core libraries shared across all instances and do not require such alignment. If each library were blindly aligned on a page boundary, the total number of code frames after memory deduplication would be 9. With compaction, only 6 frames are needed in this example. This optimization is left to the developer, who can define the core libraries and apply compaction as needed.

Disaggregate Other Sections

To support ASLR, we use trampoline tables to isolate problematic instructions. In many cases, these instructions access data from the library's own `.rodata`, `.data`, or `.bss` sections. These addresses must be included in the trampoline table because, by default, the placement of each library's sections in memory changes between builds, depending on the library order. Traditionally, all data-related sections across

libraries are aggregated together, causing their locations to be dependent on this order, as can be observed in Figure 3.9a.



(a) *Old ASLR representation.* **(b)** *New ASLR representation.*

Figure 3.9: *Old vs. New ASLR representation: In (a), all libraries sections are aggregated into global sections (only `.data` is shown). In (b), the `.rodata`, `.data`, and `.bss` sections of each library are disaggregated and placed directly after the corresponding library's `.text` section.*

To reduce this overhead, we disaggregate the `.rodata`, `.data`, and `.bss` sections of each library and place each, on individual pages, directly after the corresponding library's `.text` section. This layout ensures that the relative position of each section with respect to its code remains fixed across builds. As a result, references can rely on identical relative offsets instead of different addresses, significantly reducing the trampoline table size. Additionally, it avoids security risks by separating immutable read-only data from mutable sections (`.data` and `.bss`), allowing memory pages to be assigned appropriate protection flags and preventing unintended writes to read-only data. Figure 3.9b illustrates this optimization.

3.3 Spacer: Aligning and Inflating Unikernels

To enable effective memory deduplication for unikernels, we introduce Spacer. Its high-level architecture is shown in Figure 3.10.
Spacer's functionality is divided into three different components:

① A binary analysis framework disassembles unikernel images and object files to extract information about sections. This information gives Spacer a comprehensive view of all libraries used across unikernels running on the same machine (workspace). Additionally, it can process multiple unikernel binaries to provide theoretical statistics on potential memory sharing by comparing binary content.

② Spacer generates a global map of the different libraries. In the first stage, libraries common to all instances are associated with absolute virtual addresses (aligned on page boundaries). Then, all other libraries, depending on their

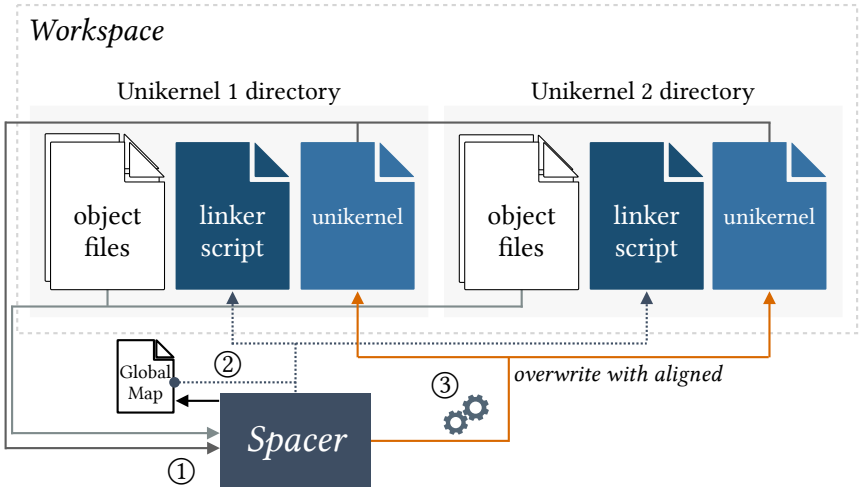


Figure 3.10: *High-level architecture of the Spacer tool, divided into three steps: ① Analysis of unikernels, ② Computation of a global map and generation of linker scripts, and ③ Final relinking.*

frequency (i.e., the number of occurrences for all unikernels), are placed in the same way. Additionally, Spacer offers developers the option to compact core libraries into a single contiguous region by passing a specific flag argument. Spacer then generates custom linker scripts that place libraries according to the previous map. To assign absolute addresses to each library and section, Spacer updates the location counter from the linker script specification [66]. An example of such linker script is given in Figure 3.11.

③ The previously generated linker scripts are used to build aligned unikernels through a final relinking procedure and existing unikernel binaries (default ones) are overwritten.

```
.text.libplat 0x106000 : { libplat.o(.text); }
.text.libpci  0x10c000 : { libpci.o(.text);  }
/* ... */
.rodata.libplat 0x29d000 : { libplat.o(.rodata); }
.rodata.libpci  0x29d8a0 : { libpci.o(.rodata);  }

/* Another alternative */
.rodata : {
    . = ABSOLUTE(0x29d000); libplat.o(.rodata);
    . = ABSOLUTE(0x29d8a0); libpci.o(.rodata);
}
```

Figure 3.11: *Snippet of a linker script used to align libraries and sections at absolute addresses.*

Using absolute addresses allows to handle the case where some libraries are only common to a subset of instances. As mentioned in Section 3.2, this approach introduces zero pages in the virtual memory space (but not in the ELF file). The virtual address space of each unikernel is relatively large (theoretically 64 bits); therefore, having memory gaps is not problematic. Spacer needs to analyze all unikernels on

the same machine in order to have global knowledge and generate the mapping according to libraries' occurrences. Nevertheless, Spacer does not have to scan the whole workspace too frequently (e.g., at every run) but only when unikernels are added or modified. In that case, it updates the global map and linker files accordingly.

3.3.1 Towards ASLR Support

To support ASLR, Spacer must be adapted accordingly. Spacer (ASLR) is an extension that developers can enable to generate ASLR-based unikernels.

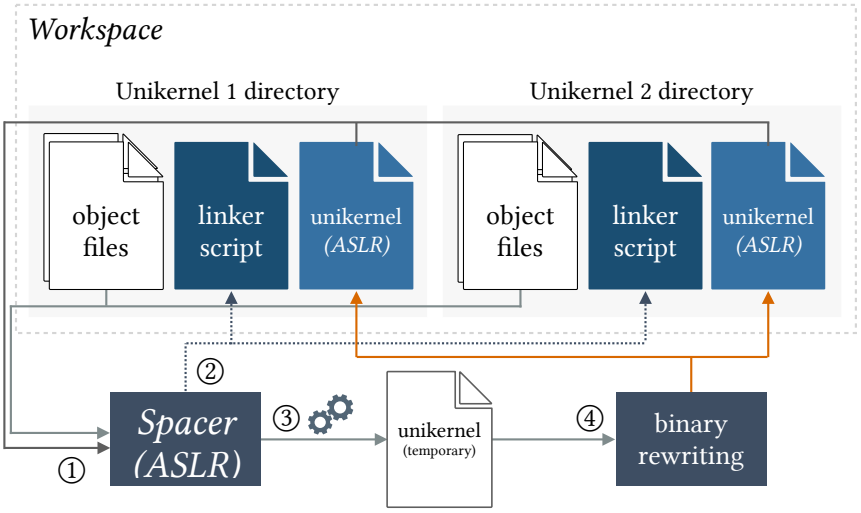


Figure 3.12: *High-level architecture of the Spacer (ASLR) tool, consisting of four steps: ① Analysis of unikernels, ② Shuffling libraries in linker scripts, ③ Building unikernels using these linker scripts, and ④ Performing an additional binary rewriting procedure to isolate problematic instructions.*

As seen in Figure 3.12, Spacer (ASLR) introduces an additional step compared with the vanilla version. It maintains the same behavior as before for step ① but slightly modifies steps ② and ③, and uses an additional step. These four steps are:

① A binary analysis framework disassembles unikernel images and object files to extract information about sections (the same as vanilla Spacer).
② It generates linker scripts in which libraries are shuffled and disaggregated into four consecutive subsections (`.text`, `.rodata`, `.data`, `.bss`). In addition, it creates empty trampoline tables for common libraries.
③ Then unikernels are linked again according to this new configuration. Note that ASLR is managed at the linking level via linker scripts, so it is necessary to link again the unikernels before each execution to benefit from a new ASLR layout.
④ A binary rewriting [224] technique is performed on the underlying ELF file to isolate problematic instructions. The binary is disassembled and analyzed with Capstone [16] and Lief [205], and each instruction that uses an address from another section or library is moved to a trampoline table.

We improve the binary rewriting process with the following optimizations, also illustrated in Figure 3.13:

1. **Handling single problematic instructions**

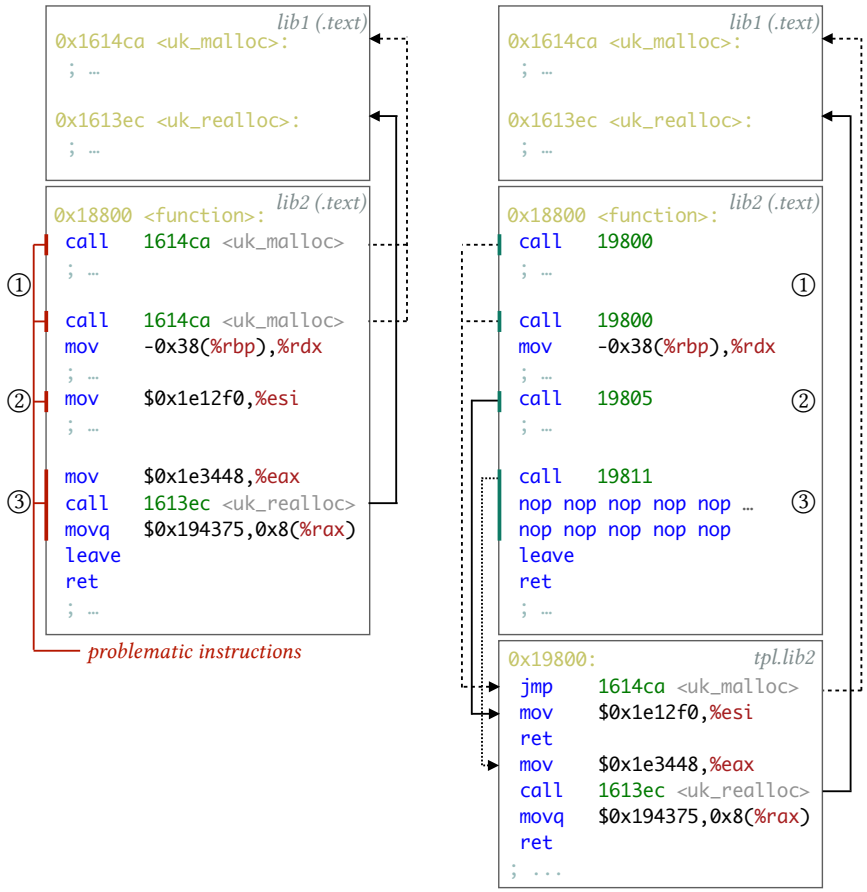
   - If the problematic instruction *is* a `call`, it is replaced by a `call` to the trampoline table, but the trampoline entry uses a `jmp` to the target instead of another `call` (see ① in the Figure).
   - If the problematic instruction is *not* a `call`, it is replaced by a `call` to the trampoline table. The trampoline entry contains the original instruction (with an updated offset if necessary), followed by a `ret` to return to the original execution point (see ② in the figure, where an address from another section is moved into register `esi`).
   - In both cases, instructions targeting the same address share a single trampoline entry to avoid duplication. `NOP` padding is applied if the original instruction is larger than the 5-byte `call` instruction.

2. **Handling consecutive problematic instructions**

   - If multiple problematic instructions appear consecutively, they are replaced by a single `call` to the trampoline table. To preserve code alignment, `NOP` padding is used in the library code (see ③ in the Figure).
   - The `call` transfers execution to the trampoline table, where these instructions are placed and updated (if necessary). After the last instruction in the trampoline table, a `ret` instruction is inserted to return to the original execution point.
   - Without this optimization, each instruction would require its own `call` to the trampoline table and a `ret` instruction to return, increasing overhead.

Although our approach requires relocating each problematic instruction and recalculating its offset relative to RIP (64-bit Instruction Pointer), it is easier to implement than directly updating the instruction with an address that would be stored in the trampoline table, in order to perform an indirect jump/call. The latter approach would require two instructions: (1) moving the address into a register and (2) using this register to perform an indirect `call` or `jump`. These instructions are always larger in byte size than the original, requiring all subsequent instructions in the binary to be shifted.

All these mechanics (e.g., re-linking and binary rewriting) can be executed "offline", long before the unikernels are executed.



(a) Program flow before binary rewriting (b) Program flow after binary rewriting

Figure 3.13: *Overview of binary rewriting optimizations: ① `call` replaced with a trampoline `call` whose entry uses a `jmp`. ② Non-`call` instruction replaced with a trampoline `call` containing the updated instruction and a `ret`. ③ Consecutive problematic instructions are grouped into a single trampoline `call` with `NOP` padding for alignment.*

3.4 Evaluation

This section presents the experimental results of evaluating various unikernel scenarios along several dimensions. The first aspect we examine is the overall memory savings achieved through library alignment. To evaluate this, we first examined memory sharing in unikernel by analyzing memory dumps (obtained from Firecracker [2]) and then conducted experiments directly with the KSM daemon. Although we chose KSM for our evaluation (since it is integrated within the Linux kernel [11, 12]), our approach, described in Section 3.2, is entirely agnostic to the underlying memory deduplication scanner, as long as it supports page sharing.

In addition to assessing unikernel memory usage, we also examined whether library alignment impacts ELF file size and application performance. To account for SaaS and FaaS use cases in our evaluations, we conducted experiments with both traditional applications ported as unikernels and unikernel-based lambda functions.

For SaaS, we worked with 10 applications that were ported as unikernels. These applications are as follows: the 'Helloworld' unikernel from Unikraft [105] (running an infinite loop), an FTP server [31], the iperf3 server [58], an HTTP proxy server [56], a DNS server [211], a DHCP server [239], SQLite [199], Nginx [163], Weborf [208], and HAProxy [193]. Since unikernels are cloud-oriented (i.e., they include a network stack and are primarily used as servers), we focused mainly on networked applications with different library subsets while ensuring diversity in workload characteristics. To capture non-networked behaviors, we additionally included Helloworld and SQLite. We specifically included the 'Helloworld' program to incorporate a different libc (nolibc [17]), whereas the other applications use newlib [162]. Table A.1 in the Appendix lists the applications used in this chapter and provides links to their sources. Although Unikraft supports some existing applications [118], porting complex new applications (e.g., Apache2 [64]) remains relatively time-consuming.

Based on this diversity, the observed trends are indicative of typical server-oriented unikernel deployments. However, the magnitude of deduplication benefits varies across applications, primarily depending on their memory behavior. In particular, memory-intensive workloads such as in-memory databases represent a lower bound for deduplication gains, as discussed in Section 3.5.1.

Afterwards, we conducted evaluations using FaaS. We first tested multiple instances of the same FaaS-based unikernel, then extended our analysis by experimenting with slightly different unikernel instances to simulate the expected behavior of compiled lambda functions running on the same server.

For each experiment, we used six different configurations: (1) *Default*, a Unikraft build with default settings; (2) *DCE*, which enables Dead Code Elimination to further reduce unikernel sizes; (3) *Spacer*, where we applied our aligned memory layout to improve sharing; (4) *Default (ASLR)*, the default build with ASLR support;

(5) *DCE (ASLR)*, the DCE build with ASLR support; and (6) *Spacer (ASLR)*, the Spacer build with ASLR support.

As explained in Section 3.3.1, ASLR has been managed at the linking level (via linker scripts). For all three ASLR configurations, we shuffled libraries and added random offsets between each library. To ensure a fair comparison, the same shuffle and offsets were applied consistently across these ASLR builds.

All experiments of this chapter were conducted on a Debian 11 (bullseye) server using a Linux kernel 6.1, gcc 10.2.1 (GNU ld 2.35.2 as linker). The host machine used for the experiments has 32 GB of RAM and an Intel® Xeon® CPU (E5-2650 v2 @ 2.60 GHz) with 32 cores. In addition, Unikraft Enceladus 0.8.0 (with Firecracker support [209]) has been used for the experiments.

3.4.1 Memory Sharing: Analyzing Memory Dumps

We conducted several experiments to quantify the number of memory pages shared (i.e., pages with identical content) across multiple unikernels, each running a distinct application. We began by working on memory dumps of unikernels without relying on KSM. To obtain these traces, we ran each unikernel with Firecracker as hypervisor and dumped the complete virtual address space of each guest into a file. To analyze the number of identical pages across multiple unikernels, we first calculated the number of pages per section using information extracted from the ELF file (e.g., section address, section size, etc.). We then divided the memory traces accordingly. Figure 3.14 presents the number of pages per section for the 10 applications across the six configurations. Spacer unikernels logically require more pages due to library padding (and trampoline tables in the ASLR version). Spacer increases the number of pages by 75% and 15% compared with the DCE and Default configurations, respectively. Link-time ASLR also increases the number of pages in the `.text` section, as random offsets are introduced between each library. Other sections, such as `.rodata` and `.data`, remain the same size except in Spacer configurations, where they are also inflated due to alignment. This inflation is particularly noticeable in Spacer ASLR, where libraries are disaggregated into different sections, leading to a higher number of pages. The trampoline tables used to isolate problematic instructions account for approximately 6% of the total pages in Spacer ASLR.

Then, we computed the sharing between multiple applications and the theoretical required frames. To perform page comparisons, we simply walk through each page of memory, calculate a hash based on its contents, and check for identical hashes. If two hashes match, we perform a byte-by-byte comparison to eliminate false positives. Figure 3.15 summarizes the key results from our measurements and represents the number of shared pages as well as their corresponding number of frames (after reduction) for 10 applications. As a first observation, we note that the DCE and Default configurations exhibit a very low sharing ratio (both <6.5%) in the vanilla setup, which decreases further with ASLR (both <5.5%). ASLR being

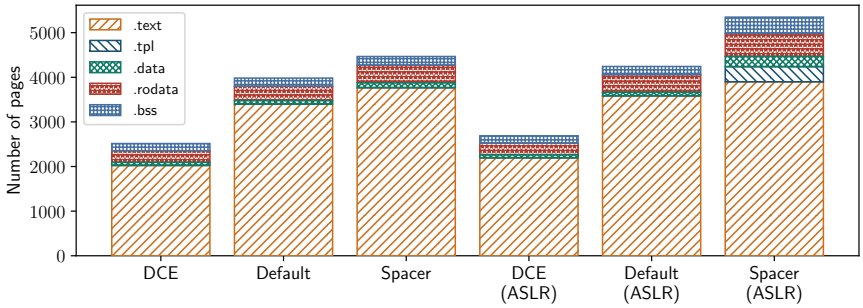


Figure 3.14: *Total number of pages used by sections of 10 applications ported as unikernel according to 6 different configurations (no memory deduplication). Spacer and Spacer (ASLR) consume more pages.*

worse is expected, as libraries are shuffled and random padding between them is introduced, increasing uniqueness and further reducing page sharing.

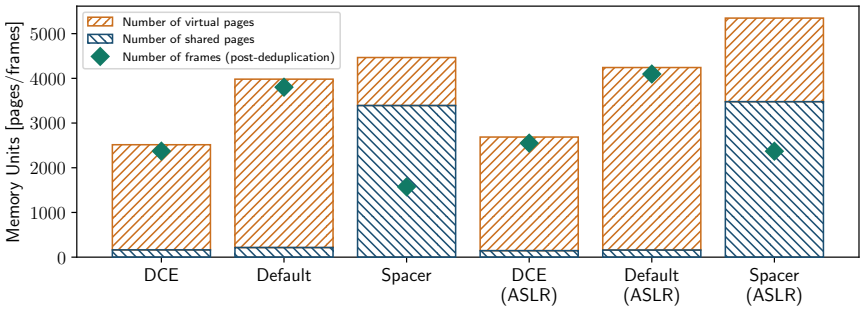


Figure 3.15: *Total number of pages, shared pages and frames (after memory deduplication) of 10 unikernels. While Spacer increases the number of pages per instance, it significantly decreases the number of physical memory frames needed.*

While Spacer significantly inflates the number of pages compared with DCE and Default instances, it achieves a much higher page-sharing rate. The vanilla Spacer configuration reaches a 75% page-sharing rate, while the ASLR version achieves a slightly lower 65% page-sharing rate due to the overhead introduced by trampoline tables. This enhanced page sharing leads to better memory reduction, resulting in significantly lower memory consumption (in terms of frames) when using unikernels aligned by Spacer. Spacer can lead to a 35% reduction in frames compared with DCE and a 60% reduction compared with Default based on memory dumps analysis. The Spacer (ASLR) version introduces a slight overhead compared with the vanilla version but still uses fewer frames than both DCE and Default configurations.

3.4.2 Memory Sharing: Relying on KSM

After analyzing potential page sharing and memory reduction through direct examination of memory dumps, we tested our method using the KSM daemon with both SaaS- and FaaS-based unikernels. We instrumented KSM to measure memory usage during unikernel execution, enabling us to verify the theoretical results computed by our binary analyzer tool. This confirms that Spacer (and Spacer ASLR)

unikernels indeed require fewer memory frames when multiple distinct instances are launched.

Unlike the previous section, which focused solely on a subset of the unikernel's virtual memory space, here we consider the entire active virtual memory allocated by Firecracker (as configured via the `mem_size_mib` field in the unikernel configuration [2]). This space encompasses not only the unikernel's code and data but also the heap, stack, and other dynamically allocated regions. All of this memory is marked as potentially shareable by applying `madvise` with the `MADV_MERGEABLE` advice [129] to the whole range managed by the hypervisor.

Software as a Service (SaaS)

For the two following experiments, we do not evaluate KSM's behavior (e.g., convergence time for complete memory reduction) or its CPU consumption, so we used the default KSM configuration [11].

First, we evaluated memory consumption in terms of the number of frames by launching various combinations of 10 different unikernels, each corresponding to a distinct application. We tested all possible combinations involving between 1 and 10 unikernels from this set and measured memory consumption after KSM's memory deduplication (each combination ran for 60 seconds). Figure 3.16 shows the average memory used for different numbers of unikernels launched simultaneously.

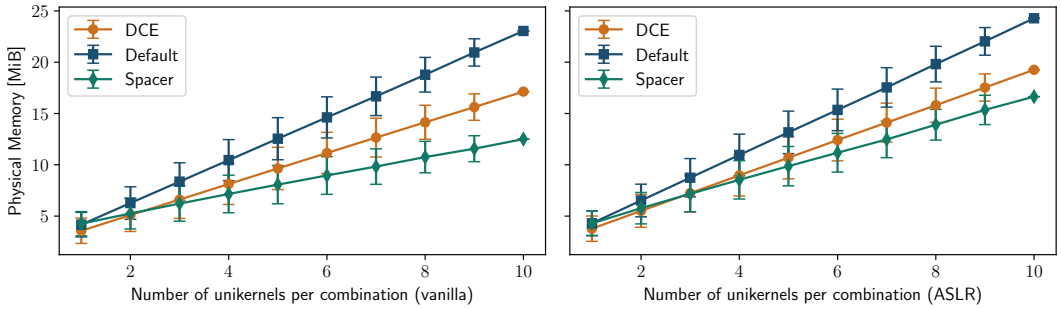


Figure 3.16: *Average memory usage (in MiB) measured for all possible combinations of 10 different unikernels. Results are shown for both vanilla and ASLR configurations.*

We first observe that memory usage is higher than in Figure 3.15, as the virtual memory address space allocated by Firecracker also includes additional components (e.g., code related to BIOS, Memory-mapped I/O (MMIO) devices, unikernel stack, etc.). Although Spacer unikernels are individually larger, memory deduplication allows them to consume less physical memory than both Default and DCE unikernels, even with as few as three applications in the vanilla version and four in the ASLR version. For instance, when running all 10 applications in parallel with the vanilla version, Spacer results in a 30% memory saving compared with DCE and a 45% saving compared with Default. This efficiency stems from the effective sharing of pages between instances, which is due to Spacer's memory alignment strategy. As the number of applications increases, so does the potential for sharing

between libraries, leading to even greater memory savings. Figure 3.17 illustrates this trend, showing the average memory consumption per unikernel as the number of distinct unikernels grows.

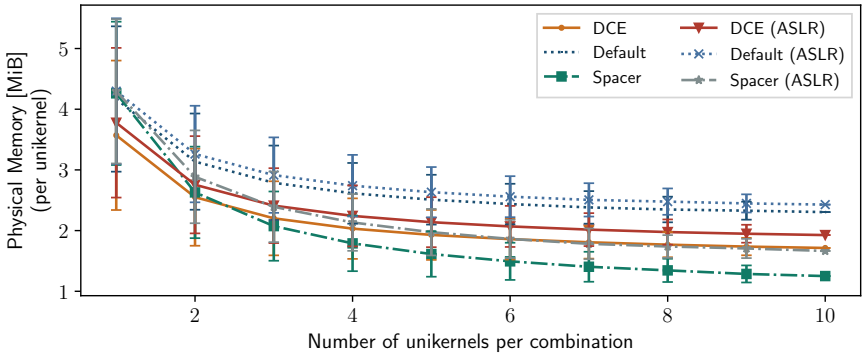


Figure 3.17: *Average memory used (in MiB) per instance for vanilla and ASLR configurations. As we run more applications, the benefits of alignment increase. Spacer and Spacer (ASLR) unikernels respectively consume less and less physical memory compared with DCE and Default.*

The same observation can be made for the ASLR versions except that Spacer (ASLR) converges more slowly because of its trampoline tables. However, it can compete with DCE (vanilla) unikernels as soon as 6 instances of different applications are running on the same server.

Function as a Service (FaaS)

After evaluating the memory consumption of various applications ported as unikernels for SaaS, we now examine the memory usage of FaaS-based unikernels. We explore two different approaches to building FaaS unikernels. First, we design a minimal unikernel that executes small and event-driven functions on demand, relying only on a libc, an HTTP server, and a network stack. This approach aims to achieve low memory consumption per instance (on average 6 MiB). Second, we develop different FaaS unikernels using Go as a programming language and relying on the Go runtime. These unikernels consume more memory per instance (at least 110 MiB) due to their reliance on the Go runtime.

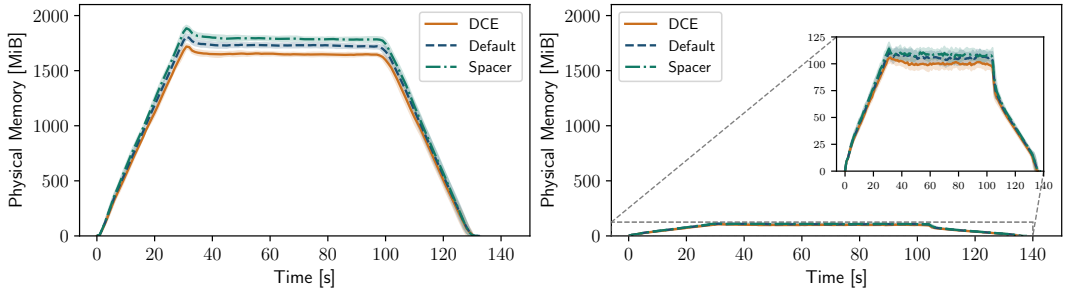
For these experiments, since we will be running significantly more instances than before, we had to consider two distinct KSM configurations, each balancing convergence time and CPU overhead differently:

1. *KSM-quiet*, the default KSM setup, scanning 100 pages every 20 ms. This configuration prioritizes low CPU usage but merges identical pages more slowly.
2. *KSM-full*, a manually tuned policy [168] optimized for memory efficiency at the cost of high CPU consumption. In this setup, KSM runs continuously, scanning 20000 pages every millisecond, fully utilizing an entire CPU core.

We first evaluated minimal FaaS unikernels under the two KSM configurations. In this setup, 1000 identical instances were launched sequentially, with a 100 ms delay between each launch. Each instance performs 2× scaling of an image, after which

an HTTP server listens on a specific port to serve the upscaled image. A timeout is applied to each instance, terminating the underlying unikernel if no requests are received within 30 seconds. This scenario serves to illustrate that memory alignment brings no deduplication benefits when all instances are identical and share the same ELF layout.

Figure 3.18 shows the memory consumption of 1000 identical unikernels (which share the same underlying ELF file) under the two KSM configurations: KSM-quiet and KSM-full. Each experiment was repeated 30 times.



(**a**) *KSM-quiet mode.* (**b**) *KSM-full mode.*

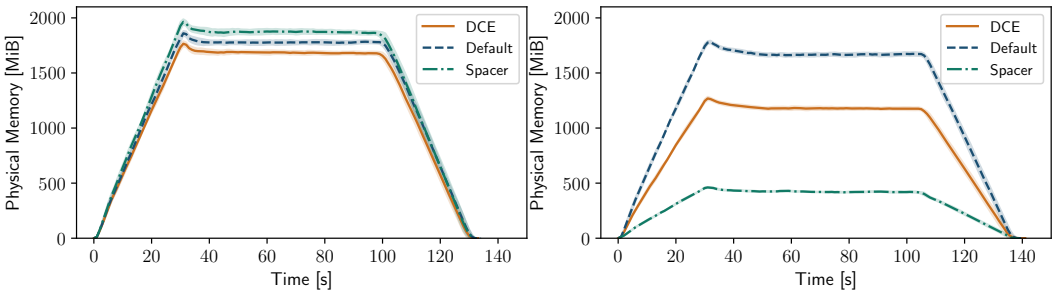
Figure 3.18: *Using Spacer with 1000 identical FaaS unikernels sharing the same underlying ELF file results in memory inflation, even when leveraging an aggressive KSM configuration.*

With the default KSM configuration illustrated in Figure 3.18a, only a few pages can be merged, as KSM scans and merges only a small fraction of the pages relative to the large number generated by all running instances. Consequently, memory reduction is limited, resulting in high overall memory usage. In contrast, the aggressive KSM configuration (Figure 3.18b) significantly reduces memory usage – by up to 16× – but at the cost of increased CPU overhead, leading to longer execution times for all instances compared with the KSM-quiet configuration.

For both configurations, aligning the unikernels with Spacer provides no real advantage, as the unikernels are inflated and have the same underlying ELF file. Since each instance is identical, they share the same pages, except for stack and heap pages, which vary during execution. KSM merges identical pages into a single frame. However, because Spacer requires additional pages, it consumes more memory than Default and DCE in both scenarios. Therefore, using Spacer vanilla on identical instances introduces a slight overhead compared with DCE and Default.

Nevertheless, combining alignment with ASLR becomes especially beneficial under aggressive page deduplication, as illustrated in Figure 3.19b. Using Spacer with ASLR-based unikernels (i.e., having different underlying ELF files) significantly reduces memory consumption compared with DCE (ASLR) and Default (ASLR) unikernels. This reduction occurs because all library code and read-only data can be shared between all instances, with only the trampoline tables remaining unshareable between instances. Consequently, more pages can be merged, allowing Spacer (ASLR) unikernels to consume, on average, up to 3× less memory than DCE (ASLR) unikernels and 4× less than Default (ASLR) unikernels under the

KSM-full configuration. This result is consistent with the SaaS-based unikernels, where Spacer (ASLR) unikernels consume less memory than DCE (ASLR) and Default (ASLR) unikernels. In contrast, the KSM-quiet configuration illustrated in Figure 3.19a results in only limited memory reduction and high overall memory consumption across all configurations.



(a) *KSM-quiet mode.* **(b)** *KSM-full mode.*

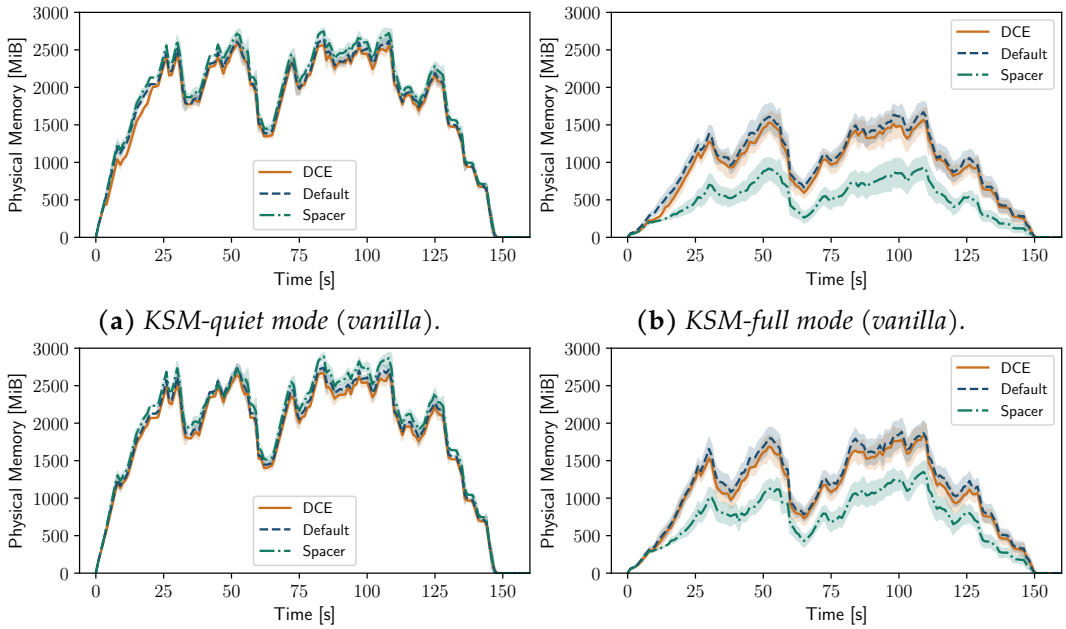
Figure 3.19: *Using Spacer with 1000 FaaS unikernels using ASLR (resulting in different ELF files) achieves lower memory usage compared with both DCE and Default configurations.*

We also analyzed the memory consumption of FaaS-based unikernels developed with Go. Since these unikernels use more memory, we limited the experiment to 128 instances, each running a different main program[2]. We designed a dynamic environment in which a new instance is launched every second, executing workloads with durations ranging from 2 to 30 seconds. Each unikernel has a different underlying ELF file, and the same KSM configurations as before were used. Figure 3.20 illustrates the memory consumption of 128 different instances (vanilla and ASLR) under the two previous KSM configurations. As before, each experiment was repeated 30 times.

As before, using KSM with the default configuration fails to showcase the full potential of Spacer, as the high number of pages combined with the limited scanning frequency prevents effective identification and merging of duplicate pages. Spacer can fully show its potential when using a more aggressive KSM configuration. With this configuration, memory consumption is significantly lower for Spacer unikernels than for DCE and Default. For the vanilla configurations illustrated in Figure 3.20b, averaged over the full observation period, Spacer uses 1.8× less memory than DCE and 2× less memory than Default. The gap is also visible at peak consumption: Spacer reaches a maximum of approximately 933 MiB, compared to 1568 MiB for DCE and 1670 MiB for Default, representing reductions of 1.7× and 1.8× respectively. When considering ASLR, illustrated in Figure 3.20d, the additional memory required for the trampoline tables in Spacer is largely compensated by increased sharing, and Spacer (ASLR) consumes on average 1.5× less memory than DCE (ASLR) and 1.6× less than Default (ASLR).

The smaller gap in the memory gain compared with Figure 3.19b is explained by the difference of unikernels. Indeed, as the application code is different in each in-

[2] We rely on GobyExample [133] and GitHub [78] as our primary resources for writing the 128 FaaS.



(a) *KSM-quiet mode (vanilla).*

(b) *KSM-full mode (vanilla).*

(c) *KSM-quiet mode (ASLR).*

(d) *KSM-full mode (ASLR).*

Figure 3.20: *With 128 different FaaS unikernels based on the Go runtime under the KSM-full configuration, Spacer unikernels consume, on average,* 1.8× *less memory than DCE unikernels. Spacer (ASLR) unikernels, on average, use* 1.5× *less memory than DCE (ASLR) unikernels. KSM-quiet is completely overwhelmed in this scenario and fails to achieve any meaningful memory deduplication.*

stance, pages containing this code are not shareable between instances, which also impacts sharing. In addition, the Go runtime also initializes different data structures to create the environment in which Go programs can run efficiently. This involves setting up dynamically allocated data structures for scheduling, memory management, and other essential runtime functions, which create memory pages outside the unikernel's static code footprint.

3.4.3 ELF File Size

We also evaluated the impact of alignment on unikernel size by analyzing the underlying ELF files of both SaaS and FaaS unikernels. Figure 3.21 represents the total ELF file size for DCE, Default, and Spacer unikernels, including their respective ASLR versions, in the SaaS-based configuration.

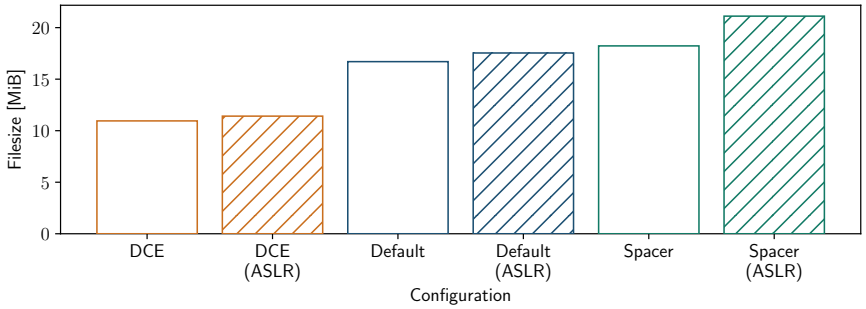


Figure 3.21: *Total size of 10 unikernel images (ELF files) according to 6 different configurations.*

Excluding ASLR, alignment does not introduce a significant size overhead since the padding required for alignment (zero pages) is added only when the unikernel is loaded into memory. As a result, Spacer unikernels are only slightly larger than Default unikernels. This minor inflation is primarily due to the section header table and the string table section (`.shstrtab`), which contain more entries (e.g., one `.text` entry per library instead of a single aggregated `.text` section). However, Spacer unikernels remain significantly larger than DCE unikernels, where large and monolithic libraries (e.g., libc, lwip, etc.) are extensively pruned.

Performing ASLR at link time results in a slight increase in size for all three configurations. This is an artifact of our ASLR implementation. Since ASLR is managed through linker scripts, `.text` sections are no longer aggregated into a single entry but instead split into multiple entries (one per library). As before, this increases the section headers table and the string table, leading to a larger binary size compared with the corresponding vanilla versions. The size increase is even more pronounced in the Spacer (ASLR) configuration, as it disaggregates not only the `.text` section but also other sections in addition to introducing trampoline tables. Having a size overhead with Spacer and Spacer (ASLR) is not a problem since storage media have a much higher capacity than RAM (which is also more expensive).

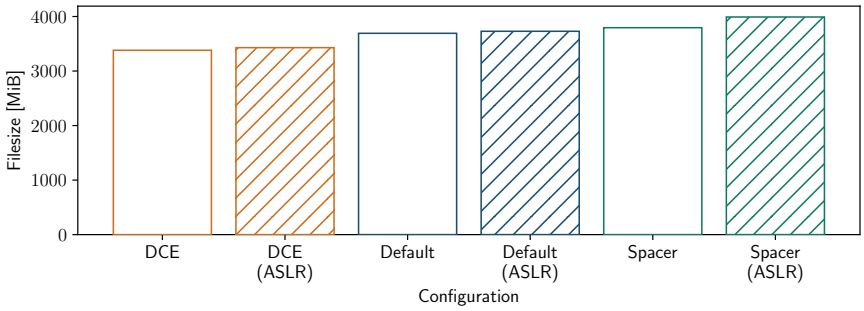


Figure 3.22: *Total size of 128 unikernel images (ELF files) according to 6 different configurations.*

We also analyzed the size of FaaS-based unikernels. Figure 3.22 shows the aggregate size of 128 unikernels based on FaaS configurations. The size-related observations align with those in Figure 3.21, where Spacer unikernels exhibit higher disk usage, while DCE versions remain smaller than other configurations.

An important highlight concerns the trade-off between size and memory consumption. Indeed, at the disk level, DCE unikernels allow a space saving of approximately 15% compared with Spacer. However, as soon as they are instantiated in memory, the trend is reversed and the Spacer versions allow limiting the memory consumption, as observed in Figure 3.20.

Beyond raw resource usage, this trade-off has important practical implications. By using Spacer, more unikernels can run concurrently on the same server, increasing consolidation density and improving hardware utilization and efficiency. In large-scale deployments or cloud environments, these benefits translate into meaningful energy savings and cost reductions, contributing to greener and more scal-

able infrastructure.

3.4.4 Performance

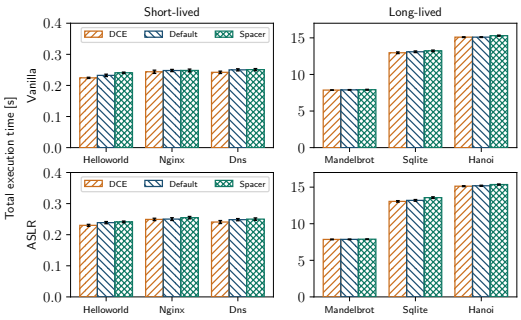
Finally, we performed different performance tests to ensure Spacer does not degrade performance compared with DCE and Default unikernels. Performance testing can be complex and multifaceted due to factors such as different memory allocators, varied metrics, and diverse workloads across applications. Therefore, we first analyzed the total execution time of the unikernels – including creation, boot, run, shutdown, and destruction – across the six configuration types. Fair benchmarking of network-based workloads proved challenging, primarily because network measurements may introduce variability unrelated to memory layout (e.g., client-side jitter or network conditions). To isolate the overhead attributable to Spacer from network noise, we instead modified the main code of some unikernels to terminate immediately after their initialization. Nginx and DNS unikernels are examples that were adapted accordingly. This controlled setup allows us to evaluate Spacer's intrinsic overhead independently of external factors; we later complement this analysis with application-level benchmarks under more realistic workloads.

Since SQLite is not server-oriented, it was relatively straightforward to benchmark. Specifically, we conducted a benchmark involving 60000 SQL insertions. In addition to SQLite, we also ported several other applications to unikernels for benchmarking purposes. One application computes `hanoi(32)`, and another generates a 1280×720 Mandelbrot image and saves it to a file. We measured the total execution time of each unikernel. This approach enables us to evaluate both short- and long-lived unikernels under diverse workloads.

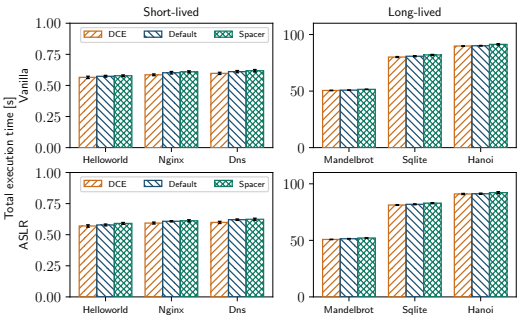
For the following experiments, we isolated a CPU core and pinned the unikernel under test to this core to accurately assess the overhead introduced by library alignment. To simulate a realistic deployment scenario, we concurrently ran idle unikernels (e.g., Nginx, HAProxy, etc.) alongside actively running unikernels (e.g., Hanoi, SQLite, etc.). Based on this setup, we defined two test cases to emulate different environments: (1) running all unikernels on the same core (the isolated one), and (2) running all unikernels on different cores.

In addition to the previously described KSM modes, we also conducted experiments with KSM disabled. This resulted in three distinct modes: (1) KSM disabled, (2) KSM-quiet, and (3) KSM-full. We performed these tests using the perf [85] tool on both vanilla and ASLR-enabled unikernels, grouping them into short-lived and long-lived categories. All the different configurations are grouped into the two test cases, which are shown in Figure 3.23 (different cores) and Figure 3.24 (same core), respectively. We use KSM disabled as a baseline and report the additional execution time when KSM is enabled. In the plots, the baseline appears in a lighter shade, while the additional execution time is shown in a darker, more opaque color to highlight the difference. Note that the y-axis for the KSM-quiet and KSM-full

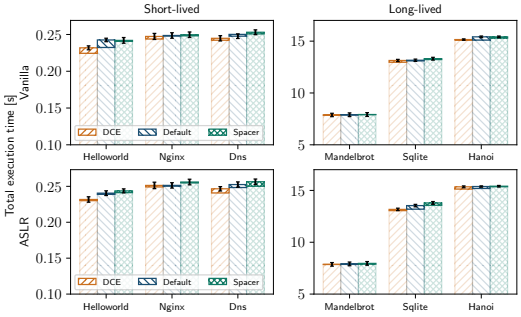
configurations is shown at different scales (zoomed views) to improve readability of the observed differences. Each experiment was repeated 30 times.



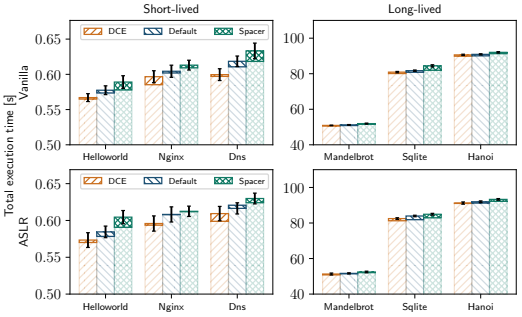
(a) *KSM is disabled (baseline).*



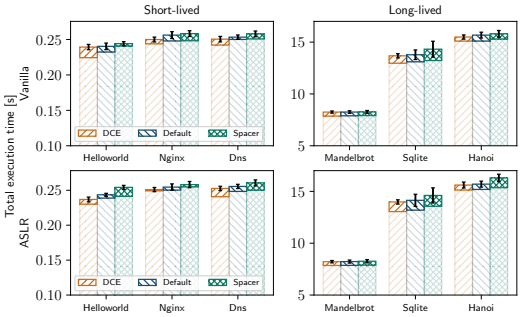
(a) *KSM is disabled (baseline).*



(b) *KSM is enabled (quiet-mode). Baseline is in lighter shade; additional execution time is in darker, opaque color.*

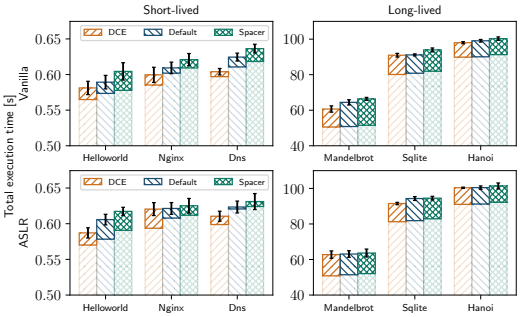


(b) *KSM is enabled (quiet-mode). Baseline is in lighter shade; additional execution time is in darker, opaque color.*



(c) *KSM is enabled (full-mode). Baseline is in lighter shade; additional execution time is in darker, opaque color.*

Figure 3.23: *Average total execution time (seconds) of the benchmarked instance (on the isolated core). All other unikernels are executed on different cores.*



(c) *KSM is enabled (full-mode). Baseline is in lighter shade; additional execution time is in darker, opaque color.*

Figure 3.24: *Average total execution time (seconds) of the benchmarked instance. All other unikernels are executed on the same core (the isolated one).*

A first general observation concerns the impact of Spacer on performance. The results suggest that the structural overhead introduced by Spacer's layout – while intended to facilitate memory deduplication – does not result in noticeable runtime performance impacts under the tested workloads. In terms of total execution time, the overhead remains modest, reaching at most 5.6% relative to DCE-based unikernels and 3.3% relative to the default configuration. By comparison, DCE-based unikernels exhibit the lowest execution times, primarily due to their reduced

size, which leads to fewer cache misses and memory pages to load and manage.

ASLR configurations also introduce a slight performance penalty compared with vanilla configurations. In ASLR-based unikernels, libraries are placed at different virtual addresses with random gaps between them, preventing compact placement. This non-contiguous layout increases the overall unikernel size and the number of memory pages that must be loaded. This, in turn, causes an increase in loading time. Furthermore, we observe Spacer ASLR introduces an additional execution overhead. This is likely due to the alignment constraints and the inclusion of trampoline tables, which again increase the memory footprint. Notably, despite the potential for page sharing due to identical content across instances, neither Spacer nor Spacer (ASLR) shows a reduction in execution time.

We also analyze the differences between short-lived and long-lived unikernels. Short-lived unikernels, which execute quickly and terminate shortly after initialization, are particularly sensitive to overhead during the loading phase. In such cases, enabling KSM – especially under the KSM-full configuration – can increase total execution time. This overhead is primarily attributable to CoW faults triggered when deduplicated pages are modified shortly after being shared. Such faults introduce latency at the point of page access, which is particularly costly for short-lived unikernels that access memory intensively during initialization and terminate before the benefits of deduplication can fully materialize. Conversely, long-lived unikernels can amortize the cost of memory scanning over time but are more likely to encounter numerous CoW faults. These faults can significantly impact performance, depending on the workload characteristics.

We then compare the two test cases: running unikernels on different cores vs. running them on the same isolated core. In both scenarios, enabling KSM – particularly under aggressive configurations, results in measurable performance degradation. This can be attributed to the increased frequency of CoW faults, as well as the overhead introduced by the memory scanning and merging process, which may involve page table locking and trigger TLB [3] shootdowns, thereby elevating memory access latency. While one might expect some benefit from improved cache locality due to colocated execution, our experiments did not reveal any significant performance gains attributable to cache hits. This suggests that, with KSM enabled, the deduplication overhead outweighs any potential advantages related to core sharing. Indeed, using KSM in full mode can lead to an increase in total execution time of up to 8% when running unikernels on different cores, and up to 28% when they are colocated on the same core. The significantly higher overhead observed in the single-core configuration is primarily due to CPU contention, which introduces more context switches and cache misses. Additionally, since multiple unikernels share the same core, the likelihood of CoW faults increases as longer lifetimes raise the probability of writes to shared pages, further amplifying the per-

[3] The Translation Lookaside Buffer (TLB) is a page table cache that accelerates the translation of virtual addresses to physical ones.

formance penalty under aggressive KSM settings.

While the execution time analysis required modifying certain unikernels to eliminate network-induced variability, we complemented this with a direct application-level benchmark to verify that Spacer does not affect runtime performance. Specifically, we evaluated an unmodified Nginx unikernel and measured throughput (MB/s) across six configurations. Experiments were conducted using the same setup as in the previous section (running all unikernels on an isolated core and on different cores). We used the wrk tool [79], configured with 14 threads and 30 connections, to request a static 612B HTML page, and repeated each experiment 30 times. Results are presented in Figure 3.25.

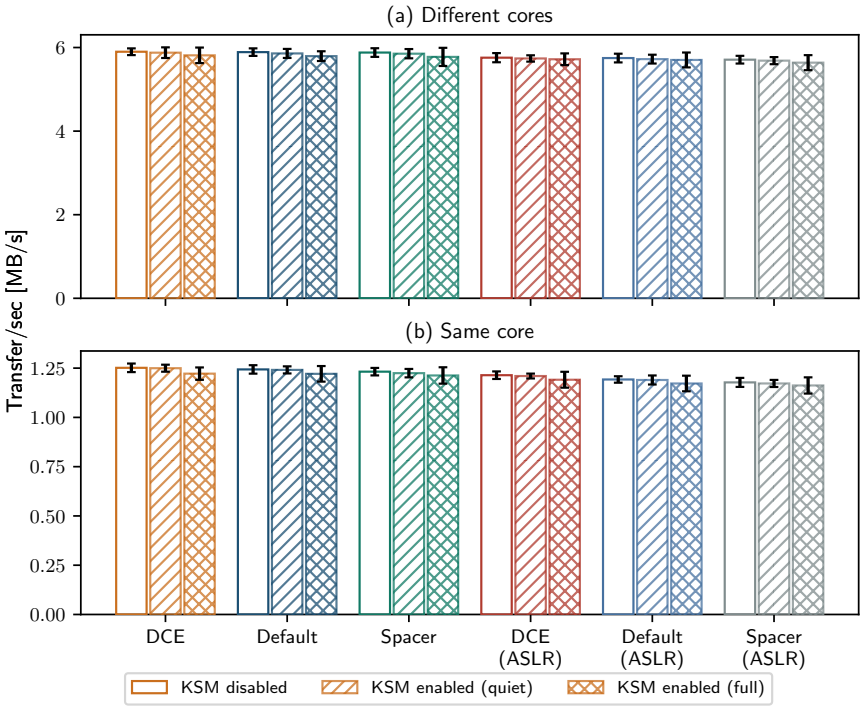


Figure 3.25: *Average throughput of Nginx with different configurations. Spacer performance degradation is minimal. KSM has a performance impact, which is mainly noticeable under the aggressive KSM configuration.*

As observed in the previous experiments, ASLR still introduces a slight throughput overhead. In particular, Spacer with ASLR incurs a noticeable performance penalty compared with both DCE and the default ASLR-enabled configuration. Specifically, Spacer exhibits a 2.1% degradation compared with DCE (ASLR) and 1.35% compared with the default (ASLR) configuration, primarily due to the use of trampoline tables. The best performance is achieved with the vanilla DCE configuration, which yields smaller binary images by eliminating unnecessary code. Consistent with earlier results, KSM introduces additional overhead, especially under the aggressive KSM configuration, where memory scanning and deduplication become more intrusive.

3.5 Discussion

This section discusses and justifies the key decisions made in our methodology and implementation.

3.5.1 For Which Applications?

Our alignment approach targets merging code and read-only data across multiple unikernels, yielding significant memory savings when many different instances run on the same machine. However, the gain is less noticeable when applications heavily utilize the heap (i.e., dynamic memory allocation), as heap allocations can dominate overall memory usage. We have observed that certain applications are less suited for Spacer, notably in-memory databases such as Redis [167] and Memcached [63]. These applications initialize large data structures on the heap, far exceeding their code size. Due to their volatility, pages containing these structures are difficult to share across instances, resulting in relatively lower memory savings.

We evaluated the memory consumption ratio between the heap, the unikernel's read-only data/code segments, and the total memory used by the unikernel. Each application was tested under load using a corresponding benchmark (e.g., wrk [79] for Nginx, redis-cli [167] for Redis, etc.) to observe how heap usage evolves during execution. Memory usage was sampled continuously during each experiment, and the reported values correspond to the time-averaged memory consumption for each application, as shown in Figure 3.26. Each experiment was repeated 30 times.

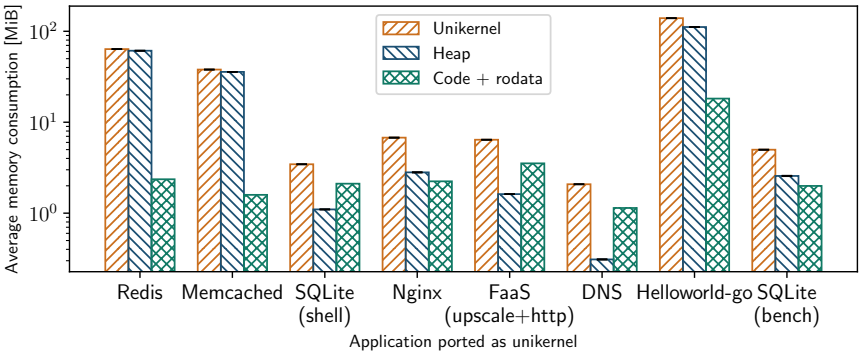


Figure 3.26: *Average memory consumption per application when they are benchmarked. Depending on the type of application, the heap can strongly dominate the memory consumption.*

As we can observe, the code and read-only data account for approximately 3% and 4% of the total memory usage in Redis and Memcached, respectively. This clearly indicates that the heap is the dominant contributor to memory consumption in these applications. Nginx was tested with 30 simultaneous connections remaining open for one minute. Therefore, the heap usage is higher as Nginx was evaluated during the benchmarked period. Once the connections are closed, Nginx uses less memory for the heap than for the code. Even a minimal Go program like Helloworld-go includes the full Go runtime, which can dominate the heap.

However, when running many instances on the same machine (as discussed in Section 3.4.2), the large underlying codebase can benefit from memory deduplication, leading to reduced overall memory consumption when using Spacer. For other applications, since they do not allocate much data, their code and read-only data dominate the heap.

Even if deduplication seems less suitable for heap-intensive applications, it can still be interesting since their code and read-only data will still be shared. Another aspect to be considered is whether the applications are short-lived or long-lived. Indeed, if an instance does not live long enough for KSM to reclaim its shared pages, Spacer currently provides little benefit. Finally, depending on its needs and priorities, the cloud operator may choose to use the ASLR-enabled version of Spacer for increased security, at the cost of increased memory usage.

3.5.2 Coping With Changes

Spacer builds a global mapping of libraries across all unikernels that can potentially run on the system. However, the set of deployed unikernels may evolve over time: some applications may be removed, while new ones may be introduced. Even when the applications and libraries remain the same, code can still evolve due to library updates, compiler changes, or modifications in compiler settings.

If a new mapping is built for a different set of unikernels, the running unikernels will continue using the old mapping, potentially reducing the level of sharing. To mitigate this, it is preferable to restart as many applications as possible (which is an inexpensive operation thanks to the fast instantiation times of unikernels). Applications that cannot be restarted will continue running, and over time, as older applications are replaced with newer ones, the level of sharing will naturally increase.

3.5.3 Read-only Pages

Memory deduplication is vulnerable to memory disclosure attacks [231, 189, 32, 22], which exploit differences in write access times on deduplicated pages recreated by CoW. Our approach considers both read-only and read-write pages, as this is the default behavior of KSM and other memory deduplication scanners (e.g., UKSM). However, the KSM daemon can be modified to process only read-only pages. Although this would slightly reduce memory savings (since data, heap and other read-write pages would no longer be shareable), it would have minimal impact on our approach, as most sharing already occurs on read-only pages (`.text` and `.rodata`). Changes can also be performed directly at the hypervisor level by calling the system call `madvise` only on read-only sections/segments. This approach is easier and more flexible to implement than modifying KSM.

3.5.4 Spacer vs. Position Independent Code (PIC)

While PIC is commonly used to support ASLR by avoiding absolute addresses, it is insufficient for enabling efficient memory deduplication in unikernel environments. Although PIC allows a binary to be relocated at load time, it does not guarantee page-level content identity across unikernels with different library compositions.

Applying PIC (e.g., via the `-static-pie` option in GCC [217]) in statically linked unikernels links the application and all its libraries into a monolithic binary. Executing this binary merely shifts the entire image to a different virtual base address at each execution. Instruction encodings for control-flow transfers, such as `call` instructions, depend on relative displacements that vary with layout and library composition, resulting in instruction-level differences across instances.

Spacer overcomes this limitation by explicitly controlling code layout. By enforcing fixed virtual addresses for library code and redirecting divergent instructions through trampoline tables, Spacer enables page-level sharing while preserving ASLR. Although PIC is attractive from a security perspective – since it enables ASLR to be managed at runtime rather than at link time – it is detrimental to memory sharing.

3.5.5 Using Another Deduplication Scanner

Enhanced memory scanners such as UKSM [230] extend the capabilities of KSM by prioritizing memory regions to accelerate the deduplication process. Although UKSM demonstrates improvements over standard KSM configurations, it requires custom patches for Linux kernel versions 4 and 5, and it is no longer actively maintained by its original developers [235]. These limitations raise concerns about its long-term viability in production systems. Consequently, we chose to use KSM, which is natively integrated into the Linux kernel and is widely supported across kernel versions.

UKSM aims to balance the approaches of KSM-quiet and KSM-full configurations. While it merges more memory than KSM-quiet, it does not achieve the aggressive memory deduplication of KSM-full. This trade-off reflects UKSM's goal of optimizing memory deduplication efficiency without significantly compromising performance. Importantly, Spacer remains agnostic to the underlying memory deduplication mechanism and is compatible with both KSM and UKSM, allowing it to benefit from enhancements in either without requiring modification.

3.6 Related Work

In terms of related work, we can mention serverless computing areas [23], but few of them use unikernels as underlying sandboxes. Indeed, several studies have been conducted on reducing boot latency and memory usage of serverless technologies

and especially FaaS [179, 220, 178]. Nevertheless, they all relied on heavyweight VMs or less-isolated containers.

With the advent of unikernels, new techniques rely on them since they offer strong isolation and even better performance. For instance, SOCK [146] considerably improves FaaS performance by using a cache of libraries and by loading them when necessary. SEUSS [39] and SAND [4] are similar approaches which allow for faster start times and improved cacheability by using CoW sharing across parallel executions and lightweight isolation procedures. Both use unikernels underneath as they have proven effective against various threat models [59, 187, 84]. Most of these previous techniques minimize start-up time, but reducing memory consumption is not their primary goal.

UaaF [192] (Unikernel as a Function) abstracts applications as a combination of different functions, and each function is built as a unikernel where the function is linked with an underlying library operating system. Applications in UaaF are thus isolated in unikernels and shared for multiple user sessions at the same time. In order to accomplish that, UaaF exploits a hardware instruction (`VMFUNC`) provided by Intel CPUs to support code sharing among different unikernels, without suffering the performance penalty of VM-exits. UaaF is close to our approach. In UaaF, applications are abstracted as unikernels, with libraries shared across instances. This approach does not rely on hardware-level mechanisms. While it offers benefits in terms of memory efficiency and reduced startup latency, it also introduces potential security risks, as it enables unrestricted memory sharing between unikernels. Indeed, if a malicious unikernel has access to another unikernel's memory, heap and stack may be altered which is a significant vector of attack [136, 99]. In contrast, we rely on a memory deduplication scanner and, if needed, only read-only sections can be shared between instances which prevents such attacks. In addition, our approach supports ASLR, different library combinations and is compared with DCE, unlike UaaF.

3.7 Summary

This chapter introduces an analysis of memory deduplication challenges in unikernels when deployed in cloud-computing environments. While unikernels are well-known for their small size and strong performance metrics – such as ultra-fast boot times – they offer limited opportunities for page sharing. Their specialized and custom-built nature results in varying library configurations across instances, reducing the number of unikernels that can be efficiently deployed on a single server.

To address this limitation, we designed a novel methodology based on page alignment. This approach automatically restructures unikernel memory layouts by aligning libraries on page boundaries, thereby increasing the number of shared pages between unikernels. From this methodology, we developed Spacer and its ASLR-enabled variant, two proof-of-concept prototypes that demonstrate the feasi-

bility of this technique.

Although introducing fragmentation within virtual memory may seem counter-intuitive, our results show that library alignment can reduce memory consumption by up to 3×, compared to unikernels optimized with DCE. Moreover, this alignment strategy incurs minimal overhead in terms of binary size and does not degrade application performance significantly, making it a viable optimization for enhancing unikernel efficiency in cloud environments.

3.8 Potential Improvements

Improving the Spacer ASLR approach. To enable efficient memory deduplication, we isolate problematic instructions using trampoline tables and a binary rewriting technique. This approach has the advantage of being compiler-agnostic and straightforward to implement. Nevertheless, binary rewriting introduces a slight time overhead. This overhead could be avoided by changing the compiler (more specifically, the linker) behavior to create and populate trampoline tables during the linking stage. In addition, this approach would allow the use of indirect calls and jumps (difficult to achieve with binary rewriting due to instruction size) which could decrease the size of the trampoline tables. Performing this at link time also ensures the technique is portable across different architectures, albeit at the cost of requiring a specifically modified compiler/linker.

Adapting Spacer to other frameworks. A natural direction for future work is extending Spacer beyond the Unikraft codebase to support a broader range of unikernel ecosystems. Although Spacer was designed around the abstractions and build model of Unikraft (relying on linker scripts and ELF files), many of its core ideas could be applied to other unikernel architectures, including OSv [101], Nanos [198] and MirageOS [127]. However, adapting Spacer to these ecosystems would require addressing differences in compilation and linking models. Since Nanos and OSv expose a POSIX-compatible interface and support a wide range of unmodified Linux ELF binaries, porting Spacer to these systems should be straightforward, as they retain ELF-based linking models consistent with Spacer's current assumptions. In contrast, language-specific ecosystems such as MirageOS rely on whole-program compilation and higher-level runtime abstractions that depart from the assumptions underlying Spacer's binary analyzer, making such a port more challenging. A generalized version of Spacer could therefore evolve into a modular framework with pluggable backends for different unikernel toolchains and execution models.

This page intentionally left blank.

4

LOAD TIME MEMORY DEDUPLICATION

Overview

In this chapter, we extend our exploration of unikernels and memory deduplication by introducing a new approach to achieve more efficient memory usage and better performance. In the previous chapter, Spacer addressed the challenge of unikernel specialization by aligning libraries at fixed, page-aligned addresses across instances, significantly improving memory sharing. Nevertheless, to achieve memory savings, Spacer depends on runtime memory deduplication scanners. These scanners struggle with convergence delays, high CPU usage, and unpredictable memory savings.
To overcome these limitations, we introduce Spacer-SLT (Share at Load Time), an extension of the Spacer framework that enables deterministic memory sharing during unikernel loading through the use of a custom loader and a common library pool. Spacer-SLT assigns libraries to fixed memory addresses and uses a Firecracker-based loader to map identical pages across instances without depending on runtime deduplication. It maintains the simplicity and performance benefits of static linking while avoiding the complexity, overhead, and security risks associated with dynamic libraries. Additionally, Spacer-SLT supports ASLR for enhanced security and achieves superior memory efficiency in both homogeneous and heterogeneous unikernel deployments.

4.1 Introduction

While specialization is a core strength of unikernels, it also introduces challenges for memory efficiency at scale [192, 73]. Despite sharing common libraries, each instance embeds them uniquely, resulting in distinct binaries and limiting opportunities for memory deduplication in cloud environments. As a result, large-scale

deployments can suffer from unexpectedly high memory usage, even when using deduplication mechanisms like Kernel Samepage Merging (KSM) [11]. Spacer, introduced in the previous Chapter, addresses this by aligning libraries at fixed addresses, significantly improving deduplication efficiency with Unikraft [105]. However, it still relies on runtime deduplication, which presents trade-offs: quiet modes miss sharing opportunities, aggressive ones add CPU overhead [230, 73], and both can struggle under bursty workloads. Moreover, deduplication mechanisms based on CoW semantics have been shown to expose timing side channels [188, 151, 189, 231, 232, 87, 32, 5], undermining isolation guarantees in cloud environments.

Another alternative is to adopt a dynamically linked model, where multiple unikernels share dynamic libraries at runtime. While this approach can significantly reduce memory usage, it compromises the immutability and simplicity that define unikernels. It also introduces runtime overhead and added complexity. This ultimately undermines key design goals such as performance and security [118, 105].

These limitations highlight the need for a better approach that preserves the core principles of unikernels while still enabling isolation and efficient memory reuse. To this end, we introduce Spacer-SLT (Share at Load Time): a novel mechanism that enables deterministic memory sharing by mapping identical pages from different unikernel instances to the same memory frames at load time. Unlike prior solutions, Spacer-SLT ensures immediate memory sharing, eliminating the CPU cost and timing convergence trade-offs of deduplication scanners. It also maintains a fully static and immutable build model, avoiding the issues associated with dynamic libraries, such as symbol resolution [55], greater attack surface, and challenges with portability and deployment complexity [53, 46]. Spacer-SLT is implemented via a dedicated toolchain and a custom loader based on Firecracker [2]. Spacer-SLT also supports ASLR for enhanced security, while still preserving deduplication efficiency. We evaluate Spacer-SLT across multiple configurations in both homogeneous and heterogeneous deployment scenarios. Results show that it achieves faster, more predictable, and lower-overhead memory reduction than both runtime deduplication and dynamic linking approaches in serverless-like workloads.

Our main contributions are as follows: (1) we extend Spacer with a toolchain and a custom loader based on Firecracker, enabling efficient memory deduplication at load time, avoiding the slower and/or more CPU-intensive runtime process. (2) we present a comprehensive performance evaluation that highlights the limitations of conventional methods for loading unikernels and/or using KSM, compared with our proposed approach. (3) We discuss the limitations of Spacer-SLT and identify some possible areas for improvement.

4.2 Problem Statement

Aligning libraries increases the number of identical pages, facilitating identification and merging by a runtime memory deduplication daemon like KSM. This results in notable memory savings when multiple instances sharing common libraries are active on the same machine. However, this method relies on a background memory scanner, which introduces inherent issues due to the scanning and merging process that we aim to eliminate with our approach. The main limitations of KSM are:

§1. *KSM convergence time*: KSM offers memory deduplication, but its effectiveness with ephemeral unikernels presents a double-edged sword. While a conservative strategy reduces CPU usage, it takes longer to identify duplicates. This significantly hinders KSM's ability to merge memory before short-lived unikernels terminate, leading to missed opportunities for memory savings. An aggressive KSM configuration is not a perfect solution either. While it speeds up deduplication, the rapid arrival of many unikernels can overwhelm KSM. This can still lead to missed sharing opportunities, limiting memory benefits.

§2. *KSM performance (CPU)*: An aggressive KSM mode significantly reduces the time to merge memory but comes at the cost of higher CPU usage. KSM uses more CPU cycles to scan and merge identical pages, which negatively affects overall system performance.

§3. *KSM variability and load sensitivity*: KSM's deduplication effectiveness is not consistent. Memory reduction depends heavily on the system load and timing of instance launches. Under low load, KSM can achieve substantial savings; however, as concurrency increases, its merging capability deteriorates, resulting in higher memory usage. This non-determinism can make resource provisioning unpredictable, particularly in multi-tenant environments.

§4. *KSM performance (I/O and TLB)*: Before being merged by KSM into a single read-only (or copy-on-write) frame, identical pages are first allocated and then mapped to different physical frames. This requires unnecessary I/O and copies to allocate and fill frames. In addition, the merging process locks the page tables [47] and issues TLB shootdowns (TLB flushes), which can increase memory access latency and degrade system performance.

§5. *KSM writable pages*: Relying on memory deduplication for writable pages is vulnerable to memory disclosure attacks [188, 151, 189, 232, 87, 32, 5], which are based on a difference in write access time on the deduplicated memory pages that are recreated by CoW. It is possible to modify this behavior, but it requires either modifying the underlying memory scanner or the VMM (i.e., the `madvise()` may be applied only to read-only pages for KSM [129]).

§6. *Spacer size (ELF inflation)*: Although alignment does not significantly increase image size, Spacer misses the opportunity to efficiently share libraries among

multiple unikernels on disk. Libraries used by several unikernels are replicated on disk –once per unikernel image (ELF file) – rather than being shared through disk-level deduplication. This limitation is independent of KSM and instead stems from the current design of Spacer.

To address these issues, two alternative approaches to KSM for implementing load time memory deduplication can be considered:

1. **Virtual Machine Manager (VMM) load time memory deduplication**: This approach is implemented in user space within the VMM and operates independently of the guest operating system. It reduces memory consumption across unikernels by utilizing shared memory from a pool of common libraries to merge identical pages at load time, eliminating the need for a dedicated memory scanner daemon. This approach requires processing unikernels offline by extracting libraries into a shared pool before deployment.
2. **Daemonless kernel load time memory deduplication**: This approach is implemented in kernel space as a replacement for traditional KSM, addressing its limitations. Unlike KSM, which relies on a dedicated kernel daemon, this approach could merge identical pages immediately upon calling the `madvise` system call, eliminating the need for a separate background process.

We opted for the VMM-based memory deduplication approach for its flexibility in managing shared memory across unikernels without requiring modifications to the guest OS. By leveraging shared memory mappings, this method enables seamless page sharing while preserving compatibility and integration with existing environments. In contrast, the kernel-based approach requires modifications to the underlying kernel, introducing additional complexity and limitation of adoption. We discuss and compare both approaches in Section 4.5.3.

4.3 Architecture

This section outlines the architecture we developed to address the challenges highlighted in the previous section. We employ the term "libraries" to encompass both internal (e.g., lib-scheduler, lib-boot, etc.) and external libraries (e.g., OpenSSL [149], newlib [162], Protobuf [81], etc.), as well as applications (e.g., SQLite [6], Nginx [163], etc.).

Our architecture was developed with the Spacer Aligner tool, described in the previous Chapter (see Section 3.3) as its foundation, upon which additional components were integrated. The Spacer tool aligns unikernels using a global map to position libraries at fixed absolute addresses. It also supports ASLR through the use of trampoline tables[1], enhancing memory sharing across multiple concurrent ASLR instances.

[1] Tables that contain problematic instructions (e.g., those using addresses from other sections or libraries). There is one trampoline table per library.

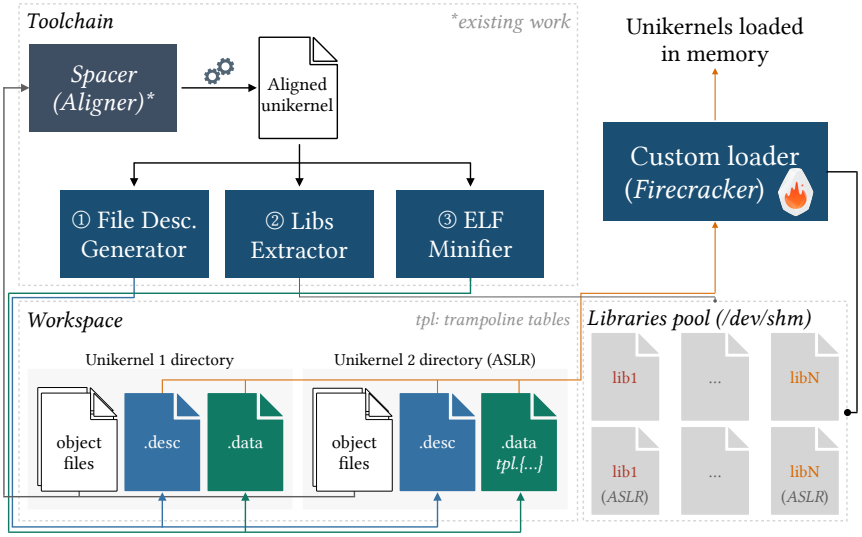


Figure 4.1: *High-level architecture: Using the Spacer aligner tool, our toolchain extracts libraries from unikernels into a shared pool (default: `/dev/shm`). A description file and app/library data enable the custom loader to run multiple unikernels by sharing common libraries.*

The high-level architecture of our solution is depicted in Figure 4.1. Four major elements can be discerned in this diagram: a toolchain, a unikernel workspace, a pool of libraries and, finally, a custom loader (based on Firecracker [2]). As for Spacer, we have based our implementation on Unikraft [105] since it supports a wide range of libraries, is open source, and is under active development. Nevertheless, the complete toolset should work with minor changes for other library OSes and unikernels. In addition, the toolchain and the custom loader can handle both regular and ASLR Spacer unikernels.

4.3.1 Toolchain

The toolchain contains the essential tools for preparing unikernels, ensuring compatibility with our custom loader. The development of the toolchain involved integrating three new tools in addition to the Spacer tool (*aligner*). This one generates aligned unikernels by using object files and custom linker scripts to place libraries at predefined memory addresses. Each tool can be executed *offline* (e.g., in a Continuous Integration/Continuous Delivery (CI/CD) pipeline). With our added contributions, the three distinct components are:

① *File Description Generator tool*: After aligning unikernels with the Spacer tool, a binary format description file (*.desc*) is generated for each unikernel by analyzing its underlying ELF file. This file includes a comprehensive list of libraries used by the unikernel, along with additional metadata such as whether the unikernel uses ASLR, and the start address and size of its data section. Each library entry contains the library name, its corresponding size, and its start address. We opted for the binary format to improve performance, but it can be easily converted to other formats such as YAML or JSON.

② *Libs Extractor tool*: As all unikernels are aligned, the code (`.text`) and the read-only data (`.rodata`) of different copies of a library are therefore identical. To avoid inflating unikernel size by including each library in every instance, this tool extracts (compiled) libraries as binary files into a dedicated library pool. We will discuss this pool further in Section 4.3.3.

③ *ELF Minifier tool*: Once all libraries are extracted, the underlying ELF file is stripped down to include only the aggregated data section (`.data`) of each library. All other sections and the ELF header itself are removed. This results in a minimal raw binary file that can be efficiently loaded directly into memory by the VMM. Note that if ASLR is used, the minimal binary file is slightly more complex and may contain several sections as illustrated in Figure 4.2.

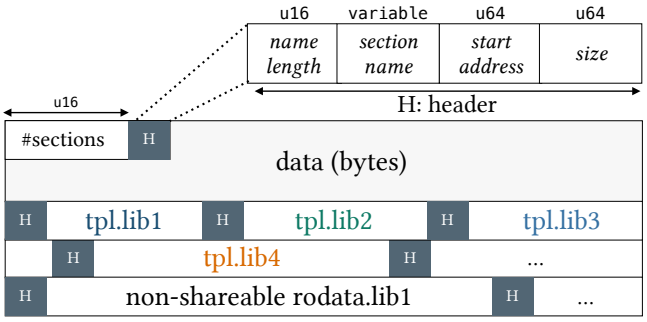


Figure 4.2: *Representation of a minimal raw binary file used by the custom loader (ASLR only). In vanilla, only the data section (without its header) is included.*

When using ASLR, a field representing the number of non-shareable sections is added at the beginning of the file. The file also contains the trampoline table (e.g., *tpl.lib{1-4}*) of each library. In this case, a header is added to the binary file, containing the library name (along with its length), the memory start address, and the size of each trampoline table. In ASLR-enabled configurations, certain libraries (e.g., in `newlibc` or `lwIP`) exhibit non-shareable `.rodata` sections. This occurs when read-only data includes link-time–resolved addresses, such as a global `const char*` that points to a symbol defined in another library. In such cases, these addresses depend on the relative arrangement of libraries in memory, and thus the contents of `.rodata` may vary across different instances. These non-shareable sections are added to this minimal file instead of being extracted to the library pool. Note that these three tools can be executed in any order. However, a unikernel must be aligned by the Spacer tool before it can be processed by the rest of the toolchain.

4.3.2 Unikernels Workspace

This is the root directory containing a set of unikernels. Each unikernel is stored in its own folder. Before being processed by the toolchain, a folder contains the object files for a specific unikernel. Once the unikernel is processed by the toolchain, its directory contains a description file named *.desc* (specifying the libraries required

by the unikernel) and a binary file named *.data*, which combines data, trampoline tables, and non-shareable read-only data (the latter two used only for ASLR) into a single and compact file that replaces the traditional ELF file.

4.3.3 Library Pool

Library code and their associated read-only data are extracted into an external library pool. By default, the *Libs Extractor tool* places these libraries in `/dev/shm`, which resides entirely in memory, providing significantly faster access compared with traditional disk-based storage. This approach not only accelerates boot times but also facilitates efficient sharing of libraries (in read-only mode) across multiple unikernels using a combination of `shm_open` and `mmap()`, as illustrated in Figure 4.3.

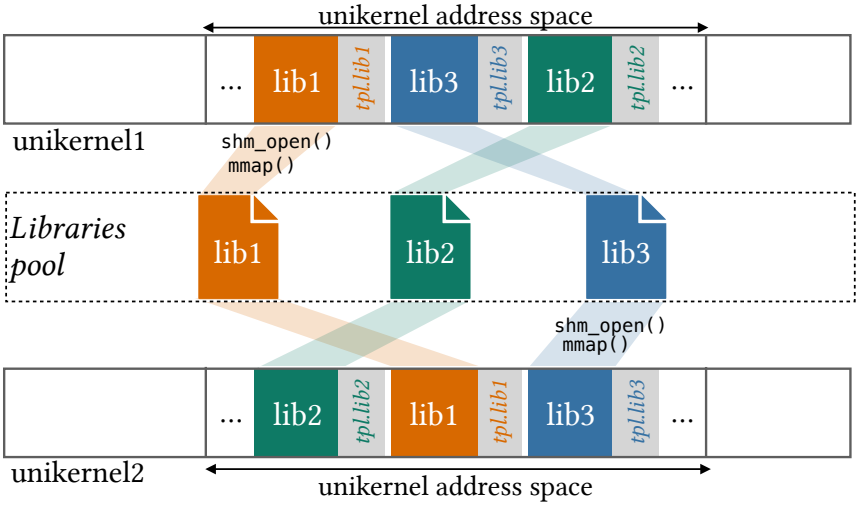


Figure 4.3: *Aligned libraries are shared across unikernels via `shm_open()` and `mmap()` by the VMM, while trampoline tables remain unikernel-specific when using ASLR.*

Each library contains two binary raw files: a vanilla version and an ASLR version (prefixed with the keyword `aslr`). This separation is necessary because the library code differs between the two versions. In the ASLR version, problematic instructions of each library are isolated in trampoline tables, unlike the vanilla version.

4.3.4 Managing the Library Pool

By default, all libraries are placed in `/dev/shm` by the *Libs extractor tool*, based on the assumption that a unikernel will be deployed soon after being processed by the toolchain. Numerous essential libraries (e.g., ukboot, uksched, etc.) are required by various unikernel instances and are therefore preloaded into memory to reduce startup latency. This preloading provides a clear gain in startup performance, leveraging additional memory for efficient startup. The library placement behavior can be modified: the tool allows placing all libraries of a specific unikernel in its current workspace on disk instead (see Section 4.4.4). Regarding runtime management, the

custom loader can be easily modified to monitor library usage and remove libraries from the memory pool once they become unnecessary, such as upon a unikernel's exit or through periodic cleanup (e.g., a cron job). This approach provides flexibility, allowing the system to trade off memory footprint against startup performance according to deployment requirements.

4.3.5 Custom Loader (Firecracker VMM)

We developed a custom loader that leverages this new unikernel representation to perform load time deduplication by directly mapping library pages onto a shared frame image in main memory. Rather than developing a new loader from scratch, we modified the Firecracker VMM [2]. We chose Firecracker instead of Quick EMUlator (QEMU) [29] because Firecracker has a much smaller codebase and delivers better performance [2]. However, our implementation could be easily ported to other VMMs.

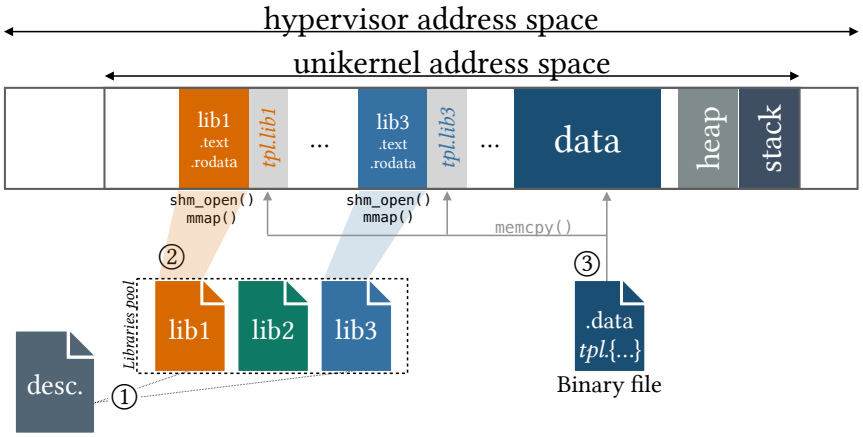


Figure 4.4: *Our Firecracker-based loader parses a library description file (①), maps libraries to shared memory (②), and places non-shareable sections (e.g., data, trampolines) into the unikernel address space from a minimal binary (③).*

Traditionally, the VMM allocates virtual memory for the unikernel using anonymous memory mapping. The size is determined beforehand, either specified in a configuration file or passed as a program argument. After being parsed (ELF parsing), the unikernel segments are loaded into this memory area and the unikernel is started by jumping to its start address. In our implementation, depicted in Figure 4.4, we depart from the conventional approach of providing an ELF file for the vanilla version of Firecracker. Instead, we supply a description file (containing a list of libraries along with their names, start addresses and sizes), alongside a minimal binary file containing library and application data. Subsequently, our loader parses the description file and iterates over the library list (step ①), mapping each library individually to shared memory via a combination of `shm_open()` and `mmap()` (step ②). In this case, the libraries are placed in shared memory segments, unlike the vanilla approach, where they were placed in anonymous memory. Moreover, com-

plete parsing of the ELF file as well as the full unikernel extraction is no longer necessary since the libraries are already in memory. Only the data section and the trampoline tables are parsed and copied into the unikernel address space using `memcpy()` (step ③). Furthermore, our approach facilitates the management of unikernels using ASLR. Specifically, only the code and relocatable read-only data are mapped to shared memory segments, while trampoline tables and non-shareable read-only data are handled similarly to the data section. We implemented our custom loader on top of Firecracker v1.1.0; the corresponding patch is available on GitHub [72].

4.4 Evaluation

This section details the experiments conducted by comparing various unikernels and their configurations. Most of the following experiments involved at least four distinct configurations, including: (1) Default, which is the basic configuration used in Unikraft; (2) DCE, which involves activating Dead Code Elimination to minimize the size of unikernels; (3) Spacer, our method that enhances memory sharing by using library alignment; (4) Spacer-SLT (*Share at Load Time*), our custom loader based on Spacer. Additionally, we present the ASLR variants for certain experiments. To implement ASLR, we used the same approach as in the previous Chapter (see Section 3.3.1), randomizing the memory layout at link time via linker scripts.

Our evaluations covered short-lived and long-lived unikernels, deployed in heterogeneous (different unikernel applications such as web servers, databases, proxy servers, etc.) and homogeneous environments (several instances of the same unikernel). We aimed to simulate SaaS and FaaS environments by using unikernels with varying lifetimes and workloads. A comprehensive list of applications, including their functionalities and sources, can be found in Appendix A.2.

We sought to answer the following questions through our evaluation: (1) In Section 4.4.1: How effective is Spacer-SLT's deduplication compared with the conventional method of loading unikernels and relying on KSM to merge identical memory pages? (2) In Section 4.4.2: How consistent is Spacer-SLT's memory reduction compared with KSM under varying system load? (3) In Sections 4.4.3 and 4.4.4: What is the impact of managing libraries through our library pool mechanism? To answer this, we evaluated overall memory usage (unikernel + VMM), disk usage, and key performance metrics including page faults, TLB flushes, load time, and total execution time (covering creation, boot, run, shutdown, and destruction of the unikernel). We also measured the time required to load a library from disk when it is not cached. (4) In Section 4.4.5: What is the impact of running multiple ASLR-based unikernels? (5) In Section 4.4.6: What are the benefits of Spacer-SLT compared with a dynamically linked approach?

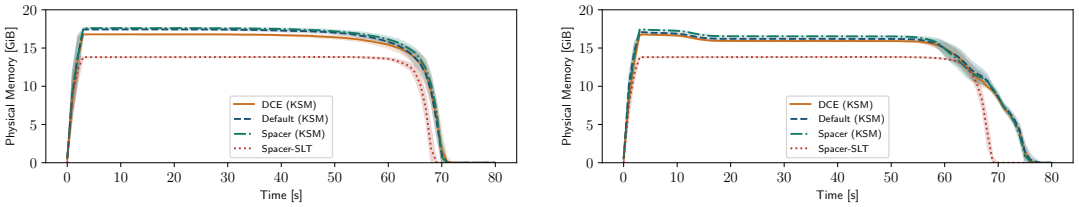
All experiments of this chapter were conducted on a Debian 11 (bullseye) server using a Linux kernel 6.1, gcc 10.2.1 (GNU ld 2.35.2 as linker). The host machine used for the experiments has 32 GB of RAM and an Intel® Xeon® CPU (E5-2650 v2

@ 2.60 GHz) with 32 cores. In addition, Unikraft Enceladus 0.8.0 (with Firecracker support [209]) has been used for the experiments.

4.4.1 Memory Reduction: Spacer-SLT vs. KSM

This section evaluates the memory reduction gain offered by Spacer-SLT compared with KSM tuned according to 2 modes. In this experiment, a homogeneous environment was selected to focus on deduplication performance. Each KSM mode offers different trade-offs between convergence time and CPU overhead: (i) KSM-quiet, the default configuration, scans 100 pages every 20 ms and is CPU-efficient, but converges more slowly when merging identical pages; and (ii) KSM-full, a manually configured policy [168], achieves higher memory efficiency for our workload at the cost of substantial CPU consumption. For this setup, KSM is set to scan 20000 pages every millisecond. This setting uses the maximum values that continuously monopolize a full CPU core for KSM. We applied these two modes to a specific unikernel under the Default, DCE, and Spacer configurations to demonstrate the advantages of Spacer-SLT.

The benchmarked unikernels use the Go runtime, which is widely adopted in FaaS environments, and are designed to emulate serverless functions for FaaS deployments. To reproduce a bursty, high-density workload scenario, 128 identical instances were launched in rapid succession. Although typical FaaS functions often execute for less than one second [178], each instance in this experiment performs approximately 10 seconds of computation to allow KSM sufficient time to converge and to avoid overstating its limitations for short-lived workloads. During the experiment, both memory usage and total execution time were measured, and the experiment was repeated 30 times. Figure 4.5 reports the average memory usage according to two KSM modes.



(a) *KSM-quiet: KSM takes time to identify and merge duplicate pages, resulting in minimal memory savings.*

(b) *KSM-full: KSM achieves better memory reduction by using more CPU cycles, but does not scale with rapid unikernel arrivals.*

Figure 4.5: *Evolution of the average physical memory used by 128 identical FaaS unikernels (+VMM) under two modes: (a) KSM-quiet and (b) KSM-full. Spacer-SLT achieves the lowest memory consumption in both cases without performance penalties.*

Figure 4.5a illustrates the performance of KSM under the KSM-quiet mode. In this setting, KSM fails to achieve substantial page merging within the duration of the experiment. DCE (KSM), Default (KSM), and Spacer (KSM) all remain close to their peak memory usage of 16.8, 17.6, and 17.8 GiB respectively throughout the

observation period, with no meaningful reduction visible. Spacer-SLT, by contrast, immediately reaches a stable footprint of approximately 13.8 GiB, without relying on any background scanning. At plateau, Spacer-SLT uses 1.2× less memory than DCE (KSM), and 1.3× less memory than both Default (KSM) and Spacer (KSM) under the KSM-quiet configuration. Differences in execution time between configurations are negligible in this mode.

Because KSM-quiet yielded only limited memory footprint reduction, we further evaluated a more aggressive configuration (KSM-full), shown in Figure 4.5b. KSM-full achieved greater overall memory reduction than KSM-quiet; however, the rapid arrival of many unikernels caused the scanner to process the memory of all instances without merging pages efficiently, resulting in missed sharing opportunities. Spacer-SLT again remained the best-performing approach, consuming approximately 1.15× less memory than DCE (KSM), 1.2× less memory than both Default (KSM) and Spacer (KSM) during the plateau phase. However, this improvement in memory usage came at a cost. We observed a slight increase in execution time for all KSM-based variants, taking 10% longer compared with KSM-quiet.

To further evaluate the efficiency of our approach, we measured the average physical memory usage per instance for the 128 FaaS unikernels from the previous experiment. The results, shown in Figure 4.6, highlight Spacer-SLT's ability to maintain consistent memory usage across all instances. In contrast, KSM-based configurations exhibit less stable behavior. With the KSM-full setting, the first few unikernels benefit from notable memory deduplication, as there are fewer pages to scan and merge. However, as more unikernels are launched in rapid succession, KSM becomes overwhelmed and is unable to keep up with the growing memory footprint.

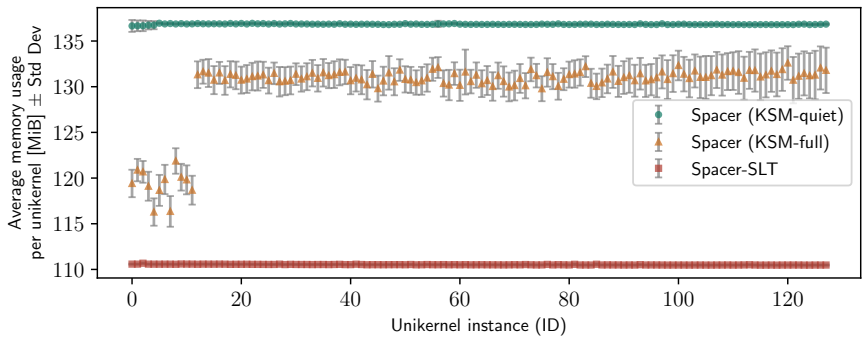


Figure 4.6: *Average physical memory usage per instance for 128 benchmarked FaaS unikernels (including VMM). Spacer-SLT ensures consistent memory usage across all instances.*

The KSM-quiet mode is overwhelmed from the outset due to its slow scanning rate, offering only marginal memory savings throughout the experiment. Even though we focused on Spacer with KSM, the same issue occurs with other configurations, such as DCE and Default (not shown for readability). In addition to average memory usage, we also illustrate the variation in memory consumption over time during each unikernel's 10-second lifetime. KSM-full exhibits greater fluctuations

due to its policy, which can still merge portions of each unikernel's memory.

These experiments demonstrate that Spacer-SLT outperforms KSM in deduplication efficiency when handling a (large) batch of multiple unikernels running concurrently. KSM struggles to merge all sharing opportunities due to the high volume of memory pages it needs to scan and potentially merge.

4.4.2 KSM Variability vs. Spacer-SLT Consistency

To better understand how KSM's memory deduplication performance scales with system load, we measured the physical memory usage of a specific unikernel instance while varying the total number of concurrently running instances (4, 8, 16, 32, and 64). Each configuration was executed five times to capture variability, and results are shown in Figure 4.7. For readability, we only compared Spacer (with KSM-full) and Spacer-SLT.

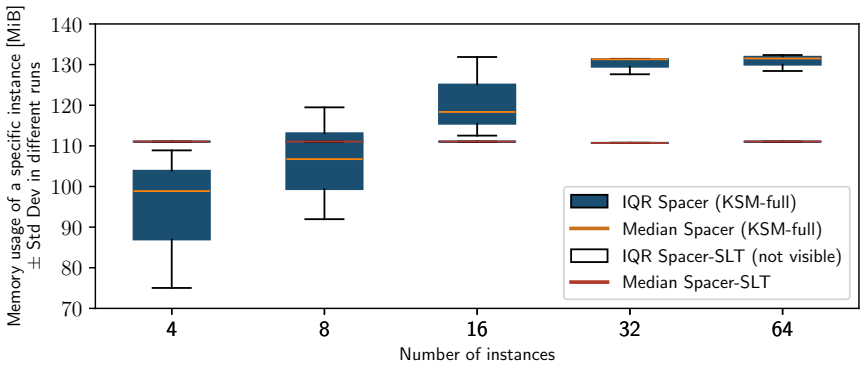


Figure 4.7: *Memory usage of a specific single instance over 5 runs with concurrent unikernels. KSM shows increasing and less variable usage under heavy load. Spacer-SLT remains stable but slightly higher at low concurrency due to non-merging of writable pages.*

The plot reveals that, when using an aggressive KSM mode, the memory usage of the observed instance increases with the number of concurrently running instances, while the variability across runs decreases. This trend suggests that KSM's ability to detect and merge duplicate pages is heavily influenced by system load. With fewer instances, KSM has more time and CPU resources to scan memory and perform deduplication, resulting in lower – but more variable – memory usage. As concurrency increases, KSM becomes overwhelmed by the growing memory footprint and cannot keep up with the rate of incoming pages, leading to higher and more stable (but less efficient) memory usage.

In contrast, Spacer-SLT exhibits no variability across runs, consistently achieving the same memory footprint regardless of the number of running instances. However, unlike KSM, Spacer-SLT does not merge writable pages. As a result, in low-concurrency scenarios where KSM has time to deduplicate more aggressively, it can achieve better memory reduction. However, Spacer-SLT offers a predictable and efficient memory profile under load, making it more suitable for high-density FaaS deployments requiring scalability.

4.4.3 Evaluating Spacer-SLT

This section showcases Spacer-SLT's ability to achieve better metrics compared with the traditional approach of loading unikernels and relying on KSM for merging identical memory pages. To ensure a fair comparison, we evaluated Spacer-SLT against KSM with its default settings (KSM-quiet) because our testing criteria are more deteriorated by higher CPU usage than by slower KSM convergence speed.

We consider an environment that includes various applications ported as unikernels. We used 20 different applications for some experiments: most of them have a network stack since unikernels are fairly well suited for a cloud environment. We then used different types of unikernels: short-lived unikernels that last only a few milliseconds and long-lived unikernels that last tens/hundreds of seconds. In addition, we also consider different workloads: some unikernels are idle, while others are put under load. We limited ourselves to 20 applications because it would have been relatively time-consuming to port new ones.

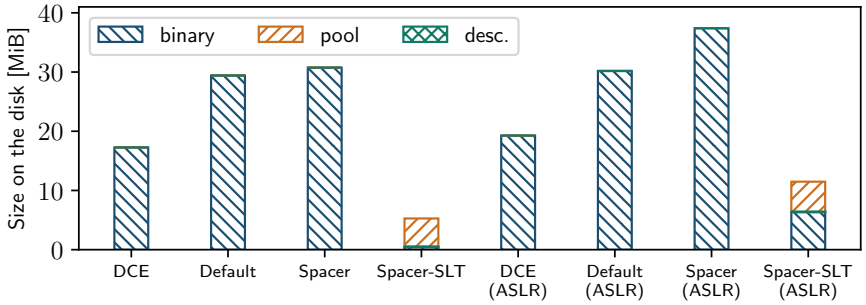


Figure 4.8: *Spacer-SLT reduces disk usage for 20 unikernels by extracting common libraries into a shared pool.*

First, we analyzed the size of the unikernel files (on the disk). Using Spacer caused inflation. This inflation is due to the header string table section (.shstrtab) which contains more entries[2], and the trampoline tables for the ASLR version. Figure 4.8 illustrates the size occupied on the disk by the unikernel files. Since libraries (code and read-only data) are extracted into a common library pool, our approach considerably reduces the total size. For the vanilla version, it leads to a 3.3× reduction compared with DCE, a 5.5× reduction compared with Default, and 5.8× compared with Spacer (KSM). This reduction is smaller for ASLR versions where trampoline tables and non-shareable read-only data (different relocations) cannot be shared between instances, resulting in a 1.7× reduction compared with DCE unikernels, and in a 2.6× and 3.3× file size reduction compared with Default (KSM) and Spacer (KSM).

We then ran these 20 applications (one application per unikernel instance) to measure the total amount of memory used and to compare our approach with KSM. Figure 4.9 shows the average memory usage (including all unikernel instances and the VMM), computed over 30 experimental runs.

[2] Instead of having aggregated sections, they are disaggregated per libraries.

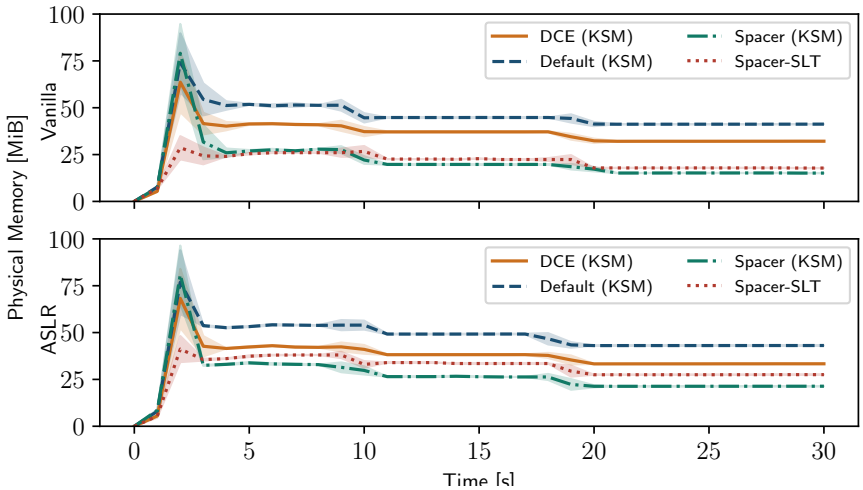


Figure 4.9: *Evolution of the physical memory used by 20 different unikernels and the VMM (vanilla and ASLR) over time. Spacer-SLT performs memory deduplication at load time, which directly reduces memory usage.*

With KSM, memory peaks appear at the beginning and then slowly decrease thanks to page merging. With Spacer-SLT, the read-only pages are mapped directly to the same physical frames and memory peaks are reduced. It can also be seen that KSM can merge more pages than our custom loader. Indeed, after convergence, memory usage fluctuates between 15 MiB for Spacer (KSM) and 18 MiB for Spacer-SLT. This is because KSM merges identical writable pages such as pages related to heap, video memory, etc. This happens mainly on idle unikernels that tend to have identical writable pages. For the ASLR version, the same observations can be made except that the consumed memory is a bit higher. This is due to the size of trampoline tables and non-shareable read-only data of some libraries (e.g., newlib [162], lwip [170], etc.), which generate a memory overhead.

We also conducted controlled experiments to ensure that deduplication at load time does not impact the overall system performance. For Figures 4.10, 4.11 and 4.12, we isolated a single CPU core using `isolcpu` and ran a specific instance for measurement while pinning it to that core. In parallel, on the other cores, a real-world scenario is simulated by launching the 20 unikernels from the previous experiment. Among those, some are idle (like Nginx and DNS) and others are actively running (like zlib and SQLite). Eight unikernels were benchmarked, one at a time. We used the perf [85] tool to measure performance on the benchmarked instance, repeating the experiments 30 times for each of the 8 unikernels. Four were short-lived: a Hello World unikernel, a lambda function performing a 2× image upscale, a modified Nginx with full lwip support (network stack), and a modified DNS using partial lwip support. The latter two were stopped just before invoking `accept()` to simulate ephemeral unikernels. The other four were long-lived: a Mandelbrot generator that saves the result in an image file (1280×720 with 10000 iterations), a parallel 2000×2000 matrix multiplier that also saves the result in a file, the SQLite speed test [194], and a zlib unikernel that inflates and deflates a 10MiB file. This

selection covers a range of short-lived and long-lived workloads.

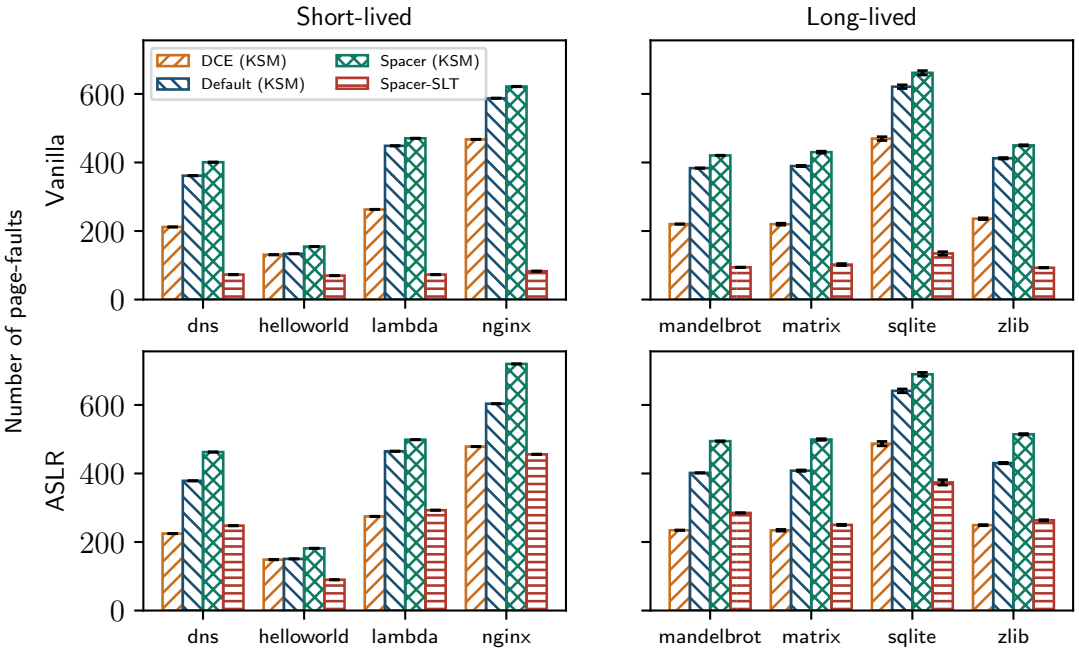


Figure 4.10: *Average number of page faults for vanilla and ASLR unikernels: Spacer-SLT reduces page faults in the vanilla configuration. However, under ASLR, the use of trampoline tables and no-shareable (read-only) data can lead to more page faults than with DCE.*

To assess the library pool's effectiveness, we begin by analyzing the number of page faults. Figure 4.10 represents the number of page faults issued by different unikernel configurations (aggregated for the unikernel and the VMM).

Our findings show that the number of page faults depends on the configuration. Spacer (KSM) generates the most page faults due to memory inflation. Compared with other configurations, Spacer-SLT significantly reduces page faults. For vanilla workloads, Spacer-SLT generates on average 3×, 4.6×, and 5× fewer page faults than DCE (KSM), Default (KSM), and Spacer (KSM), respectively. For some unikernels like Nginx, the reduction is even more substantial, with Spacer-SLT incurring 5.6×, 7×, and 7.6× fewer page faults than the configurations above. The low number of page faults in Spacer-SLT is likely due to its pre-loading of library code and read-only data into memory. For ASLR, the impact of Spacer-SLT is diminished because trampoline tables and non-shareable `.rodata` sections introduce additional page faults when loaded into memory. As a result, Spacer-SLT exhibits an average number of page faults similar to DCE (KSM), but it achieves 1.5× and 1.9× fewer page faults compared with the other two configurations.

We also analyze the VMM load time and the total execution time of different unikernels. In this context, load time refers to the time the VMM takes to initialize and fully load the unikernel into memory, without starting execution. Figure 4.11 presents these load times for different unikernel configurations. Although load time has only a minimal impact on long-lived unikernels, we include these results for completeness and consistency.

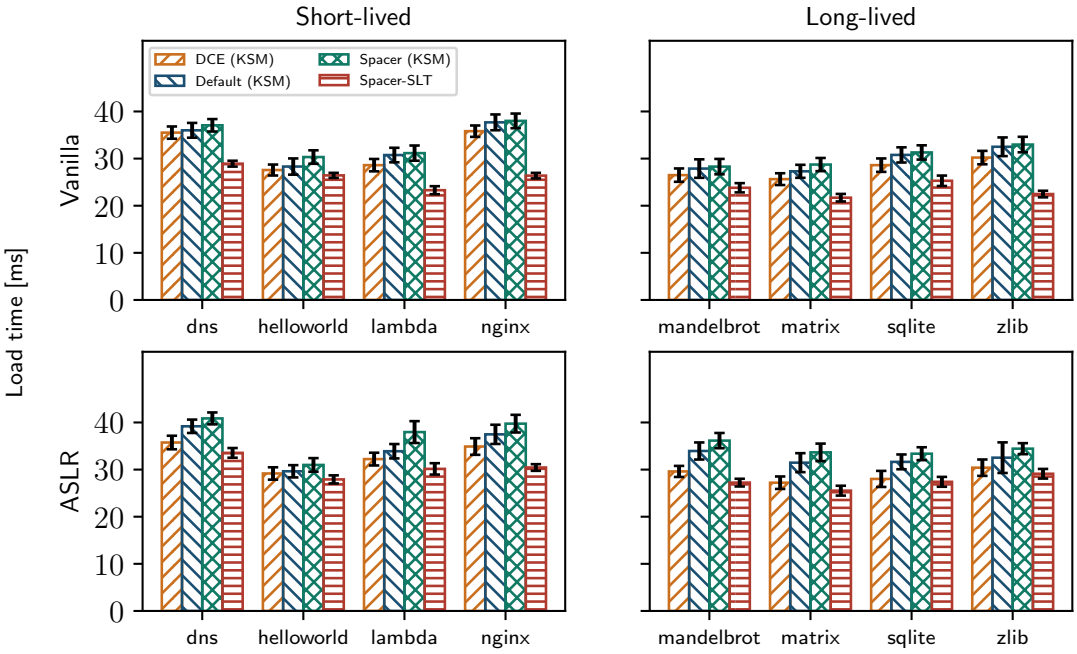


Figure 4.11: *Average load time (ms) for heterogeneous unikernels: Spacer-SLT is slightly faster than DCE, Default, and Spacer (KSM) as most libraries are pre-loaded into memory.*

To measure load time, we used Firecracker's internal timestamping mechanism, which records the time elapsed until a specific I/O port is written [150]. As can be seen, Spacer-SLT reduces the load time since most of the libraries are already loaded into memory. Other configurations require parsing the underlying ELF file and then loading the unikernel into memory, which requires I/O and unnecessary copies to allocate frames (page faults). For the vanilla version, Spacer-SLT starts up faster than DCE (KSM), Default (KSM) and Spacer (KSM) by up to 1.2×, 1.3× and 1.4× respectively. As expected, the ASLR version incurs a load time penalty due to trampoline tables and non-shareable read-only data loading. In this case, Spacer-SLT with ASLR has on average similar load times to DCE (KSM) but remains faster than Default (KSM) and Spacer (KSM) by around 17% and 25%, respectively.

The total execution time is the time the unikernel takes to boot, run a certain load, shutdown and be destroyed. It is shown in Figure 4.12. Spacer (KSM) introduced a slight penalty in performance due to memory fragmentation (i.e., biggest underlying ELF file to parse and more pages to load) [73]. Spacer-SLT changes the status quo and provides better results than its three counterparts.

This approach consistently yields lower execution times compared with KSM-based configurations. The first explanation for this difference is that KSM's memory scanning daemon consumes CPU cycles – especially in the case of long-lived unikernels, where KSM has enough time to analyze and merge identical pages. In contrast, short-lived unikernels might be partially missed by KSM. A second factor, which affects both short-lived and long-lived unikernels, is the overhead of fully parsing the ELF file and loading the unikernel into memory. This process involves I/O operations and redundant copying to allocate memory frames, which Spacer-

SLT eliminates. In some cases, Spacer-SLT improves execution time by 9%, 11%, and 13% compared with DCE (KSM), Default (KSM), and Spacer (KSM), respectively. Although ASLR-based unikernels introduce a slight overhead, Spacer-SLT still demonstrates the same performance advantages.

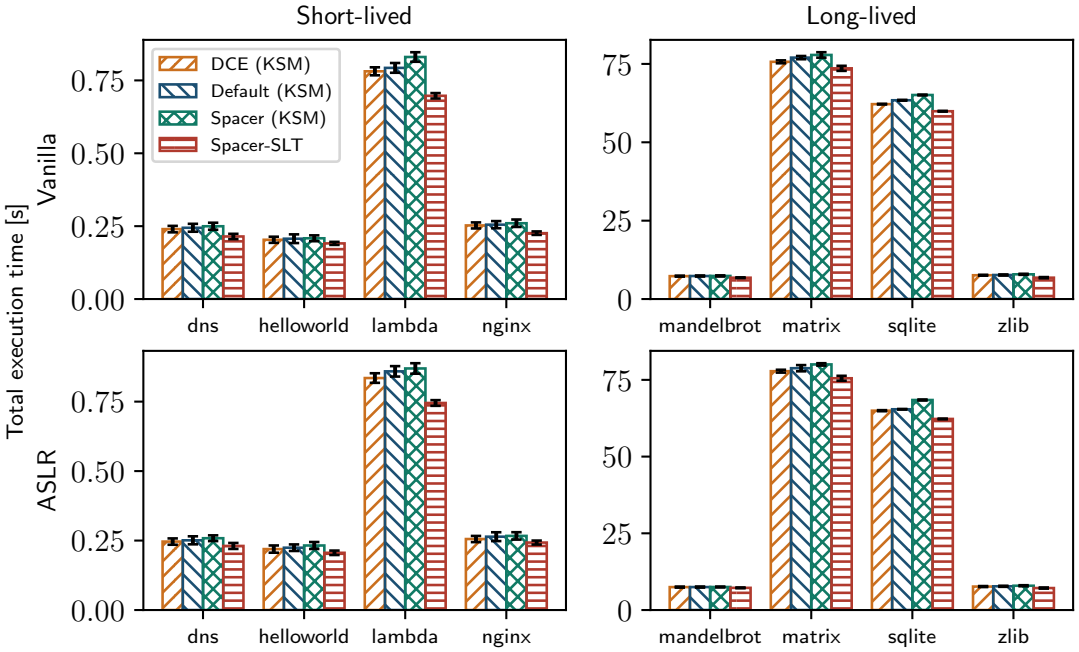


Figure 4.12: *Average total execution time (seconds) for different unikernel configurations: Spacer-SLT outperforms KSM-based configurations, especially for long-lived unikernels.*

The number of TLB flushes of 20 different unikernels is shown in Figure 4.13. The reported values correspond to the mean over 30 runs, and the error bars represent the standard deviation. In this experiment, we employed ftrace[164] to trace `flush_tlb*` calls (e.g., `flush_tlb_mm`, `flush_tlb_range`, etc.[137]) issued by the Linux kernel, allowing us to indirectly monitor TLB shootdowns. These calls were triggered by KSM and memory management operations performed across the full VM stack – both the Firecracker VMM and its unikernel guests. The benchmark is the same as the one used for Figure 4.9. We observe that Firecracker produces fairly similar results across the different configurations, except for Spacer-SLT, which performs much better since the libraries are already cached. When we look at the results with KSM, the number of TLB flushes is quite high, particularly for Spacer (KSM). Once again, this can be explained by the fact that more pages are merged into the same physical frame by KSM, invalidating the TLB and hence the higher number of TLB flushes. Spacer-SLT, which operates independently of KSM, significantly reduces TLB flushes compared with other configurations. It can achieve up to 10× fewer flushes compared with DCE (KSM), 12× fewer compared with Default (KSM), and 13× fewer compared with Spacer (KSM).

As demonstrated by our experiments, Spacer-SLT consistently outperforms KSM configurations, including the KSM-quiet setting. The performance gap would be even wider when compared with a more aggressive KSM mode.

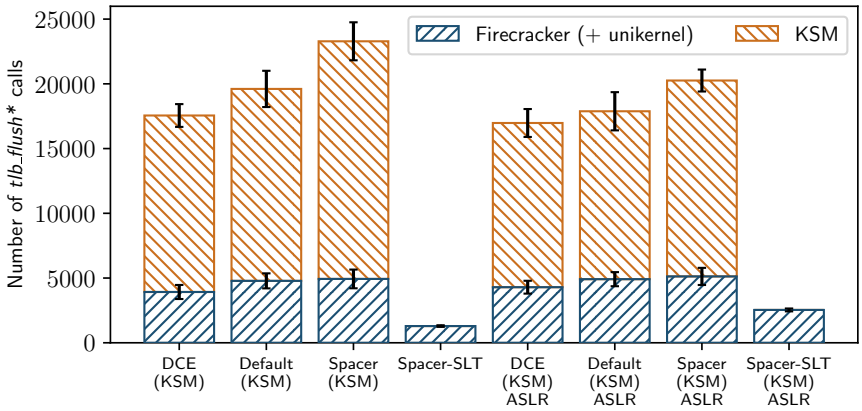


Figure 4.13: *Average number of TLB flushes issued by 20 heterogeneous unikernels (monitored at the VMM and KSM level) for vanilla and ASLR configurations. Spacer-SLT generates fewer TLB flushes than the KSM-based configurations.*

4.4.4 Managing the Library Pool

This section evaluates the impact of loading libraries when they are initially stored on disk. By default, our system preloads all libraries into memory, assuming they will be used soon after unikernel startup. To quantify the cost of loading libraries from disk into `/dev/shm`, we measured the latency our custom loader takes to move libraries from a dedicated unikernel workspace on disk into the shared memory pool, if the library is not already cached there.

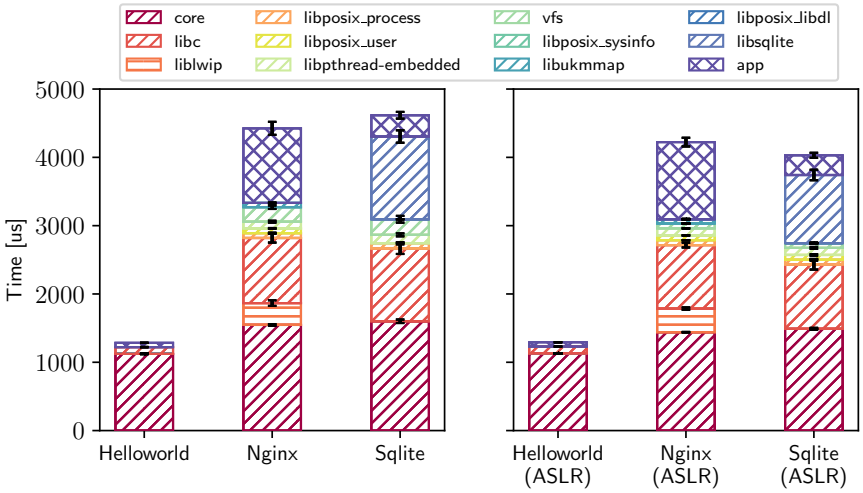


Figure 4.14: *Library caching latency (µs). Our custom loader moves libraries to `/dev/shm`. Core libraries, libc, the application, (and lib-sqlite for SQLite) account for 80% of caching latency.*

For this experiment, we considered three different unikernels: Helloworld, Nginx, and SQLite (vanilla and ASLR). Figure 4.14 shows the average time required for each library to be cached in `/dev/shm` over 30 runs, with error bars indicating the standard deviation. In general, copying libraries into `/dev/shm` has a minimal impact on performance. For instance, caching all libraries for Helloworld, Nginx, and SQLite takes only 1.3ms, 4.5ms, and 4.6ms, respectively. The significant difference between Helloworld and the others is due to Helloworld using a minimal libc,

while the others rely on a larger libc. The core libraries, the libc and the application itself (as well as lib-sqlite[3] for SQLite) take up most of the caching time (around 80%). For ASLR, the conclusions are similar, except for libraries with unshareable (read-only) data sections, which lead to smaller library sizes and faster caching. This applies to libc and lib-sqlite, which take less time for the ASLR version (e.g., 4ms for SQLite and 4.2ms for Nginx).

4.4.5 Running Multiple ASLR Instances

We also analyze multiple ASLR-based unikernels by running the same unikernel several times. This experiment uses a similar setup as Section 4.4.1 but with 64 ASLR-based unikernels. As before, we launched these instances sequentially, but this time, each with an underlying unique file (on disk) and memory layout[4]. The results can be seen in Figure 4.15, which reports the average memory usage over 30 runs together with the standard deviation.

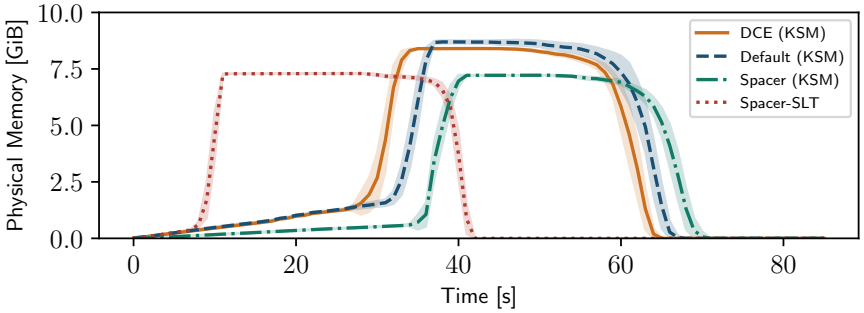


Figure 4.15: *Average memory usage of 64 FaaS ASLR unikernels (+VMM). KSM-based configurations exhibit delayed execution due to I/O wait times. Spacer-SLT reduces execution time by using a shared library pool as an I/O cache.*

When using link-time ASLR, each instance has a unique memory layout, enhancing security but introducing significant disk I/O overhead when instances load simultaneously, delaying execution. This overhead stems primarily from loading numerous distinct underlying files and is most pronounced when instances are launched in rapid succession. The effect is more observable with Spacer (KSM), which uses larger ELF files due to trampoline tables. Spacer-SLT mitigates this issue by caching libraries in memory, thereby reducing disk I/O to only essential components and achieving up to 1.7× faster execution than Spacer (KSM). When all unikernels are launched (at t≈9s for Spacer-SLT), Spacer-SLT exhibits slightly higher average physical memory usage than Spacer (KSM), as it does not merge writable pages. However, it still maintains lower memory consumption than the DCE and Default configurations. An in-depth discussion of the ASLR trade-off is provided in Section 4.5.5.

[3] For SQLite, we distinguish between the SQLite shell (application), and the library providing the SQLite API for interacting with the database.

[4] With our Unikraft implementation, ASLR is managed at link time, which creates a different underlying file per instance.

4.4.6 Spacer-SLT vs. Dynamic libraries

As a final experiment, we compared Spacer-SLT with dynamically linked unikernels. Although Firecracker and Unikraft can be modified to support dynamically linked unikernels, doing so would require substantial changes due to the position-dependent nature of certain KVM-internal libraries in Unikraft. To simplify development and reduce engineering overhead, we ported Spacer-SLT and the Unikraft codebase to user space. We also extended Unikraft – originally limited to static linking – to support dynamically linked unikernels, which involved extensive modifications (50 files changed, 448 insertions, and 82 deletions), and implemented a lightweight user space loader to run them. After these changes, we ported the same applications used in previous experiments, as well as a Python-based lambda function (with a 104MiB initrd), and systematically monitored memory usage and execution time (via perf [85]), including that of the user space loader. Each experiment was repeated 30 times, and we report the mean along with the standard deviation.

Table 4.1 presents the average execution times of dynamically linked unikernels, Spacer-SLT and Spacer-SLT (ASLR). Dynamically linked unikernels incur a performance penalty due to runtime library relocations and symbol resolution (via lazy binding), which are avoided by Spacer-SLT. As previously observed, applying ASLR to statically linked unikernels introduces a small overhead.

Applications	Dynamically linked	Spacer-SLT	Spacer-SLT (ASLR)
Helloworld	0.004 ± 1.56e-05	0.002 ± 6.44e-06	0.003 ± 6.81e-06
Lambda	0.564 ± 7.52e-05	0.541 ± 7.34e-05	0.551 ± 7.35e-05
Mandelbrot	0.373 ± 7.49e-05	0.358 ± 7.55e-05	0.359 ± 7.11e-05
Nginx	0.215 ± 6.45e-03	0.175 ± 6.01e-03	0.182 ± 6.61e-03
Python	2.406 ± 1.31e-04	2.101 ± 4.28e-06	2.195 ± 6.17e-04
SQLite	48.491 ± 4.22e-03	47.521 ± 3.80e-03	47.892 ± 4.43e-03
Zlib	4.002 ± 1.87e-04	3.817 ± 2.25e-04	3.932 ± 5.17e-04

Table 4.1: *Average total execution time (in seconds) of 7 heterogeneous unikernels (and the loader) across various categories. Spacer-SLT reduces execution time relative to dynamic linking, while enabling ASLR introduces a small additional overhead.*

Figure 4.16 compares memory consumption (during 60 seconds) for the same set of unikernels running together, each with an infinite loop at the end of the workload. Dynamically linked unikernels consume, on average, 1.2× more memory due to the overhead introduced by the underlying loader, dynamic linker (ld.so) and additional structures such as PLT/GOT tables. Once again, statically linked unikernels with ASLR show slightly higher memory usage than Spacer-SLT.

We discuss the relevance of using Spacer-SLT as opposed to dynamic libraries in more detail in Section 4.5.4.

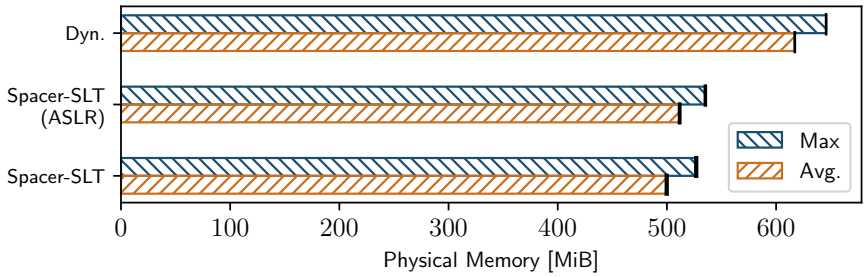


Figure 4.16: *Average memory usage of 7 heterogeneous unikernels (and the loader) across various categories. Spacer-SLT consumes less memory than dynamically linked unikernels.*

4.4.7 Towards a Solution

Our evaluation demonstrates that the proposed architecture effectively addresses the issues outlined in Section 4.2. By using a custom loader, we eliminate the need for a background deduplication daemon. Spacer-SLT performs memory deduplication at load time, thereby avoiding KSM's convergence time (§1). It also conserves CPU resources by eliminating the need to scan and merge memory pages across unikernels, leaving more processing power available for unikernels (§2). Additionally, Spacer-SLT provides consistent memory savings regardless of system load, unlike KSM, whose effectiveness deteriorates as concurrency increases, leading to degraded memory reduction (§3). The approach also reduces I/O overhead and TLB shootdowns (§4) by caching and avoiding the allocation of redundant memory frames that would otherwise require merging. By mapping only read-only pages, it reduces exposure to known CoW-based deduplication attacks and the performance overhead of deduplicating writable pages (§5). Finally, Spacer-SLT minimizes disk usage by disaggregating unikernels into shared libraries, eliminating the need for complete ELF binaries per instance (§6). Instead, each unikernel requires only a description file and a minimal binary containing non-shareable read-only data and trampoline tables (for the ASLR version).

4.5 Discussion

This section discusses various key aspects of our approach and implementation.

4.5.1 Combining Spacer-SLT with KSM

Spacer (KSM) uses less memory (after KSM convergence) than performing deduplication at load time. This is due to identical writable pages merged into the same single frame. Merging both read-only and read-write pages is the default behavior of KSM. Although this behavior allows consuming less memory[5] than Spacer-SLT, it leads to memory disclosure attacks which are based on a difference in write access times on the deduplicated memory pages that are recreated by CoW [231, 189, 32,

[5] Especially when identical long-lived and inactive unikernels are running on the same server.

22]. It also has several disadvantages as shown in our evaluation section. However, a cloud provider can always combine our approach with KSM by simply enabling it using our custom loader. The read-only pages will already be merged, and KSM will only merge the identical writable pages, resulting in lower CPU cycles. System administrators are expected to keep KSM on or turn it off according to their environments and type of workload, with all the consequences that it implies.

4.5.2 Using Another Deduplication Scanner

Improved memory scanners like UKSM [230] provide an enhanced version of KSM by prioritizing memory regions to accelerate the scanning and merging process. While UKSM offers improvements over traditional KSM configurations, it only applies through custom patches on Linux kernels 4 and 5. The project is also no longer actively maintained by its original developers [235], raising concerns about its long-term suitability for production environments. UKSM aims to balance the approaches of KSM-quiet and KSM-full configurations. While it merges more memory than KSM-quiet, it does not achieve the aggressive memory deduplication of KSM-full. This trade-off reflects UKSM's goal of optimizing memory deduplication efficiency without significantly compromising performance. However, UKSM still lags behind Spacer-SLT in both performance and memory deduplication. Spacer-SLT delivers the lowest execution time and memory footprint due to its proactive memory deduplication approach. Unlike traditional memory scanners, such as those in KSM and UKSM, which analyze all pages at runtime, Spacer-SLT performs this analysis during load time. By eliminating CPU-intensive runtime scanning and merging, and by leveraging caching, Spacer-SLT achieves superior performance during unikernel execution.

4.5.3 Load Time Memory Deduplication in the Host Kernel

To implement our solution, we focus on the scenario in which unikernels are deployed and managed by a VMM. This choice is motivated by the fact that the Unikraft platform relies on Firecracker as its VMM [204]. Additionally, Amazon Web Services (AWS) Lambda uses Firecracker to provision secure sandboxes for executing customer functions [7].

Another key decision was determining where to implement memory deduplication during load time. We chose to implement it within the VMM (i.e., in user space), as this approach is both practical and straightforward. It eliminates the need for a modified host kernel, which can be restrictive for cloud operators. For instance, UKSM is not yet part of the standard Linux kernel and requires custom patches. These patches are only available for older kernel versions and are obsolete for newer ones, as discussed in Section 4.5.2.

As discussed in Section 4.2, implementing our approach within kernel space is a viable alternative. Specifically, we could develop Daemonless Kernel Samepage

Merging (DKSM), which identifies and merges identical read-only pages across different unikernels during the loading process. Unlike KSM and UKSM, DKSM eliminates the need for a dedicated kernel thread to scan and merge memory. This efficiency is achieved by focusing exclusively on read-only pages, thereby removing the requirement to rescan already merged regions. To implement this, the kernel could maintain a global hash table and override the `madvise()` system call. When called on read-only pages, the kernel would attempt to merge the advised pages with those already in the hash table. DKSM could use a page hashing algorithm to determine whether the pages are identical. If a match is found, the kernel merges the pages by updating their page table entries to point to the same physical frame via the `replace_page` function [12]. If no match exists, the page is added to the hash table for potential future merging.

4.5.4 Using Spacer-SLT or Dynamic Libraries?

The primary goal of Spacer-SLT is to provide a simple and scalable system for performing memory deduplication at load time on statically linked binaries, effectively replacing KSM. It is not designed to support dynamic libraries. Unlike traditional dynamic libraries, which are typically confined to specific operating system environments, Spacer-SLT uses a lightweight, system-agnostic representation. This design facilitates adaptation to formats beyond ELF, including Windows' Portable Executable (PE) format [206], requiring only minimal modifications to the toolchain allowing multiple formats to be supported simultaneously. While supporting dynamic libraries could be a future direction, it introduces complexity, security issues, and overhead due to dynamic loading and symbol resolution. Moreover, as most unikernel projects use static linking [97, 105, 34], Spacer-SLT aligns naturally with their architecture and can leverage compiler optimizations, which are incompatible with dynamic libraries. In summary, supporting dynamic libraries would require substantial changes to the paradigm and architecture, adding runtime overhead and limiting adaptability across diverse systems [46, 53].

4.5.5 Managing ASLR Efficiently

When hundreds of instances are launched simultaneously, we have noticed that the use of ASLR has a significant impact on performance. This is a direct consequence of link time ASLR implementation, which generates distinct binary files for each instance and thus impacts I/O operations. In Section 4.4.5, we considered an extreme configuration in which all instances differ (libraries are also shuffled). This approach provides greater security but negatively impacts performance. Alternatively, some instances can be identical (sharing the same memory layout and underlying file), which improves performance. Another possibility is to assign one ASLR configuration per machine and change it periodically offline. Overall, these choices involve a trade-off between security and performance.

4.5.6 Spacer-SLT and Snapshots

Snapshotting is a common technique used in FaaS platforms to reduce cold-start latency and enable fast instantiation of functions by reusing pre-initialized runtime states [4, 10, 179]. These snapshots typically capture a function's memory image after initialization, allowing subsequent invocations to resume from this preloaded state rather than starting from scratch. While effective, this approach introduces significant storage and I/O overhead – especially for heavyweight runtimes like Python or Go – where snapshots can exceed several hundred megabytes. Moreover, snapshot reuse is often complicated by ASLR, which introduces variability in memory layouts and may require generating new snapshots per instance.

Spacer-SLT offers a fundamentally different approach by performing memory deduplication at load time, without capturing runtime memory images. Unlike existing approaches that require one snapshot per application, it operates directly on minimized ELF files (which are considerably smaller than full snapshot states) to identify and share identical memory regions across unikernels. Consequently, Spacer-SLT delivers a scalable, low-overhead approach that both simplifies deployment and enhances unikernel performance.

4.5.7 Securing the Library Pool

In Spacer-SLT, a library pool is maintained by extracting libraries (code and read-only data) into `/dev/shm`, allowing unikernels to efficiently share memory and reduce redundancy. To ensure the security of this setup, strict roles and privileges must be enforced by the cloud provider: libraries must be created and managed with restrictive permissions – read-only for authorized users and writable only by the administrator responsible for maintaining the pool – to prevent unauthorized access or modification. These measures help Spacer-SLT maintain a strong security posture while enabling efficient and secure memory sharing.

4.6 Related Work

This section reviews related work on memory deduplication and unikernels, as well as serverless computing, and compares it to our approach.

Memory deduplication. Given the limitations identified in our evaluation of the memory deduplication scanners [11, 230, 218, 158], this section presents a comparison between our approach and content-based deduplication techniques. Disco [37] uses transparent page sharing with deduplicating CoW disks, while Satori [140] enhances Xen [26] by analyzing para-virtualized virtual disks to improve sharing. Transcendent Memory [130] introduces an API for sharing identical memory pages across VMs. Unlike these para-virtualization-dependent methods, our approach operates without such modifications and is tailored specifically for unikernels.

If we refer to memory deduplication related to unikernels, there are fewer research papers on the subject. Most frameworks focus on reducing the memory footprint of individual instances but overlook high memory usage when running hundreds/thousands in parallel. Solutions like KylinX [238] use pVM forks and inter-pVM communication APIs, while Nephele [125] enables memory and I/O cloning, both relying on Xen and heavy hypervisor modifications. In contrast, our approach performs deduplication at load time without relying on forks or cloning and is straightforward to implement. ORC [168] enables fine-grained binary object sharing with strong tenant isolation via a minimal Trusted Computing Base (TCB), relying on hardware memory capabilities. While both ORC and Spacer-SLT improve memory deduplication, ORC uses object-level deduplication in a shared address space, requiring custom compilers, loaders, and specialized hardware like CHERI/Morello [227, 86]. Spacer-SLT, in contrast, abstracts unikernels into a shared library pool, achieving deduplication without hardware dependencies. It ensures security within a shared read-only pool and operates on standard x86-64 architectures, offering greater flexibility and compatibility with existing systems.

Unikernels. Several prior works have focused on optimizing various aspects of unikernel design and deployment. These include efforts to reduce application porting complexity [105, 148, 97], as well as various techniques for minimizing startup latency [34, 107]. Spacer-SLT complements these approaches by enabling memory-efficient and secure unikernels, further reducing cold-start overheads through library caching and early-stage deduplication of read-only memory pages. Efforts such as uIO [141], CubicleOS [169], and FlexOS [116] introduce intra-unikernel isolation mechanisms using Intel Memory Protection Keys (MPK) [93] to confine faults within memory domains. However, these systems do not address memory efficiency in multi-tenant serverless environments, nor do they offer integration with content-based deduplication or ASLR-aware memory sharing strategies as provided by Spacer-SLT.

Serverless computing. Serverless computing [23, 134, 41, 96], and more particularly FaaS [177, 7, 126], has become a prominent research focus, with efforts targeting reduced start-up latency and memory usage. Solutions like SEUSS [39] optimize execution by reusing computed results, SAND [4] enhances scalability via elastic provisioning and adaptive load balancing, and OFC [144] introduces in-memory caching for resource-efficient function invocations. Faasm [179] employs lightweight isolation to minimize virtualization costs, while Medes [171] reduces memory duplication in warm sandboxes by merging memory regions across function instances. Although these approaches focus on reducing startup latency and resource consumption, they are not designed with unikernels in mind and fail to address the drawbacks of using external memory scanning mechanisms. UaaF [192] links serverless functions with necessary libraries in unikernels, leveraging VM-

FUNC [93] for low-latency cross-sandbox invocations and a novel programming model to reduce memory and boot time. However, VMFUNC poses security risks by allowing writable memory sharing between unikernels [99, 188, 32, 5]. Our approach avoids these risks by deduplicating only read-only memory pages, preventing such attacks. Additionally, it supports ASLR, manages heterogeneous unikernels, and provides direct comparisons to DCE, offering broader flexibility and security enhancements compared with UaaF.

4.7 Summary

In this chapter, we designed, implemented, and evaluated Spacer-SLT, a loader for statically linked unikernels that enables deterministic read-only memory deduplication at load time. Spacer-SLT has been integrated into Firecracker, demonstrating significant improvements across various metrics, including reduced memory usage, decreased disk space consumption, and enhanced startup and execution times. These gains are achieved through a combination of fewer disk operations, reduced page fault rates, and minimized TLB flushes, compared with traditional scanner-based memory deduplication methods or dynamically linked approaches.

Spacer-SLT contributes to improving the efficiency of cloud computing infrastructure by increasing memory efficiency, which enables higher consolidation density. This allows a given workload to be deployed on fewer servers, thereby reducing overall energy consumption, particularly for microservices and short-lived workloads, such as lambda functions. Furthermore, Spacer-SLT can be adapted to other unikernel ecosystems and VMMs, broadening its potential impact.

4.8 Potential Improvements

Adaptability to other formats, unikernels and VMMs. The system-agnostic representation of Spacer-SLT's design facilitates its adaptation to other binary formats beyond ELF, such as Windows' PE format. This would require only minimal modifications to the toolchain, allowing simultaneous support for multiple formats. In addition, Spacer-SLT is designed to be easily adaptable to other unikernel ecosystems and VMMs beyond Unikraft and Firecracker, which could broaden its potential impact.

Exploiting new hardware with Spacer-SLT. Spacer-SLT opens several hardware-oriented directions for future investigation. Instead of placing shared-library code and read-only data in a temporary file system backed pool, Spacer-SLT could exploit Non Volatile Memory (NVMEM) regions, enabling persistence across reboots without relying on Linux-specific volatile storage. At the rack level, Compute Express Link (CXL)-disaggregated [93] memory could expose a single shared read-only library pool across multiple nodes, eliminating the current per-node replication over-

head. Hardware capability-based architectures such as CHERI/Morello [86] could potentially enable finer-grained memory isolation mechanisms beyond page-level protection, which may eventually permit safer forms of writable-page sharing and deduplication.

This page intentionally left blank.

5 VERSIONING

Overview

In the previous chapters, we explored the benefits of memory alignment when working with different unikernels. However, we did not examine the impact of running multiple unikernels with different library versions. Library versioning introduces both challenges and opportunities for memory deduplication. Indeed, managing updates and versioning in statically linked unikernels is particularly difficult due to their tightly coupled structure and the lack of built-in versioning mechanisms. As a result, even minor changes in a library can lead to entirely new memory layouts, significantly increasing memory consumption when multiple instances are run concurrently.
To address these challenges, we propose Spacer-Δ, a novel framework for managing library versioning in statically linked unikernels. Spacer-Δ combines library alignment with differential analysis to refine memory layouts across library versions. By enabling memory sharing through Kernel Samepage Merging (KSM) and/or a custom loader using a common library pool, our approach significantly reduces memory usage. We evaluate Spacer-Δ across key metrics – including execution time, memory consumption, and disk footprint – and demonstrate efficiency gains, especially when used with a custom loader and library pool. Spacer-Δ integrates seamlessly with Unikraft and could be adapted to other unikernel frameworks with minimal changes.

5.1 Introduction

The static linking model inherent to unikernels [106, 105, 111, 148, 97], while providing benefits such as simplified deployment, low execution overhead, and strong isolation guarantees [46, 53], poses significant challenges for library version man-

agement. Due to their tightly coupled components and compact form, statically linked unikernels are highly sensitive to implementation changes. This affects their underlying ELF structure and, in turn, how their binary is mapped into memory. Even minor differences between instances – such as the addition or removal of a single function – can alter and shift code and data section layouts, yielding distinct ELF binaries and, consequently, divergent memory pages at runtime.

In practice, different projects often embed different versions of the same library, reflecting the tendency of developers and maintainers to rarely update third-party dependencies to the latest version [153, 108]. Unlike dynamically linked systems, where multiple library versions can coexist and be updated independently [55], unikernels rely on static linking, requiring all libraries to be resolved at link time. Consequently, deploying a unikernel with a different library version alongside existing instances frequently results in divergent memory layouts. This reduces the number of identical pages that can be merged by memory deduplication mechanisms such as KSM [11] or the load time deduplication provided by Spacer−SLT (Chapter 4). Each unikernel must therefore maintain its own private memory pages, thereby increasing overall physical memory consumption, even when most of the underlying library functionality remains unchanged.

To mitigate these inefficiencies, we propose Spacer-Δ, a comprehensive framework that ensures backward compatibility across library versions while improving memory efficiency for statically linked unikernels. Spacer-Δ represents each new library version as a delta of the previous one(s), reusing existing symbols and functions wherever possible. It leverages Spacer's library alignment and applies differential analysis to detect and align common library code at the page level, thereby enabling the sharing of pages that would otherwise diverge due to different library versions. The framework also enhances memory sharing through a binary rewriting step performed prior to execution. This step isolates differing instructions between instances – such as `call` instructions targeting distinct functions – thereby preserving page similarity. Although our current implementation targets the Unikraft framework [105], the approach is generic and could be adapted to other unikernel codebases with only minor modifications.

Our main contributions are as follows: (1) We introduce a novel methodology based on page alignment and differential analysis that reduces memory consumption (frame usage) when multiple unikernels with different library versions run on the same machine. (2) From this approach, we derive Spacer-Δ, a proof-of-concept toolset that integrates into existing build pipelines and enables the creation of unikernels that maintain backward compatibility across library versions. (3) We provide a performance evaluation of Spacer-Δ, comparing our approach to other configurations, including DCE, and Spacer. (4) We discuss the limitations of Spacer-Δ and identify some possible areas for improvement.

5.2 Problem Statement and Possible Solutions

This section highlights the challenges associated with library versioning in statically linked unikernels and presents step-by-step solutions to address them.

Unikernels adopt the library OS model [8, 60, 119, 156, 237], decomposing OS functionality into modular libraries. Different unikernels may embed distinct versions of the same library, each potentially adding, removing, or modifying functions. Even when using the alignment provided by Spacer to avoid previously discussed issues (see Chapter 3), this creates two main sources of divergence that hinder page sharing:

1. Divergence from modified code and subsequent shifting. Pages containing modified code are naturally unshareable across versions. However, even minor changes – such as inserting, deleting, or reordering functions – shift the layout of all following code and data. This misalignment propagates across pages, preventing sharing. For example, Figure 5.1 shows two unikernels using different versions of the same library (*uklibV@v1* and *uklibV@v2*). The newer version introduces `f0` and `f4`, removes `f2`, and modifies `f3`, producing divergent pages (marked in red in the figure) that cannot be deduplicated.
2. Divergence from references to shifted code. References to the shifted library functions will encode different addresses across versions, making the pages containing them unshareable. This affects not only the modified library itself but also other libraries that call into it. In Figure 5.1, *uksched* calls `f1`, but the instruction encoding differs across versions of *uklibV*, yielding different pages.

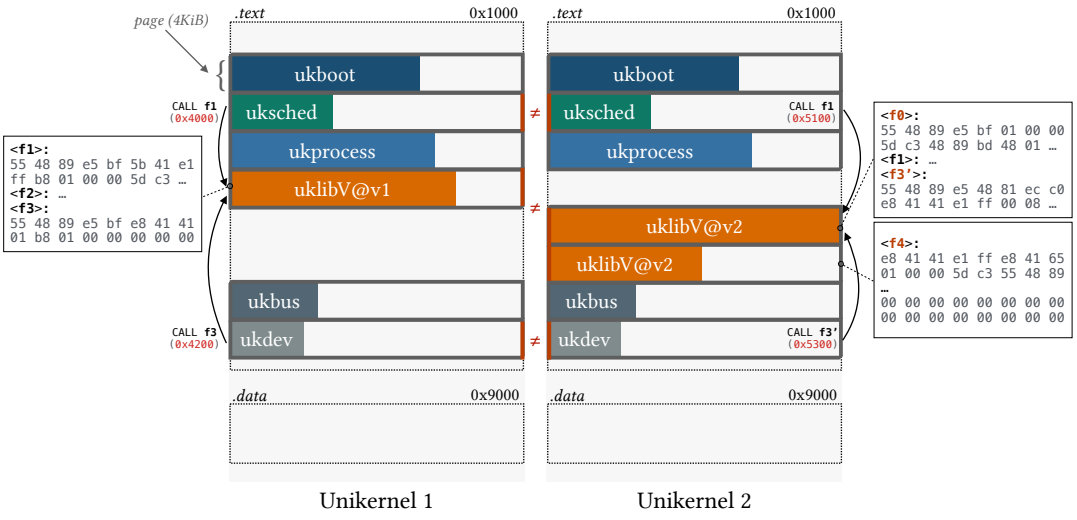


Figure 5.1: *Alignment becomes insufficient when handling various modifications between different library versions: functions may be added, removed, or altered. This results in differences in memory pages across instances and prevents effective memory sharing.*

In the example illustrated in Figure 5.1, only six pages can be shared, while seven remain unshared, for a total of 10 frames. While the figure illustrates the problem

on a small scale, the impact grows significantly with larger libraries and multiple versions or instances.

Given these issues, it becomes necessary to explore function placement strategies that preserve identical pages and improve sharing across library versions, while retaining alignment as a foundation to avoid the issues discussed in Chapter 3. In the remainder of this section, we explore several function placement strategies, starting from simple baselines and progressively introducing more complex techniques.

5.2.1 First Attempt

One straightforward approach is to track the functions of a versioned library across instances and arrange them by similarity (as done by Spacer for libraries): functions whose code is identical in all versions are placed first, followed by those whose code is shared across some but not all versions, and finally unique functions at the end. This concept is illustrated in Figure 5.2, where three unikernels rely on different versions of the same library (*uklibV*). In this example, each function is assigned a fraction of a page and arranged according to its occurrences. Here, we restrict our focus to function placement within a single versioned library, without considering cross-references.

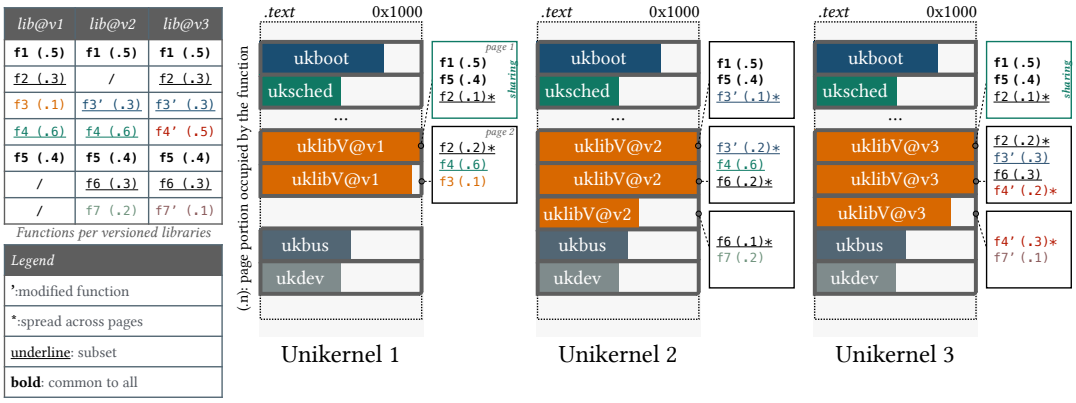


Figure 5.2: *Tracking all functions used across instances and arranging them in a specific order (per occurrence) for each version is not efficient. With this approach, only two pages (green boxes) are identical and can be merged into a single frame. The other pages (black boxes) are mapped to unique frames, requiring a total of 7 frames.*

For example, in the first unikernel, `f1` and `f5`, common to all three instances, are grouped onto the same page. Next, `f2`, present in two instances, is spread across two pages, followed by `f4`. Finally, `f3`, unique to the first instance, is placed individually. With this approach, only two pages (green boxes) are identical and can be merged into a single frame. The other pages are mapped to unique frames, requiring a total of 7 frames.

While this strategy seems promising at first glance, it faces challenges in achieving effective placement when multiple instances with varying subsets of common

functions are involved. As the example illustrates, such variation introduces significant complexity and results in large numbers of distinct pages to manage across instances.

5.2.2 Other Attempts

The approach illustrated in Figure 5.2 provides a basic strategy for arranging functions across pages; however, more sophisticated methods exist to improve space utilization per instance. For example, the problem could be abstracted as a bin-packing formulation [115], where functions are treated as items and pages as bins. However, two key challenges would likely arise: first, the computational complexity increases with the number of functions, library versions, and instances, making exhaustive optimization impractical as the number of unikernels and versions increases; second, cross-instance consistency must be maintained, since identical functions across multiple instances require identical placement to enable page merging. Independently optimizing each instance could produce conflicting layouts among instances, thereby preventing deduplication and reducing memory savings. While we have not implemented such an approach, these challenges suggest it would be difficult to apply in practice.

Another potential strategy is to allocate each unique function to a separate page. While conceptually simple, this would lead to substantial internal fragmentation, particularly when functions are much smaller than the page size (e.g., 4KiB), and increase the number of page faults, further reducing efficiency. Moreover, depending on the situation, it might not even reduce the total number of frames used, which was the primary objective. Taken together, these considerations indicate that none of these strategies can effectively manage the variability of function usage across instances and library versions, leaving the problem largely unresolved.

5.2.3 A Better Approach

Given that the previously discussed strategies proved inefficient in managing library versions, we propose a refined approach that ensures backward compatibility between versions. This strategy, called Spacer-Δ, represents each new library version as a delta (Δ) of the previous version(s), reusing existing data and functions wherever possible. Modified variants and newly introduced functions are placed on a separate page to avoid interfering with pre-existing layouts.

Figure 5.3 illustrates this method using the same library configuration as the previous example. In the second instance, functions from the first version are reused, while `f3'` (a modified version of `f3`), `f6`, and `f7` are placed on a new page which corresponds to Δ*uklibV@v2*. Although this approach increases the total number of pages allocated (9 pages), it increases the sharing (8 pages can be shared) and therefore reduces the number of frames required through deduplication. In this

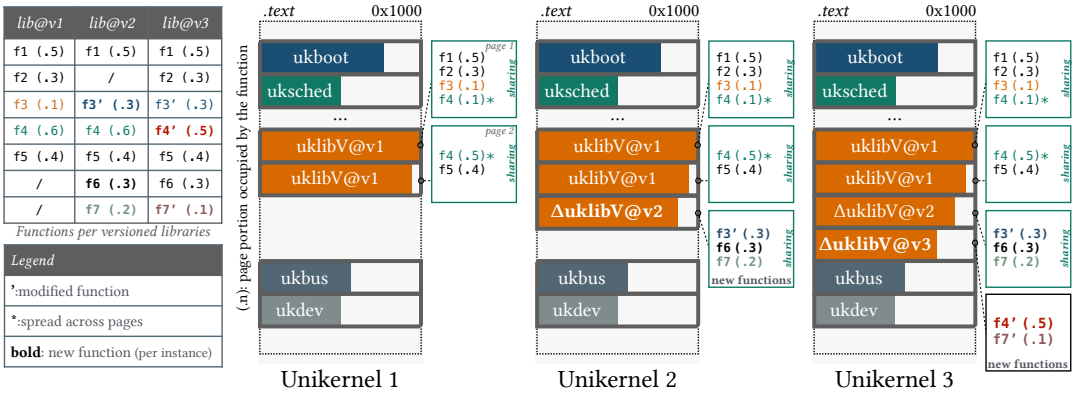


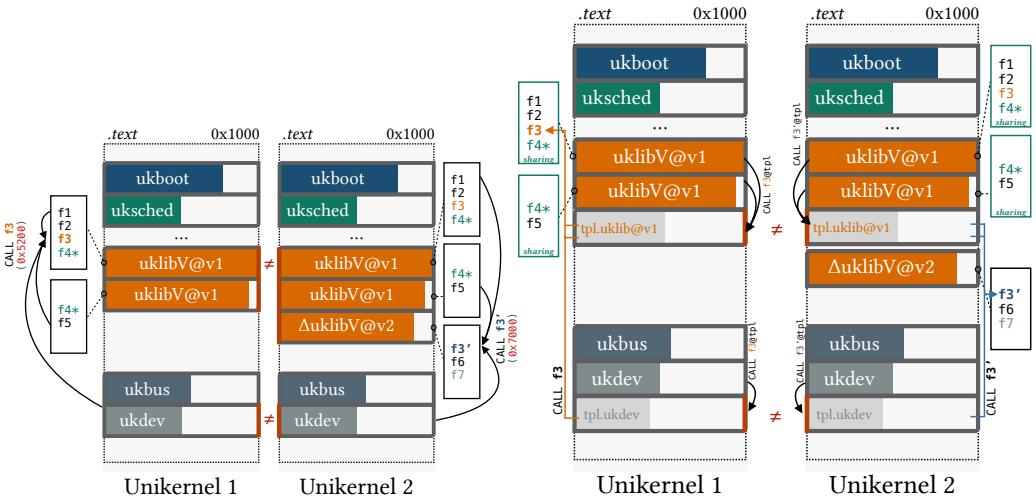
Figure 5.3: *Our approach leverages backward compatibility between library versions by treating each new version as a delta of the previous one. Existing functions remain fixed, while new and modified functions are grouped to new page(s) within each instance. Thanks to memory deduplication, only 4 memory frames are required.*

example, only 4 frames are used, representing an improvement over the previous arrangement and demonstrating notable memory savings.

To further preserve page sharing across unikernels, it is necessary to handle instruction-level differences that arise when code refers to functions or data located at different virtual addresses in different versions. Such differences result in address-dependent instruction encodings that prevent page-level deduplication. To address this issue, we use trampoline tables that redirect such diverging instructions. Figure 5.4 illustrates how instruction-level differences between library versions can prevent page sharing, and how trampoline tables resolve this issue. In this example, functions such as `f1` and `f5` invoke `f3` in one library version, while in another version the corresponding calls target `f3'`, a modified implementation of `f3`. As shown in Figure 5.4a, the differing target addresses lead to distinct `call` instructions, causing the affected page to diverge and preventing deduplication.

To overcome this limitation, we use trampoline tables (*.tpl*) that act as indirection layers. Instead of encoding direct calls to version-specific function addresses, call sites are rewritten to target fixed entries in the trampoline table, which in turn redirect execution to the appropriate function implementation. The same approach is applied to other control- and address-transfer instructions (e.g., `lea`). This eliminates instruction-level differences in library code, enabling page sharing across versions. Moreover, the same mechanism naturally extends to cross-library references, where functions or data from a versioned library are accessed by other libraries. In Figure 5.4b, calls to `f3` and `f3'` are rewritten to point to the trampoline table (e.g., `call f3@tpl` in the figure), which then redirects execution to the corresponding function implementation.

Spacer-Δ extends the original Spacer design to support memory sharing across multiple versions of the same library. As with changes in the set of deployed unikernels, the set of library versions present in the system may also evolve over time: new unikernel instances may be built against newer library releases, while existing in-



(a) *Without trampoline tables.* (b) *With trampoline tables.*

Figure 5.4: *Impact of instruction-level differences on page sharing. (a) Without trampoline tables, calls to different function versions (e.g., `f3` vs. `f3'`) generate divergent instructions that prevent page sharing. (b) With trampoline tables (.tpl), such problematic instructions are rewritten to a dedicated trampoline table, placed at the next page boundary, eliminating instruction-level differences in library code and enabling sharing across library versions and cross-library references.*

stances continue to rely on previous versions.

When a new library version is introduced, Spacer-Δ recomputes the placement of library code and its associated deltas to maximize page sharing across all known versions. This recomputation applies to newly built or redeployed unikernels, which adopt the updated layout and can therefore benefit from improved deduplication across versions. Unikernel instances that are already running, however, continue to operate with the layout determined at their deployment time and are not affected by the new mapping, which may temporarily limit sharing between older and newer instances. Nevertheless, as discussed in Section 3.5.2, older instances can be gradually replaced or restarted allowing the system to converge toward layouts that reflect the most recent set of library versions and thereby increase sharing over time.

5.3 Architecture and Implementation

This section introduces the architecture of the Spacer-Δ framework, which enables versioning. As in previous chapters, the framework leverages Unikraft [105], though the toolset could be adapted to other library OSes and unikernels with minimal modifications.

Spacer-Δ performs versioning at the level of object files rather than at the source or executable (ELF) level. Object files retain both symbolic and relocation information and can be seamlessly integrated into the linking process, which simplifies delta computation. In contrast, ELF binaries may be stripped and lack relocation details (already resolved at link time), making them significantly harder to analyze and process. Source files, although theoretically usable, were not considered be-

cause they are more difficult to handle. Achieving results comparable to object-file analysis would require a deep understanding of the language syntax and the ability to predict compiler behavior. Indeed, many differences at the source level could result in identical compiled code. For instance, changing variable names, refactoring to store intermediate results, or reordering independent instructions (e.g., `i++; j++;` instead of `j++; i++;`) can produce the same binary output.

Enabling versioning involves four distinct steps, as depicted in Figure 5.5. These steps are:

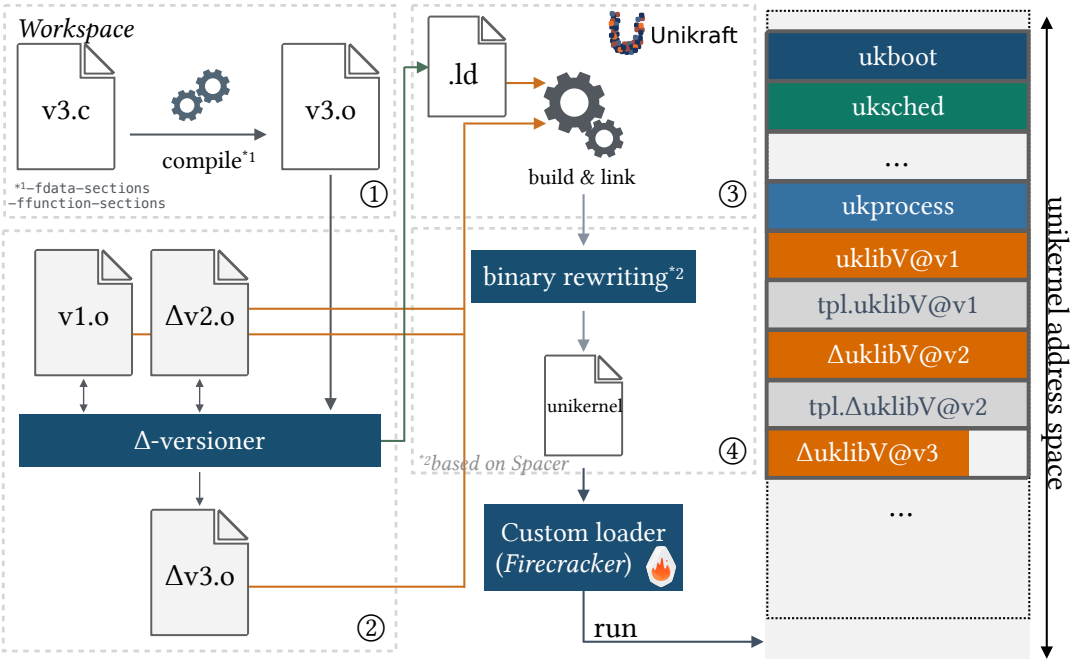


Figure 5.5: *Supporting versioning involves four key steps: ① Library object files preparation, ② Delta generation via a specific tool, ③ Unikernel linking, and ④ Binary rewriting to include trampoline tables. Finally, the unikernel is launched using Firecracker.*

① *Library object files preparation:* Similar to Spacer (see Section 3.3), our approach relies on object files, with the requirement that library code be compiled with section-level granularity flags (`-fdata-sections` and `-ffunction-sections`). These flags ensure that each function, relocation and data element is placed in a separate ELF section (instead of being aggregated), thereby simplifying subsequent processing. If object files were not compiled with these flags, their source(s) would need to be recompiled accordingly.

② *Delta generation:* The current version of the library object file (`v3.o` in the figure), together with all previous (delta) versions (kept in ELF format), is processed by the Δ-versioner. This tool generates a new object file (`Δv3.o`) containing the delta relative to earlier versions (`v1.o` and `Δv2.o`) and adjusts symbols when necessary (see Section 5.3.1 where we further explain its behaviour). To ensure consistent placement of libraries across versions, the Δ-versioner relies on Spacer's global map and produces a corresponding

linker script specifying alignment rules for the libraries, which is used in the next step. The Δ-versioner also copies each required object file into a specific workspace. For example, v1.o, Δv2.o, and Δv3.o are added to the workspace of the unikernel(s) using *uklib@v3*.

③ *Unikernel linking:* The unikernel ELF file is generated by the linker, which combines the newly created object file with all other object files. This process is guided by a linker script [66] that defines library placement.

④ *Binary rewriting:* The final step involves binary rewriting to isolate differing instructions into page-aligned trampoline tables. For this purpose, we rely on the binary rewriting tool developed in Spacer (see Chapter 3). By default, the tool rewrites instructions that reference addresses in other libraries or sections, redirecting them to trampoline table entries. In Spacer-Δ, we extend this mechanism to also handle instructions in libraries that call functions or access data in versioned libraries. These instructions are rewritten to use version-specific trampoline tables, ensuring that identical pages remain identical across instances using different library versions.

Spacer-Δ preserves the standard ELF format, ensuring full compatibility with existing VMMs such as Firecracker [2] and QEMU [29]. Spacer-Δ is also designed for straightforward integration into existing software pipelines. The delta-processing phase – including binary rewriting – is performed entirely offline during the build process, introducing no runtime overhead. This offline approach makes Spacer-Δ particularly well-suited for automated CI/CD workflows.

Finally, the tool operates at the object file level and requires no modifications to the underlying compiler or linker[1]. This design enables seamless integration with existing build systems. Further implementation details are discussed in Section 5.5.1.

5.3.1 The Δ-versioner Tool

As noted earlier, our design leverages backward compatibility with earlier library versions to support newer ones, representing the differences between versions as deltas at the object-file level. However, the object files produced by the linker contain complete symbol and section definitions, but no information about which functions or data are new or modified relative to earlier versions. To address this limitation, we developed a custom tool, the Δ-versioner, which operates on ELF-format object files compiled with section-level granularity flags. The tool is implemented in C++ and uses the ELFIO library [110] to parse and manipulate ELF files.

To generate a new object file containing the delta relative to earlier versions, the Δ-versioner processes the current version of the library object file together with all previously stored versions, following the steps outlined below:

[1] Supposing `-fdata-sections` and `-ffunction-sections` (or equivalent flags) are available.

1. *Parsing and processing object files:* The tool extracts ELF metadata, including symbols, sections, and relocations, from the current version of the library object file and all previous versions. Relocations indicate where references to symbols must be adjusted once their final addresses are known (see Section 2.2.3 for background on relocations).
2. *Mapping information:* For each function, the tool associates its corresponding symbol and relocation entries and identifies the content of the sections it manipulates (e.g., `.data`, `.rodata`). The use of section-level compiler flags simplifies this process, as each function resides in its own `.text` section with a corresponding `.rela.text` section containing its relocation entries. In contrast, traditional representations aggregate functions and relocation entries, complicating accurate mapping.
3. *Identifying new and modified functions:* Using the mapping information, the tool tracks changes across library versions. A function is classified as new if it does not exist in any previous version, and as modified if its content or attributes have changed. The comparison proceeds in stages: first, the tool checks function names and sizes from the symbol table. If two functions share the same name and size, their bodies (where relocations are not resolved) are compared using hash values. If the hashes match, relocation entries are inspected, and the function is considered modified only if the target symbol names differ (e.g., a call to `f2` changes to a call to `f3`). Finally, a function is also marked as modified if the content of any `.data`, `.rodata`, or `.bss` section it manipulates differs from previous versions.
4. *Adding unique elements to the new object file:* New and modified functions are added to the newly generated object file through the following steps:

   (a) The symbol table is updated with the name of each new or modified function and its associated section. Each new or modified symbol's binding is set as *global* (strong visibility, default value) [207], while symbols with the same name in prior object files have their bindings updated to *weak*, ensuring that the new global symbol overrides the previous one(s) during linking.
   (b) The content of the `.rodata`, `.data`, and `.bss` sections manipulated by the function (analyzed via the relocation entries) is added with the corresponding sections in the resulting object file.
   (c) All the relocation entries associated with the function are updated and added to the object file. During this process, fields such as *addend* and *value* [207] are recomputed to account for potential changes in offsets due to the addition/modification of content. We follow the same approach for handling relocation entries associated with the `.rodata` and `.data` sections.

5. *Object file generation*: When all previous operations are finished, the object file

containing the delta of all previous functions and data is generated.

Figure 5.6 illustrates a simplified ELF representation of object files across different versions. The source code is also shown to highlight the differences. In this example, the second version of *uklibV* is expressed as a delta of the first: the Δ-versioner tool computes this delta by adding only new or modified functions (`f2`, `stub`, and `f4`) to the symbol and section tables. The `.rodata` section is updated to include their associated data, and relocation entries (illustrated here only for `f2`) are updated accordingly.

Before linking these two object files with others (such as the ones of other versions) to produce the final ELF file, it is necessary to update symbol bindings for symbols that have been modified between versions. Retaining multiple symbols with a strong binding would lead to redefinition errors during linking. To circumvent this, each modified symbol in the latest library version is marked as *strong* (global), while symbols with the same name in earlier object files are changed to *weak*, as illustrated with `f2`. This adjustment is performed by the Δ-versioner tool in each specific workspace, using the versions of libraries required by that unikernel instance.

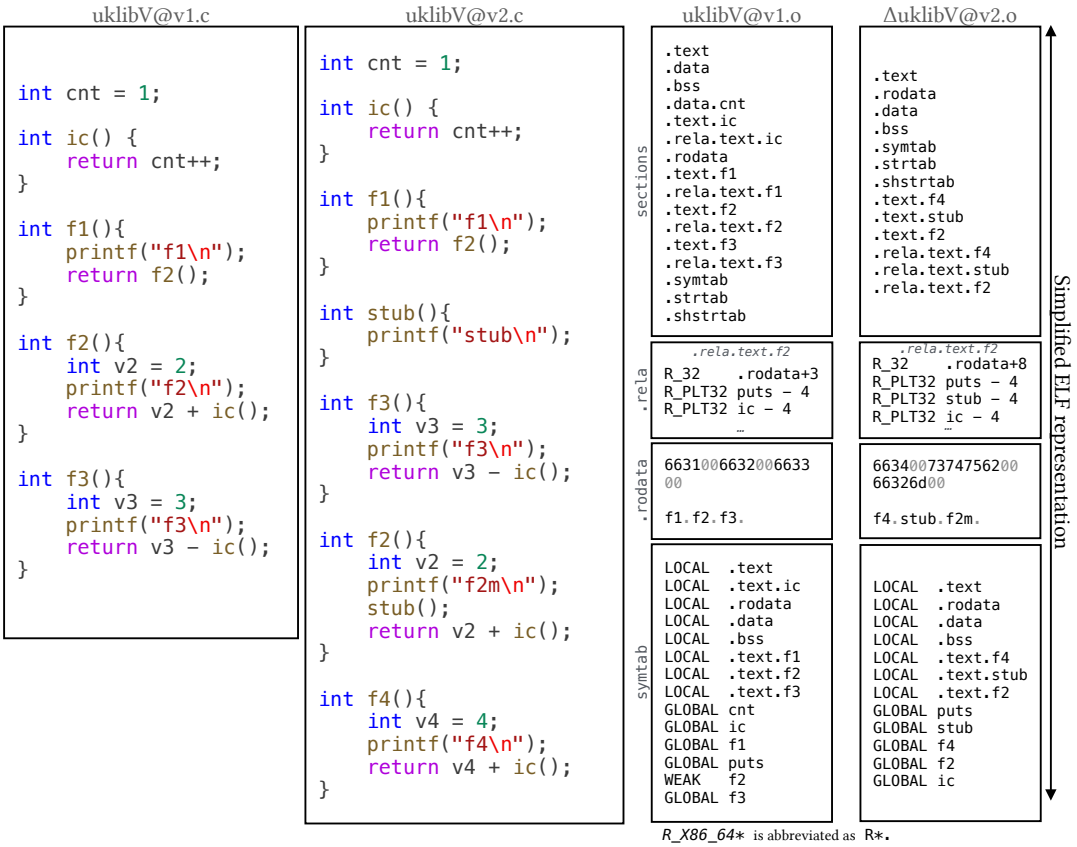


Figure 5.6: *Example of a simplified ELF representation of a versioned library. The second version of uklibV is expressed as a delta of the first. Only new or modified functions (`f2`, `stub` and `f4`) are added to the symbol and section tables.*

5.4 Evaluation

This section presents the experimental results, highlighting the memory savings achieved through our versioning approach, Spacer-Δ. Experiments were conducted using two different memory deduplication mechanisms: (1) Firecracker with KSM, using the default mode (*ksm-quiet*) that scans 100 pages every 20 ms [11], and (2) our load time memory deduplication custom loader (explained in Chapter 4).

We evaluated memory consumption, ELF binary size, and performance using both traditionally compiled applications ported to unikernels and unikernels specifically designed for FaaS workloads. The evaluation considered six configurations: (1) the default Unikraft setup without optimizations; (2) Dead Code Elimination (DCE) [196]; (3) Spacer combined with Kernel Samepage Merging (KSM); (4) Spacer-SLT (Share at Load Time), relying on a custom loader and a libraries pool; (5) Spacer-Δ combined with KSM; and (6) Spacer-Δ-SLT (Share at Load Time).

Configurations (3) and (4) rely on the default Spacer approach (introduced in Chapter 3), which treats different library versions as distinct libraries. In contrast, configurations (5) and (6) adopt the approach described in Section 5.2.3, representing new library versions as deltas relative to previous ones.

For certain experiments, we also present the ASLR variants. To implement ASLR, we used the same approach as before (see Section 3.3.1), randomizing the memory layout at link time via linker scripts. Unlike vanilla versions, which include trampoline tables only for libraries that request services from versioned libraries, ASLR approaches require a dedicated trampoline table for each library. This results in an increased number and larger overall size of trampoline tables.

To manage different library versions, we relied on Git tags [77], where each library release is associated with a specific tag. The extent of changes between versions varies widely depending on the developers' contributions: a new release may introduce substantial modifications, such as a restructured code architecture, or minimal adjustments, such as minor bug fixes. For each library, we cloned its GitHub repository [78], analyzed successive versions in chronological order, and excluded intermediate versions that introduced no code changes. For every version, we built a unikernel using the Default, DCE, and Spacer configurations, and additionally generated Spacer-Δ variants using the Δ-versioner tool (Section 5.3). Table 5.1 summarizes the tested libraries and their corresponding commits.

Library	commits (oldest to latest)
lib-sqlite [199]	6b54e32, fc44ea1, 2c6d801, 1da038f, 9927df2, 8dbe27e, d87000c, 60d9e2a
lib-pcre [90]	09fb6ce, 986d5c5, 1e0fcfc, 25c72e6, 2d6b260
lib-python [203]	a5f8ef1, 5900336, 04fad4f, dc93f53, 2d070a4
lib-nginx [163]	0febe9a, 2eedb3f, 3229ec6, 6c5955f, 9cbe052
lib-pthread [202]	49a2433, 2dd7129, 955a702, bf7c1f6, e2705f9d

Table 5.1: *Versioned libraries and their respective commits (cloned from Github [78]).*

Our evaluation aimed to address the following research questions: (1) In Section 5.4.1, how effective (in terms of memory usage) is Spacer-Δ compared with conventional approaches – including Spacer, which treats different library versions as independent libraries placed at distinct virtual addresses – that do not efficiently manage library versioning in statically linked unikernels? (2) In Section 5.4.2, what is the impact of Spacer-Δ on disk usage? (3) In Section 5.4.3, how does Spacer-Δ affect runtime performance?

To answer these questions, we assessed overall memory usage (unikernel + hypervisor), disk usage, and performance metrics such as total execution time. All experiments of this chapter were conducted on a Debian 11 (bullseye) server using a Linux kernel 6.1, gcc 10.2.1 (GNU ld 2.35.2 as linker). The host machine used for the experiments has 32 GB of RAM and an Intel® Xeon® CPU (E5-2650 v2 @ 2.60 GHz) with 32 cores. In addition, Unikraft Enceladus 0.8.0 (with Firecracker support [209]) has been used for the experiments.

5.4.1 Memory Usage

The first experiment evaluates whether Spacer-Δ reduces memory consumption compared with four configurations: DCE (KSM), Default (KSM), Spacer (KSM), and Spacer-SLT. To this end, we developed a web server that interacts with SQLite and ported it as a unikernel. We then linked this application against successive versions of *lib-sqlite* [199], resulting in eight unikernels having different versions of SQLite. For this experiment, we selected *lib-sqlite* as the target library because it represents a practical middle ground: it is neither trivial in size (e.g., a single-function library) nor excessively large (e.g., a language runtime such as Go [197]).

Figure 5.7 reports the average total memory usage across 30 repeated runs. As shown in the figure, a new unikernel instance is launched every 10 seconds starting at time 0. This creates an incremental scaling scenario in which additional instances are continuously added while existing ones remain active. For each KSM configuration, the launch of a new instance induces a temporary memory peak, which gradually decreases as the memory scanner identifies and merges identical pages.

For vanilla configurations, Spacer-Δ (KSM) achieves better memory savings than its counterparts. Averaged over the full experiment duration (80 seconds), Spacer-Δ uses 1.8× less memory than DCE (KSM), 2× less memory than Default (KSM), and 1.7× less memory than Spacer (KSM). In contrast, Spacer-Δ-SLT merges memory pages directly at load time through the custom loader and shared library pool, thereby eliminating the transient peaks observed in KSM-based setups. However, because this configuration intentionally merges only read-only pages (code and read-only data) to mitigate CoW attacks [231, 189, 32, 22], its overall memory consumption is slightly higher than that of Spacer-Δ (KSM).

Similar trends hold for ASLR-enabled configurations, although average memory usage is higher across all setups. This increase is primarily due to reduced sharing

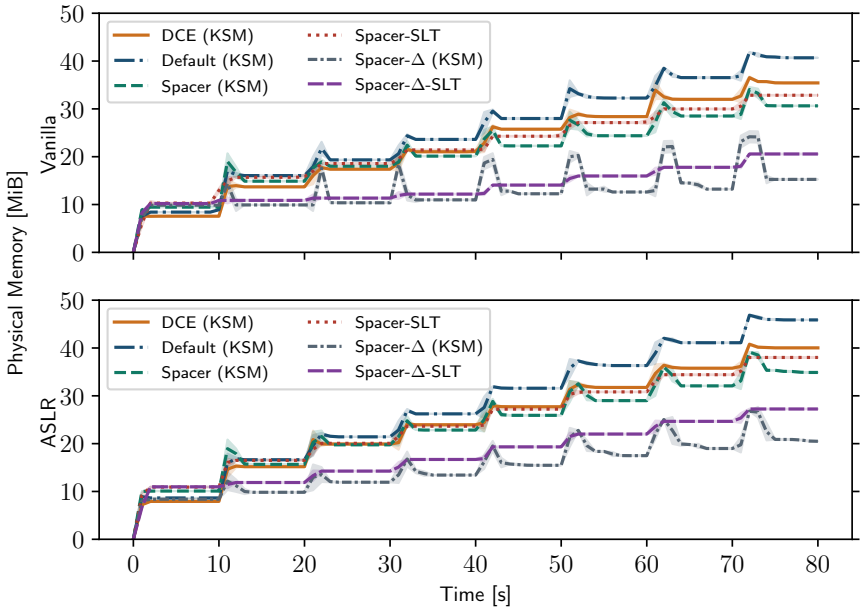


Figure 5.7: *Evolution of the average physical memory usage of 8 web server unikernels (+ hypervisor). Each instance uses a different version of lib-sqlite and is launched every 10 seconds (starting from $t = 0$). Compared with other configurations, Spacer-Δ can result in a reduction up to 2× in memory consumption.*

opportunities. In the Default and DCE configurations, ASLR causes code pages to diverge as libraries are mapped at randomized addresses. For Spacer and Spacer-Δ, the additional overhead arises from trampoline tables and non-shareable read-only data (`.rodata`), since library shuffling introduces version-specific relocations. Nevertheless, even under ASLR, Spacer-Δ (KSM) still achieves average memory reductions of 1.5× compared with DCE (KSM) and Spacer (KSM), and 1.8× compared with Default (KSM).

5.4.2 ELF File Size

We next evaluated the effect of versioning on unikernel size by analyzing the generated ELF files. The experiment was conducted using the same unikernels and libraries described in the previous section. Figure 5.8 presents the total disk footprint obtained by summing the ELF sizes of all eight unikernel instances for the DCE, Default, Spacer, and Spacer-Δ configurations, along with their respective ASLR variants.

We first analyze disk usage for configurations that do not rely on a custom loader. Notably, Spacer-Δ (without SLT) significantly increases the disk footprint. This growth stems from embedding multiple versions of the same library into a single binary, a design choice that enables efficient memory sharing across versions. As a result, ELF file sizes grow substantially. Compared with DCE, which aggressively prunes large monolithic libraries (e.g., `libc`), as well as with the Default and Spacer configurations, Spacer-Δ exhibits file size increases of 1.9×, 1.6×, and 1.5×, respectively. ASLR at link time introduces additional overhead. For the DCE and Default

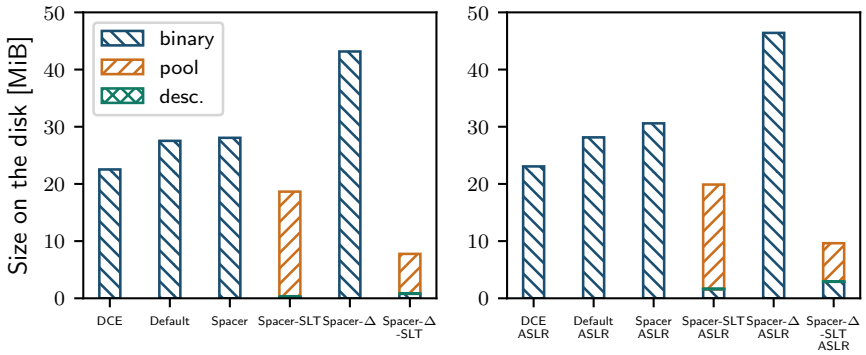


Figure 5.8: *Aggregated disk space occupied by eight web server unikernel instances, each linked against a different version of SQLite, under the evaluated configurations. Spacer-Δ without a custom loader increases the total disk footprint, while combining Spacer-Δ with a custom loader and a shared library pool substantially reduces the aggregated ELF size.*

configurations, this overhead stems from our ASLR implementation based on linker scripts: instead of consolidating all `.text` sections into a single entry, each library retains its own section entry. As a result, the section header string table grows, increasing the overall ELF size. For Spacer (ASLR) and Spacer-Δ (ASLR), this growth is further amplified by the presence of additional and larger trampoline tables compared to the vanilla versions.

We now turn to configurations that rely on a custom loader and a common library pool (denoted as SLT). As discussed in Chapter 4, using a custom loader enables substantial reductions in ELF file size by extracting common libraries (code and read-only data) into a shared pool, rather than embedding them into each unikernel binary. Compared with Spacer-SLT, Spacer-Δ-SLT further reduces disk usage by a factor of 2.4×. This improvement stems from the different treatment of library versions: Spacer-SLT considers each library version as a distinct library and therefore stores them independently in the library pool, increasing its size. In contrast, Spacer-Δ-SLT represents successive library versions as deltas of a common base, significantly reducing redundancy in the shared library pool. Although this results in a smaller pool, Spacer-Δ-SLT additionally requires storage due to the presence of trampoline tables to handle instruction-level differences between versions, whereas Spacer-SLT only stores `.data` sections as binaries (see Section 4.3.1). In vanilla configurations, Spacer-Δ-SLT achieves file size reductions of 2.9×, 3.5×, 3.6×, and 5.5× relative to DCE, Default, Spacer, and Spacer-Δ without SLT, respectively. Similar trends are observed under ASLR, though the reductions are less pronounced due to the overhead of trampoline tables and unshareable `.rodata` segments.

5.4.3 Performance

We conducted a series of performance experiments, using total execution time as the primary metric. This encompasses all stages of a unikernel's lifecycle – create, boot, run, shutdown, and destroy. Our objective is to evaluate how Spacer-Δ, in

both KSM- and SLT-based configurations, affects overall execution time compared with traditional statically linked unikernels, which provide no versioning support.

For the following experiments, we selected six applications ported as unikernels, each tested across five successive versions of a specific library. To capture different runtime behaviors, we divided them into short-lived and long-lived workloads. In the descriptions below, the versioned library associated with each unikernel is indicated in parentheses.

Three of the applications are short-lived. (1) Nginx (lib-nginx): a modified Nginx with full lwIP support, terminated just before the `accept()` call to emulate an ephemeral unikernel. (2) Perl Compatible Regular Expressions (PCRE) (lib-pcre): a PCRE-based unikernel that processes a 6 MiB text file and returns matches for specific patterns. (3) Python FaaS sorting (lib-python): a Python-based FaaS unikernel that sorts a list of 64 elements and returns the result in JSON format.

The remaining three applications are long-lived. (4) Matrix multiplier (lib-pthread): a parallel 1500×1500 matrix multiplier that stores the result to a file. (5) SQLite bench (lib-sqlite): a benchmark involving 60000 SQL insertions [194] and running entirely in memory (via RamFS). (6) Python Mandelbrot (lib-python): a Python function that generates a 1920×1080 Mandelbrot and saves it to a file. These workloads stress memory usage differently. For instance, the SQLite benchmark allocates large amounts of memory with a high degree of intra-sharing (self-sharing), mainly due to its initial in-memory table structures. The matrix multiplier, in contrast, exhibits minimal intra-sharing but still allocates heap memory for computation.

Two scenarios were considered for each unikernel: (1) a standalone scenario, where each unikernel instance, embedding its own library version, is executed individually (no previous instances remain active), and (2) a cumulative scenario, where new unikernel instances with successive library versions are added one after another while previous instances remain active (modified to stay idle if their execution is short). The newly added instance is the one being benchmarked. We used the perf [85] tool to measure performance on the benchmarked instances, repeating the experiments 30 times. The results can be observed in Figures 5.9 and 5.10, where we isolated a single CPU core using `isolcpu` and ran the benchmarked instance while pinning it to that core.

Figure 5.9 presents the performance results for the standalone (single-instance) scenario. Spacer-Δ (KSM) introduces a slight execution-time overhead compared with DCE (KSM), Default (KSM), and Spacer (KSM), both with and without ASLR. This overhead stems from increased page faults caused by the extra pages required for trampoline tables and backward-compatibility symbols. ASLR also introduces a minor penalty for DCE (KSM) and Default (KSM), as it shuffles libraries and randomizes their memory addresses (see Section 3.4.4). Among KSM-based configurations, DCE achieves the fastest execution times because it loads the fewest pages.

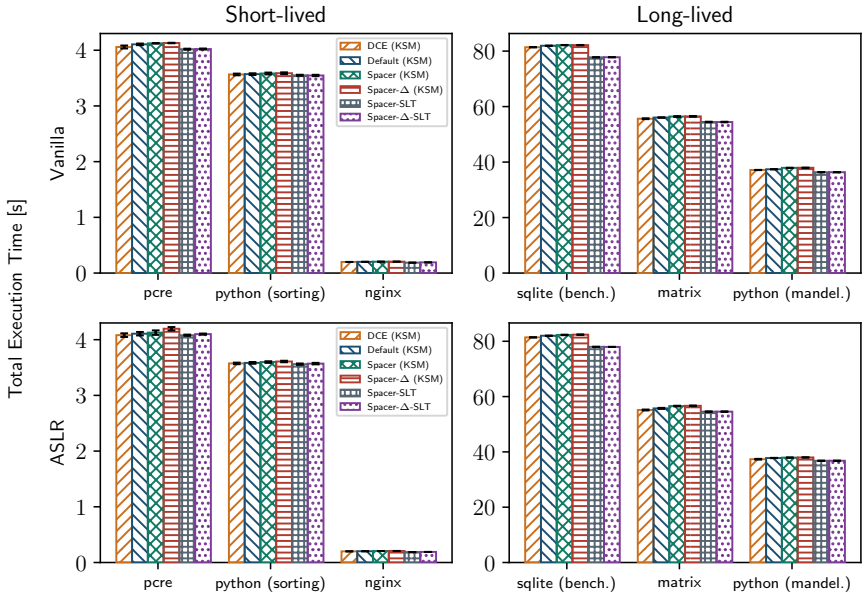


Figure 5.9: *Total execution time (seconds) of several applications using five different versions of a specific library, executed in standalone mode. Spacer-Δ achieves performance similar to Spacer.*

Spacer-Δ-SLT provides better execution time compared with all KSM-based setups by preloading code and read-only data into a library pool, thereby avoiding the overhead of fully parsing the ELF file and loading the unikernel into memory. With ASLR, however, larger trampoline tables and non-shareable `.rodata` sections slightly diminish these gains. Even with a single instance, KSM can still scan and merge pages by identifying self-sharing – identical pages within the same unikernel. Under the ksm-quiet configuration, these effects are most pronounced in long-lived workloads. This is especially observable in the SQLite benchmark, which initially creates many identical writable pages for its internal table structures. KSM merges these pages, but as execution proceeds and they diverge, frequent CoW faults occur, introducing non-negligible performance penalties. In contrast, Spacer-Δ-SLT avoids the merging of writable pages altogether, thereby eliminating KSM overhead and improving execution time. In our evaluation, it reduces execution time by up to 7%, 9%, 11%, and 13% compared to DCE (KSM), Default (KSM), Spacer (KSM), and Spacer-Δ (KSM), respectively. The same trend persists under ASLR, although with a small additional overhead.

Finally, we observe no significant performance difference between Spacer-SLT and Spacer-Δ-SLT, as both rely on a library pool and perform page deduplication at load time, resulting in a maximum overhead of 0.2%. Spacer-Δ-SLT is occasionally slightly slower due to a higher rate of page faults and the use of trampoline tables; however, this overhead remains negligible across all benchmarks.

Figure 5.10 presents the performance results for the dynamic (multi-instance) scenario. For long-lived unikernels, we observe more pronounced variability in total execution time under KSM-based configurations compared with the single-instance scenario. This arises because the number of pages to be scanned increases

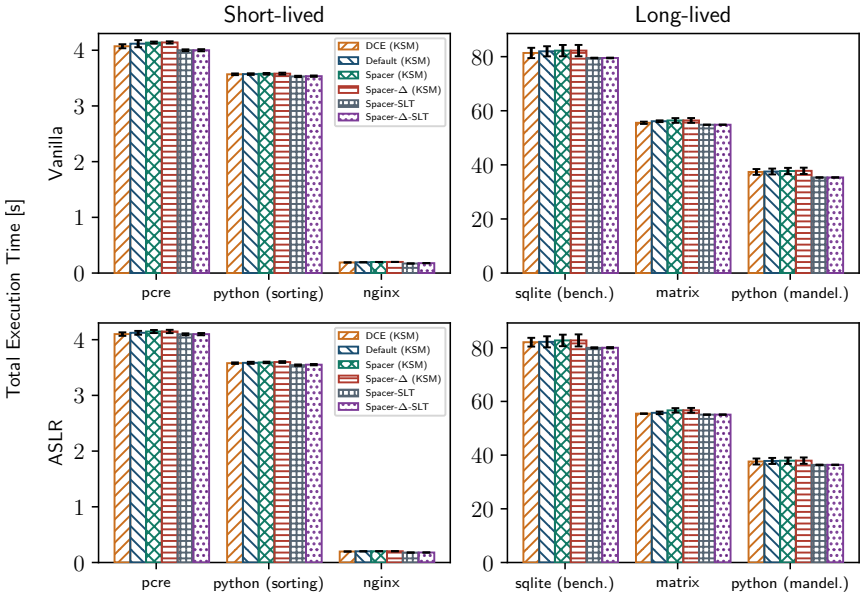


Figure 5.10: *Total execution time (seconds) of several unikernels using five different versions of a specific library, executed with previous versions on different cores. Spacer-Δ achieves performance similar to Spacer.*

as additional unikernel instances run concurrently. Due to the ksm-quiet configuration, which prioritizes CPU efficiency over efficient memory reduction, later-launched instances experience less interference from KSM, mitigating its impact on performance. This effect is especially observable for the SQLite benchmark, which allocates memory for all its inner benchmarks, resulting in a large number of pages to scan. For short-lived unikernels, the impact of KSM remains fairly low, and the variability in execution time is minimal, resulting in observations that are essentially identical to those seen in the single-instance scenario.

In conclusion, Spacer-Δ enables efficient memory sharing, improving memory reduction while preserving execution times comparable to Spacer. When combined with a custom loader that performs memory deduplication at load time, it further minimizes file size and improves performance.

5.5 Discussion

This section discusses various technical aspects of our implementation and explores potential adaptations to existing systems.

5.5.1 Integration and Orchestration

Our system operates directly on relocatable ELF object files, maintaining compatibility across compilers that support the fine-grained separation of functions and data into individual sections (e.g., via `-fdata-sections` and `-ffunction-sections` or equivalent flags). Because object files are a standard output of existing compilation workflows, this design avoids altering compiler or linker internals and fits nat-

urally into diverse build environments. The only requirement is that the compiler produce separately identifiable functions and data, a capability already available in mainstream toolchains such as GCC [217] and Clang [123].

Operating at the compiler stage, for example, by processing individual source files and computing version deltas during compilation, can offer tighter integration and potentially reduce processing steps. However, this approach typically requires compiler plugins, custom toolchain builds, or – in some designs – modified linkers, all of which introduce long-term maintenance burdens and compatibility risks across platforms and compiler versions. By contrast, operating at the object-file level preserves fine-grained control while avoiding the adoption barriers associated with toolchain modifications.

5.5.2 Dynamic Library Versioning

While dynamic linking provides transparent memory sharing between processes in traditional operating systems, it is generally less applicable and less common in unikernel environments. Unikernels are typically statically linked by design to ensure strong isolation and fast startup [106, 105, 198, 148, 111, 97]. Although some unikernels support dynamic linking [198, 226], it introduces additional runtime complexity, overhead, and potential security risks. Furthermore, each version of a shared library is loaded independently, with the loader treating ELF shared libraries as complete binaries; there is no mechanism to apply deltas between versions.

In contrast, our approach preserves the benefits of static linking while enabling memory sharing across different library versions through fine-grained differential analysis and page-level alignment – avoiding full copies of each library version in memory and significantly reducing memory usage.

5.6 Related Work

This section provides an overview of related work in the fields of versioning, unikernels, memory deduplication, and dynamic software updates. Additionally, it discusses how these existing approaches differ from our proposed methodology.

Unikernels and memory deduplication. Unikernel projects such as Unikraft [105] and HermitCore [111] have explored static linking optimizations, but do not provide systematic solutions for handling library versioning or deduplicating libraries across multiple instances. Library alignment techniques, such as Spacer [73], improve memory deduplication by aligning library code at fixed addresses across unikernels. However, Spacer focuses on memory layout optimization and does not address broader versioning challenges, such as API evolution, backward compatibility, or multi-version co-existence. Some approaches attempt to reduce memory usage by employing CoW during forking. For instance, KylinX [238] introduces

pVM (process-like VM) fork along with a set of APIs for inter-pVM communication, enabling reduced memory consumption when creating unikernels. Nephele [125] proposes a cloning solution for unikernels/VMs that supports both memory and I/O device cloning. Both approaches are based on Xen [26], require adding hypercalls to the hypervisor, and do not address versioning issues. Some unikernel projects [226] dynamically load the unikernel similarly to how dynamically loaded libraries are handled. However, they do not address the challenges of versioning in statically linked unikernels. IncludeOS [34] introduces a "LiveUpdate" feature to enable updates with minimal downtime. However, this approach does not address memory deduplication and fails to reduce memory usage when multiple instances are running concurrently. Similarly, methods such as SEUSS [39] and SAND [4] adopt a snapshot-and-restore approach from an initial snapshot by overwriting memory space but without explicitly dealing with broader versioning challenges.

Dynamic software updates. Ksplice [14] employs 'pre-post differencing' and 'run-pre matching' techniques. These methods generate replacement code, resolve symbols, and verify the security of updates, all based on object code. However, Ksplice is designed for general-purpose operating systems and thus relies on support for dynamically loadable kernel modules – a feature available in modern OSes but absent in unikernels due to their single-application design. Other dynamic software update solutions have been proposed, such as MVEDSUA [154] and Jvolve [186], but most of these projects do not consider statically linked unikernels and memory deduplication as their primary objectives.

5.7 Summary

In this chapter, we explored the challenges of managing library updates and versioning in statically linked unikernels. We identified that, due to their compact structure and lack of built-in versioning, even minor library updates can disrupt memory layouts, increasing memory consumption when multiple instances run concurrently.

To address these challenges, we introduced Spacer-Δ, a framework designed to enable efficient library versioning while improving memory usage in statically linked unikernels. By leveraging library alignment and a novel representation based on differential analysis between library versions, Spacer-Δ preserves memory layout consistency and enables efficient page-level sharing. When combined with memory deduplication, the framework effectively reduces overhead in environments where unikernels use different versions of the same libraries.

Our evaluation demonstrated that Spacer-Δ, especially when relying on load time memory deduplication, can achieve improvements in execution time, memory usage, and disk footprint, proving its practicality and efficiency. Its straightforward

integration with Unikraft and minimal adaptation needs for other unikernel frameworks make it a practical and versatile solution for cloud-native workloads.

5.8 Potential Improvements

Handling additional languages. Currently, our system focuses exclusively on the C programming language. We conducted several tests to validate its functionality within this scope. Future work will extend support to additional compiled languages, such as C++, Go, and Rust.

Supporting standard object files. Our current implementation assumes that object files are compiled with section-level granularity flags (`-fdata-sections` and `-ffunction-sections`), which simplifies symbol-to-function mapping and relocation tracking. However, many off-the-shelf libraries are distributed without these flags in their build system. A natural extension of our work is to support such standard object files by applying finer-grained binary analysis to distinguish functions and associated data from aggregated sections.

Dynamic software updates. A potential research direction is enabling runtime updates of unikernels by applying memory-level deltas computed by Spacer-Δ. This approach could allow live, in-memory patches, supporting seamless and adaptive updates while preserving system state.

This page intentionally left blank.

Part III

System Call Analysis & Unikernel Porting

This page intentionally left blank.

6

PORTING TO UNIKERNELS

Overview

This chapter addresses the practical challenge of porting existing applications to unikernel environments, using Unikraft as the primary research and development platform. Unlike previous chapters, which focused on memory deduplication and system-level research, this chapter adopts an engineering perspective. We present the methodology and tools we developed to reduce porting effort and enhance the practical adoption of unikernels in production environments.

We begin by examining the necessity of identifying system calls in traditional applications and the challenges inherent in developing robust system call detection tools. To this end, we design a hybrid analysis approach that combines static and dynamic binary analysis to infer an application's operating system dependencies, compensating the limitations of each technique when used alone. These tools form the foundation of a broader toolchain that streamlines the unikernel porting process by identifying the libraries that implement the required system calls. Our tools have contributed to several research efforts and publications, including *Unikraft* [105], *Loupe* [117], and *Unikraft and the Coming of Age of Unikernels* [118]. All the tools developed as part of this chapter are open source and available on Github [70, 71].

Author's Contributions. The results presented in this chapter are derived from the Application Support and Porting section of the Unikraft paper [105] and were carried out entirely by me, with the exception of Experiment 6.1, which is included for clarity. My contributions include the design and implementation of the static and dynamic binary analysis tools, the toolchain that streamlines the unikernel porting process, the porting of `musl` (base support) to Unikraft, and the associated experimental evaluation.

6.1 Introduction

Unikernels have long promised significant advantages, including high performance, ultra-fast boot times, and strong security benefits due to a smaller attack surface [106, 101, 34, 127, 111, 97]. Yet, despite these benefits, they have not achieved widespread adoption in the software industry. Building a high-performance unikernel remains labor-intensive, often requiring expert knowledge to select and integrate the appropriate components of a library operating system, with limited tooling support. Moreover, most unikernel projects lack full POSIX compliance [148], complicating the execution of existing applications without significant modification. As a result, porting applications to unikernels remains difficult due to complex build systems, limited tooling, and incomplete OS interfaces, confining their use largely to niche scenarios.

In this chapter, we focus on the fundamental reason these challenges persist: the absence of systematic, automated methods to bridge the gap between existing applications and unikernel environments. The Unikraft framework was designed to alleviate this problem by decomposing the OS into fine-grained, reusable libraries, enabling developers to assemble unikernels tailored to specific applications [105, 201]. However, effectively using Unikraft still requires identifying the set of system calls an application may invoke and determining which libraries provide their implementations – tasks that Unikraft cannot automate on its own. Without dedicated tooling, developers must manually detect system calls, select the appropriate libraries, and integrate them into the build. Omitting a required system call may cause runtime failures because its implementation (in the corresponding library) is missing, whereas including unnecessary calls increases the unikernel's codebase and thus the attack surface.

Several techniques exist for identifying system calls, but each has inherent limitations [91, 120, 143, 219, 49, 152, 38]. Dynamic analysis observes system calls during program execution but depends on the quality of test coverage and can miss unexercised code paths. Static analysis, on the other hand, examines code without execution and can detect potential calls across multiple control-flow paths. However, it often struggles with challenges such as indirect calls, dynamically loaded libraries, or obfuscated code, which can lead to missed or spurious results. To reduce these shortcomings, we adopt a hybrid methodology that combines static and dynamic binary analysis on statically and dynamically linked executables. Our approach prioritizes robustness and coverage: missing a required system call can lead to runtime failure, whereas including a few unnecessary calls is acceptable. To perform a comprehensive analysis, we developed two complementary tools: StaticBinSys performs static binary analysis by reconstructing call graphs and using symbolic backward exploration [181] to infer potential system calls even in stripped binaries, while DynBinSys executes dynamic tracing to capture system and library calls under diverse workloads. We also discuss and address several key challenges (such as

indirect calls) that must be considered when developing system call identification tools.

As part of our contribution to the Unikraft ecosystem, we used these tools to analyze the system call usage of 30 widely-used server applications [50] and compared the results against the system calls currently implemented in Unikraft. Our analysis shows that only a subset of Linux system calls is required to support these applications, and that Unikraft already implements the majority of the system calls they actually exercise.

We also explore several strategies for porting applications to Unikraft, each representing a different trade-off between development effort and performance. At one end of the spectrum, source-based porting requires adapting an application's source code to Unikraft's build system and APIs. While this approach allows fine-grained optimization and tight integration, it demands substantial manual effort and deep system expertise. At the other end, binary compatibility [148] executes existing binaries with minimal or no modification, reducing developer effort. However, Unikraft support was limited at the time (no support for dynamically linked executables) and it often resulted in a significant performance penalty.

To address this gap, we adopt an intermediate strategy that leverages the application's external build system. In this approach, we reuse externally compiled relocatable object files and link them directly into the Unikraft build. This method retains much of the efficiency of source-level integration while significantly simplifying the porting process. To make this approach practical, we extended the Unikraft ecosystem with two complementary contributions: (1) a port of the musl libc [61], enhancing POSIX compliance and expanding application compatibility, along with a glibc compatibility layer to handle specific glibc [68] symbols; and (2) UkTools, a framework that semi-automates the porting workflow by identifying required symbols, matching them to Unikraft libraries, generating build configurations, and integrating externally compiled object files. We demonstrate that our contributions form a cohesive pipeline that enables the successful compilation and linking of all tested libraries and applications.

Our main contributions are as follows: (1) A detailed analysis of the challenges in system call identification for unikernel porting, highlighting the inherent limitations of both static and dynamic analysis. (2) The design and implementation of two system call identification tools, StaticBinSys and DynBinSys, enabling more robust detection of system calls. (3) An empirical evaluation of application compatibility with Unikraft, providing quantitative data on system call footprints for real-world software. (4) The integration of musl libc along with a glibc compatibility layer, significantly enhancing POSIX compliance and expanding application support. (5) The development of UkTools, a command-line tool that reduces manual intervention during porting.

6.2 System Call Identification

Porting traditional applications to unikernels presents unique challenges, as these applications are generally developed for POSIX environments and depend on a wide range of system calls. In the context of unikernels, which aim to minimize the execution environment to the smallest set of required components, it becomes crucial to identify which system calls an application actually invokes. This information is necessary to determine either (i) which parts of the operating system interface must be implemented or (ii) which specialized libraries should be included during the unikernel porting process. Without accurate detection, the resulting unikernel may either crash at runtime because required system calls are omitted, fail during linking (e.g., due to missing system call wrapper implementations), or unnecessarily bloat by including unused functionality.

In this context, our goal was to investigate how system calls performed by applications can be systematically detected. To this end, we explored several techniques, identified the challenges inherent in each approach, and developed dedicated tools capable of retrieving the set of system calls that may be executed by an application.

6.2.1 Techniques to Identify System Calls

Two main approaches exist for identifying the system calls that may be invoked by an application: static analysis and dynamic analysis.

Static Analysis

Static analysis examines a program without executing it. In the context of system call detection, this entails analyzing the program's code – at the source, intermediate representation, and/or binary (assembly) level – to identify the instructions that could trigger system calls. Its main advantage is the ability to conservatively capture the set of system calls that *may* be invoked at runtime, since the entire codebase is inspected independently of specific inputs, workloads, or execution paths, including those that arise only in rare error conditions.

A key limitation is its difficulty in capturing dynamic behaviors, such as indirect call invocations through function pointers or dynamically loaded modules, which may lead to missed system calls. Furthermore, its comprehensive nature introduces the risk of *false positives*. For instance, analyzing unreachable code (e.g., considering all execution paths in a function when only a subset are ever taken) can inflate the predicted set of system calls, overestimating those actually exercised during execution.

Static analysis can be conducted at different abstraction levels, ranging from source code to binary executable:

1. *Source analysis*: Source-level analysis examines the program's logic directly from its source code to identify system call invocations. Nevertheless, even

with full access to the codebase, this approach is labor-intensive and programming language-dependent, as it requires manually inspecting various constructs such as language-specific abstractions. Furthermore, it relies on the availability of complete sources, which is often not guaranteed, especially for third-party components.

2. *Binary analysis*: Analyzing binary code offers the advantage of being independent of the programming language and enables analysis even when the source code is unavailable. Nevertheless, it introduces challenges, such as architecture dependence, potential code obfuscation, and difficulties in tracking library calls.

Dynamic Analysis

Dynamic analysis captures the actual system calls invoked by an application during runtime under a specific workload, typically by monitoring interactions between the application and the system call API [210, 185]. A major drawback of this approach is the difficulty of achieving complete coverage, which leads to *false negatives*: system calls not invoked during testing with a given workload may be misclassified as unnecessary, even though they may be essential under different execution conditions. Furthermore, dynamic analysis is also harder to fully automate since manual intervention is often required for tasks such as selecting different representative workloads.

Additional limitations include the necessity of executing the program under analysis – thereby implicitly trusting it – as well as the potential presence of anti-reversing techniques designed to alter behavior when monitoring is detected.

6.2.2 Static or Dynamic Analysis?

Detecting all system calls invoked by an application is thus inherently challenging. As discussed earlier, both static and dynamic analysis present intrinsic limitations that prevent complete coverage. No single method can guarantee that all possible system calls used under every execution path will be discovered. To mitigate this, we adopt a hybrid approach that combines static binary analysis with dynamic tracing.

Static analysis provides a broad exploration of potential execution paths, capturing system calls that may occur. This phase ensures comprehensive coverage but may include calls that are never actually executed. Dynamic analysis complements it by observing real executions and capturing system calls that may be missed statically – such as those triggered through dynamic code loading, or runtime-dependent conditions. Static source-level analysis was not considered because it does not scale to large collections of applications and is infeasible when source code is unavailable (as with proprietary or third-party libraries).

This design choice reflects a pragmatic engineering philosophy rather than a theoretical guarantee. In the context of unikernel porting, missing even a required system call *may* prevent an application from running, whereas including unnecessary ones merely enlarges the supported interface. Our hybrid approach therefore aims to favor broad coverage over strict interface minimality, accepting slight overestimation as the necessary cost of robustness.

We developed two dedicated tools to implement our hybrid methodology. The first, StaticBinSys, performs static binary analysis to identify potential system calls within the binary itself and its linked shared libraries when dealing with dynamically linked executables. The second tool, DynBinSys, complements this analysis by executing the program under diverse workloads and configurations, while tracing system calls with tools such as strace [185]. These are command-line tools and are both open-source [70, 71]. In the following sections, we discuss in detail the challenges encountered when developing these tools and the strategies adopted to reduce their impact.

6.2.3 Static Binary Analysis: Challenges and Implementation

Static binary analysis for identifying system calls presents several fundamental challenges. These arise from the nature of compiled code, the structure of the Executable and Linkable Format (ELF) [207], and the architecture-dependent mechanisms used to invoke system calls. This section describes the key challenges encountered during the design and implementation of our static analysis tool.

Challenge 1: Dealing with binaries. The ELF file format is designed to support multiple linking strategies (e.g., static vs. dynamic linking), a feature that significantly increases the complexity of its parsing and subsequent analysis. Dynamic linking, in particular, introduces indirection through the Procedure Linkage Table (PLT) and the Global Offset Table (GOT), where call targets are resolved only at load time or runtime (see Section 2.2.4). Further complications arise from the conceptual distinction between sections (the linker's abstraction) and segments (the loader's abstraction). Metadata required during the linking phase (e.g., symbol tables) is frequently stripped in release binaries, thereby limiting the information available to static analysis tools.

Another source of complexity stems from library dependencies resolution in dynamically linked executables. The location of shared libraries is not hardcoded within the ELF object but is instead determined at runtime through a combination of search paths and environment variables, complicating dependency inference via static inspection. On Linux systems, for instance, library resolution is delegated to the dynamic loader (`ld.so`) [113], whose behavior is influenced by environment variables such as `LD_PRELOAD` and `LD_LIBRARY_PATH`.

Challenge 2: Disassembly. A foundational step of many binary analyzers is disassembly, the process of translating machine code into a sequence of assembly instructions. Obtaining a complete and precise disassembly of arbitrary binaries is formally undecidable [221, 222]. The difficulty arises mainly from two factors: (a) the ambiguity in distinguishing code from data [222], and (b) the difficulty of reconstructing function boundaries [25].

Challenge 3: Identifying reachable library functions. For dynamically linked executables, a primary challenge is defining the scope of analysis. A naïve strategy that considers the full code of all linked shared libraries leads to substantial overestimation of the system call footprint, since most library functions are never invoked by the application. The analysis must therefore identify which library functions are reachable from the binary's ELF representation, typically by constructing a call graph rooted at the binary's entry point.

A large portion of an executable's call graph can be reconstructed from disassembled code by following direct call instructions, which in straightforward cases use immediate addresses within the code segment as their targets. However, as discussed earlier, dynamic linking mechanisms such as the PLT and GOT complicate this reconstruction, since some call targets are only resolved at load or run time. Further complexity arises from indirect calls and jumps, whose targets are determined by memory values and often cannot be resolved accurately statically. Such constructs, frequently introduced via function pointers, may leave portions of the call graph unresolved by the system call identification tools. Consequently, failure to account for these cases can lead to incomplete system call detection, undermining the analysis.

During call graph processing, redundant exploration becomes a key concern. Without safeguards such as visited-function tracking, the analysis may traverse the same function several times, leading to exponential growth in traversal cost or even non-termination due to recursion or cycles in the graph. Robust exploration therefore requires mechanisms to detect and avoid revisiting already explored paths.

Challenge 4: Identifying system call numbers. Once the binary has been disassembled, locating potential system call sites should be relatively straightforward: on `x86-64`, this involves identifying each occurrence of the `syscall` instruction used to invoke a system call. The specific system call requested, however, is not encoded in the instruction itself but rather in the contents of the `%rax` register, which holds the system call identifier (as an integer) according to the Linux `x86-64` calling convention [124]. This mechanism is architecture-dependent: other ISAs (e.g., ARM, RISC-V) rely on different dedicated instructions and registers for issuing system calls [172, 223].

In the simplest case, the system call number is assigned directly by an immediate move into `%rax` just before the `syscall` instruction. In practice, however,

more complex patterns may arise, for instance due to compiler optimizations. The value may flow through intermediate registers before being copied into `%rax`, be reconstructed through arithmetic or bitwise operations, or be loaded indirectly from memory (e.g., from the stack or a lookup table). Furthermore, partial-register updates (e.g., writing to `al` or `eax`) implicitly affect the full `rax` register, complicating naive inference. As a result, accurately identifying the invoked system call requires more than local inspection: it demands backward tracking of register definitions across multiple instructions.

Challenge 5: Dynamic loading. On POSIX-compliant operating systems, the dynamic loading interface [54] (`dlopen`, `dlsym`, etc.) adds complexity by enabling programs to load shared libraries at runtime that were not visible during linking and to bind newly resolved symbols to function pointers. If not properly handled, this can lead to missed execution paths and, consequently, overlooked system calls.

Implementation of StaticBinSys

To cope with these challenges, StaticBinSys, operates directly on binary code without requiring source code and remains effective even on stripped ELF executables. The workflow of our tool is structured as a sequence of steps, each intended to handle specific practical difficulties encountered during static binary analysis.

Step 1: Extracting ELF information and dependencies. StaticBinSys requires an ELF binary as input, regardless of whether it is statically or dynamically linked. Once the main binary is verified to conform to the ELF format, it is added to the analysis scope. For parsing ELF objects, StaticBinSys leverages the LIEF library [205], which provides an interface for inspecting and manipulating ELF metadata, thereby simplifying and streamlining the parsing process.

If the binary is dynamically linked, StaticBinSys resolves its library dependencies by inspecting the `.dynamic` section – an ELF metadata table used by the dynamic loader to describe required shared libraries and relocation information (see Section 2.2.1) – and extracting entries that reference the names of needed shared libraries. Since the actual locations of these libraries are system-dependent and not encoded within the ELF file itself, StaticBinSys invokes the `ldd` utility [114] to query the system's dynamic linker and retrieve the concrete file paths of the linked libraries. Furthermore, if the environment variables `LD_PRELOAD` or `LD_LIBRARY_PATH` are set, `ldd` also considers them, ensuring that all dynamically linked libraries are correctly identified. Through this combination of strategies, StaticBinSys reduces the impact of *Challenge 1* and establishes a foundation for subsequent analysis.

Step 2: Disassembly. With the analysis scope established, StaticBinSys proceeds to the problem of disassembly. Most binaries of interest are compiler-generated

and thus follow conventions that render precise disassembly tractable. Modern toolchains such as GCC and LLVM [112, 217] enforce, in most cases, a strict separation between code and data, and frequently embed stack unwinding metadata (in the `.eh_frame` section) in C/C++ binaries on x86-64 architectures. This metadata encodes call frame information describing function entry points and stack frame layouts, which can facilitate the recovery of function boundaries even when symbols have been stripped. StaticBinSys relies on the Capstone disassembly engine [16] to decode instructions. Although the theoretical undecidability of perfect disassembly remains, Capstone applied to compiler-generated binaries provides sufficiently accurate decoding to distinguish code from data. To reconstruct function boundaries, StaticBinSys parses the symbol table when available or falls back to the `.eh_frame` section if symbols have been stripped. This approach enables the recovery of function boundaries in most compiler-generated binaries, thereby alleviating the main practical difficulties of disassembly described in *Challenge 2*.

Step 3: Resolution and scope of library function calls. StaticBinSys incorporates heuristics to handle the resolution of library calls and to restrict the scope of analysis. These measures provide a practical way of handling some difficulties summarized in *Challenge 3*, while acknowledging that accurate resolution is not always possible. The overall procedure is illustrated in Figure 6.1 and can be divided into the following steps:

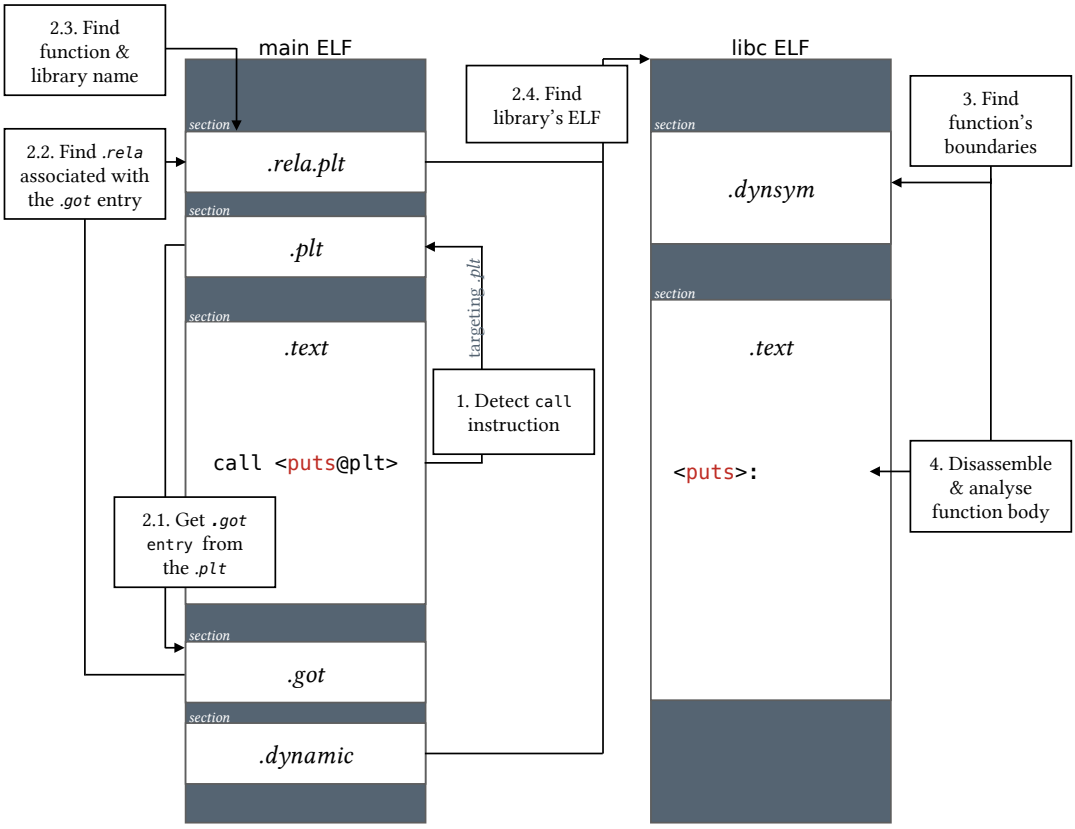


Figure 6.1: *ELF sections consulted during resolution of library function calls.*

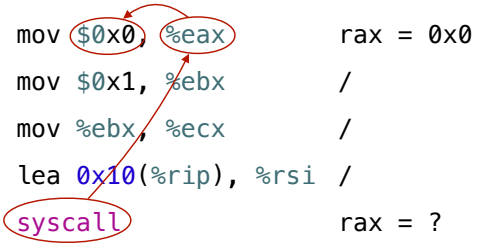
1. *Detecting function calls:* StaticBinSys begins by scanning all executable sections of the main binary to identify function calls and treat them in accordance with their target type:
   - *Direct calls* (immediate targets): These use an immediate address operand that points either to an internal function within the binary or to an entry in the `.plt` section. Calls targeting internal symbols are immediately resolved using the symbol table. Calls targeting `.plt` entries are also resolved at this point but with further processing (see Step 2).
   - *Indirect calls* (register or memory targets): Binary instructions such as `call *%rax` are not resolved directly at the call site. StaticBinSys takes care of them indirectly by treating each instruction performing a register assignment as a potential function pointer assignment. If the assigned value is an address corresponding to the start of a function, StaticBinSys assumes that this function may be invoked through an indirect call. Performing this analysis at the assignment site instead of the call site allows detection of function pointers passed as arguments without needing inter-procedural backtracking support.
2. *Resolving dynamic calls via PLT/GOT:* For calls identified as targeting the `.plt` section, StaticBinSys resolves the actual library function through the following steps:
   2.1 Identify the `.got` entry referenced by the `.plt` instruction.
   2.2 Inspect the relocation record in `.rela.plt` associated with that `.got` entry. If the relocation type is `R_X86_64_JUMP_SLOT`, classify the call as a dynamically linked function invocation.
   2.3 Extract the symbolic name of the target function and the index of its associated library from the relocation entry. Cross-reference these with the list of shared libraries in the binary's `.dynamic` section.
   2.4 Locate the corresponding ELF file of the identified library to complete symbol resolution.
3. **Delimiting function scope:** Once a library function is identified, StaticBinSys determines its precise code boundaries. For library functions, the `.dynsym` section of the corresponding library ELF file encodes the function start address and size. This metadata enables StaticBinSys to recover accurate boundaries, excluding unrelated instructions.
4. **Disassembly and analysis of function body:** For each identified function, *StaticBinSys* disassembles the function body and inspects its instructions to detect: (i) `syscall` instructions, (ii) `call` instructions to `dlopen` and `dlsym` (more details in Step 5), and (iii) `call` instructions to other functions. Functions containing `syscall` instructions are processed to the subsequent system call analysis step. When a `call` instruction is encountered, its target is

resolved as described previously. To handle functions that invoke others – either within the same library or across libraries – *StaticBinSys* traverses the call graph of each discovered function, analyzing all reachable call targets. A visited-function set ensures that every function is analyzed at most once, preventing redundant work and guaranteeing termination even in the presence of recursion or cycles.

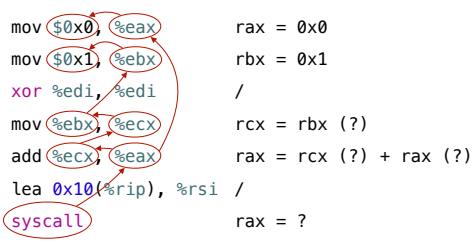
Step 4: System call number inference. StaticBinSys applies a lightweight symbolic backward exploration to reconstruct the contents of the designated system call register at each `syscall` site discovered in the previous step. Starting from the instruction, it symbolically traverses the control flow in reverse, tracking assignments to the target register and propagating symbolic values. In the trivial case where `%rax` is set directly by an immediate move, the system call number is resolved immediately. For more complex idioms, StaticBinSys models register transfers, partial-register writes, and simple arithmetic transformations (e.g., increments, zeroing idioms such as `xor rax, rax`, or masking). When values are derived from memory loads, they are tracked symbolically, with stack-based accesses and their associated values recorded.

Backward symbolic exploration terminates under five conditions: (i) a concrete constant value is inferred; (ii) a function boundary (e.g., prologue) is reached; (iii) a user-defined instruction traversal limit (by default, 30) is exceeded without resolution, meaning that the system call number cannot be determined; (iv) the analyzer fails to handle an instruction relevant to the backtracking process; or (v) Capstone fails to disassemble an instruction, leading to an error.

Figure 6.2 illustrates our approach with two different examples. In the first case (6.2a), the instruction manipulating `%rax` is easily identified, as it involves an immediate move. In the second example (6.2b), StaticBinSys requires additional processing to recover the system-call number. Here, the value is propagated through several registers (`%ebx`, `%ecx`) before reaching the `syscall` instruction. To handle this, StaticBinSys performs data-flow tracking to resolve the register transfers and ensure correct system-call identification.



(**a**) *Simple case: The system-call number is assigned directly via an immediate move instruction.*



(**b**) *The system-call number is propagated through multiple registers, requiring additional data-flow tracking.*

Figure 6.2: *Symbolic backward exploration starts from the `syscall` instruction and determines the system-call number by symbolically tracing the value stored in the corresponding CPU register.*

Although the complexity of the x86_64 instruction set complicates the exhaustive handling of all instruction patterns that may affect the `rax` register, and the analysis remains constrained by the inherent limits of static exploration, `StaticBinSys` nonetheless mitigates several of the issues outlined in *Challenge 4*.

Step 5: Dealing with dynamic loading. Dynamic loading through the POSIX interface (`dlopen`, `dlsym`, etc.) introduces additional execution paths. Analysis of these calls is postponed until all code, except the one of dynamically loaded libraries, is processed and all libraries loaded with dlopen are recorded. Doing this earlier could cause errors, for example, trying to resolve a function from dlsym before its library (loaded with dlopen) has been analyzed, since the analysis does not follow the actual execution order. To address dynamic loading, `StaticBinSys` inspects the arguments of calls to `dlopen` and `dlsym`. The analyzer identifies the constant string argument passed to `dlopen` to determine the names of shared libraries that may be loaded at runtime and the functions that may be dynamically resolved. For each function called with `dlsym`, it is searched for in the set of libraries found with `dlopen`. Each newly discovered function is then analyzed using the same procedure as in Step 3: `StaticBinSys` disassembles the function body, traverses its call graph to identify all reachable callees, and analyzes them when available. Functions containing `syscall` instructions are processed according to Step 4. If additional dynamically-loaded functions or libraries are discovered during this process, the analysis loop is repeated to ensure completeness. This iterative handling of dynamic loading extends the static analysis to runtime-resolved code, thereby reducing the risk of missing system calls, as discussed in *Challenge 5*.

6.2.4 Dynamic Binary Analysis: Challenges and Implementation

Dynamic binary analysis complements static analysis by observing program execution under real workloads, thereby confirming the use of system calls that may be overlooked by purely static inspection. Nevertheless, dynamic analysis suffers from a fundamental limitation: it can only detect system calls that are exercised under the specific workloads and runtime environments in which the program is executed.

This limitation is particularly problematic for programs with rarely executed branches, error-handling code, or input-dependent behavior. Unless workloads are carefully crafted to cover these paths, the resulting system call set will be incomplete. Consequently, coverage dependence represents an inherent weakness of dynamic analysis: it cannot guarantee completeness of system call detection.

Implementation of DynBinSys

To increase the number of system calls detected, we developed DynBinSys, a dynamic tracing tool that systematically explores program execution under multiple

configurations and scripted workloads. While DynBinSys does not eliminate the inherent incompleteness of dynamic analysis, it increases the likelihood of exercising diverse execution paths, thereby reducing under-detection in practice.

Compared with the static workflow, DynBinSys has a simpler architecture, primarily relying on user-provided metadata and `strace` for monitoring.

Like the static analysis workflow, the dynamic analysis process is structured as a sequence of steps that require the user to write a combination of configuration metadata and scripted workloads:

Step 1: Writing program arguments and configuration metadata. DynBinSys relies on a JSON-based metadata file provided by the user/developer, which specifies the command-line arguments and configuration files required to run the program. For instance, the configuration for Nginx [163], shown in Figure 6.3, can include:

```
{
    "arguments": ["", "-h", "-c", "/home/test/nginx/config1", "-c",
    ↪ "/home/test/nginx/config2", "-q", "-v", "-V"]
}
```

Figure 6.3: *Example of an Nginx configuration file where each argument instructs the program to load specific configurations at runtime.*

Each argument set instructs the program to load specific configurations at runtime, enabling DynBinSys to observe system calls triggered under these conditions.

Step 2: Define script interactions for workload coverage. For programs that require interaction, DynBinSys supports scripted execution through user-provided test files. These scripts can be defined in two modes, depending on how the program accepts input: (i) running external commands to interact with the program (`exec`) or (ii) directly interacting with the application through its standard input (`stdin`).

```
{
    "typeTest": "exec",
    "timeMsCommand": 1000,
    "listCommands": [
        "curl -v http://localhost:80",
        "curl -v http://localhost:80/index.html",
        /*...*/,
        "wrk -t12 -c400 -d30s http://localhost:80/index.html"
    ]
}
```

Figure 6.4: *Example of an Nginx test file where commands are executed as external ones.*

The example in Figure 6.4 shows a test file for Nginx using the `exec` mode. In this mode, the program runs in the background, and external commands – such as HTTP requests or benchmarking tools – are executed against it.

In contrast, the example in Figure 6.5 illustrates the `stdin` mode, used for programs such as SQLite [6], where commands are provided directly through the program's interactive interface.

```
{
    "typeTest": "stdin",
    "timeMsCommand": 2000,
    "listCommands": [
        "create table tbl1(one varchar(10), two smallint);",
        "insert into tbl1 values('hello!',10);",
        /*...*/,
        "DROP TABLE tbl2;",
        ".quit"
    ]
}
```

Figure 6.5: *Example of an SQLite test file where commands are provided directly to the program through its interactive interface.*

Each test file also defines a `timeMsCommand` field, which specifies the maximum time (in milliseconds) allotted for the execution of each command. This timeout prevents long-running or blocking operations from stalling the analysis and ensures that all scripted interactions complete within a bounded time frame.

Step 3: Tracing system calls. After specifying program arguments and workloads in user-provided files, DynBinSys performs a restart of the program for every defined configuration and scripted interaction. Across these several iterations, it traces all invoked system calls using `strace`. By combining multiple configurations and input scenarios, DynBinSys increases the coverage of execution paths and reduces the likelihood of missing system calls that would remain unobserved in a single run.

6.2.5 Residual Limitations

Although we chose a hybrid methodology to improve system call detection, certain inherent limitations remain. By their very nature, static and dynamic analysis cannot guarantee complete and correct coverage of all possible execution paths or system interactions.

These residual challenges define the pragmatic boundaries of our approach. While the goal is to exhaustively identify the system calls that an application may invoke, practical limitations of static and dynamic analysis necessitate conservative approximations. As a result, the approach favors over-approximation rather than

risking omissions, providing a comprehensive system call set to enable effective and robust unikernel porting. The following subsections discuss the remaining limitations associated with static and dynamic analysis, respectively.

Static Analysis

Similarly to all other works on static binary system call identification [51, 40, 15], StaticBinSys cannot totally guarantee the absence of false negatives. Several main limitations remain:

1. **Environment variables.** System behavior may be influenced by environment variables such as `LD_PRELOAD` or `LD_LIBRARY_PATH`, which can modify the set of dynamically loaded libraries and the resulting system calls. Although our framework inspects these variables during analysis, it cannot anticipate modifications introduced at runtime or under different execution environments. Moreover, calls to `dlopen` can also be affected by such variables, since they influence the search paths and priority of libraries loaded dynamically, potentially introducing additional system calls that remain invisible to purely static inspection.
2. **Limited backtracking.** When inferring the system call number, backtracking terminates once a value for the `rax` register is found along a single backward control-flow path (i.e., without exploring alternative branches), potentially missing alternative values. Similarly, if the system call identifier is set in a function different from the one issuing the `syscall` instruction, such values will be undetected. In some cases, backtracking may reach its traversal limit without resolving the value, resulting in unresolved system calls. Resolving such cases requires inter-procedural, path-sensitive and complex symbolic execution, which lies outside the scope of our current prototype.
3. **Disassembly.** Disassembly of arbitrary programs is an undecidable problem [222, 25, 221], which imposes an inherent limitation on static analysis. For most compiler-generated binaries that exhibit a clear separation between code and data and include `.eh_frame` or symbol information, the disassembly can be performed accurately. In contrast, for binaries lacking such separations and metadata, potential omissions or misinterpretations in the recovered instructions may occur.
4. **Obfuscation and packing.** Packed or obfuscated applications hinder disassembly and obscure symbol information, significantly reducing the effectiveness of static analysis.
5. **Overestimation.** Static analysis tends to overestimate the set of possible system calls for two main reasons: (1) it conservatively assumes that all instructions within a function body may be executed, including those residing in branches that are never taken by the analyzed program (e.g., branches in library functions that are taken for argument values that are never used by the

analyzed program), thereby inflating the apparent system call footprint. This overestimation is an inherent consequence of the conservative nature of static analysis; (2) the heuristic used for StaticBinSys for detecting indirect calls (at the function pointer assignments site), which treats every source of a store operation as a potential start address of a function, may also introduce spurious targets.

Dynamic Binary Analysis

Although DynBinSys reduces the impact of coverage dependence, several limitations remain inherent to dynamic binary analysis:

1. **Workload quality.** The completeness of system call detection still depends on the diversity and the representativeness of user-provided workloads and scripts.
2. **Environment sensitivity.** Program behavior may vary depending on system configuration, environment variables, and available resources. DynBinSys cannot exhaustively explore all environmental variations.
3. **Runtime modifications and anti-analysis techniques.** Programs that dynamically alter behavior (e.g., via self-modifying code or late-loaded libraries) or detect tracing tools (e.g., `strace`) can evade observation.

6.3 Application Compatibility

Having developed tooling for identifying system calls, we use it to evaluate application compatibility with Unikraft by comparing Unikraft's implemented system calls against those required by widely deployed server applications.

To select representative applications, we relied on the Debian Popularity Contest [50] and chose the 30 most widely used server applications (Apache, MongoDB, PostgreSQL, Avahi, etc.). For each application, we combined static binary analysis with dynamic analysis to extract the set of system calls exercised under representative workloads. The aggregated results were then compared to Unikraft's system call coverage and visualized as a heatmap in Figure 6.6.

The entire analysis and heatmap generation process is automated. Each application is analysed with our static and dynamic tools, and the collected results are processed by a script to produce the heatmap. Each square represents one Linux system call, numbered from 0 (`read`) to 467 (`open_tree_attr`). Light-coloured squares denote calls required by none or few applications (up to 20%), whereas black squares (e.g., square 1: `write`) denote calls required by all or nearly all applications. A number within a square indicates that Unikraft implements the call; an empty square indicates a lack of support.

When this analysis was first performed in 2021, Unikraft supported 146 system calls. To assess the evolution of Unikraft's compatibility over time, we re-ran the

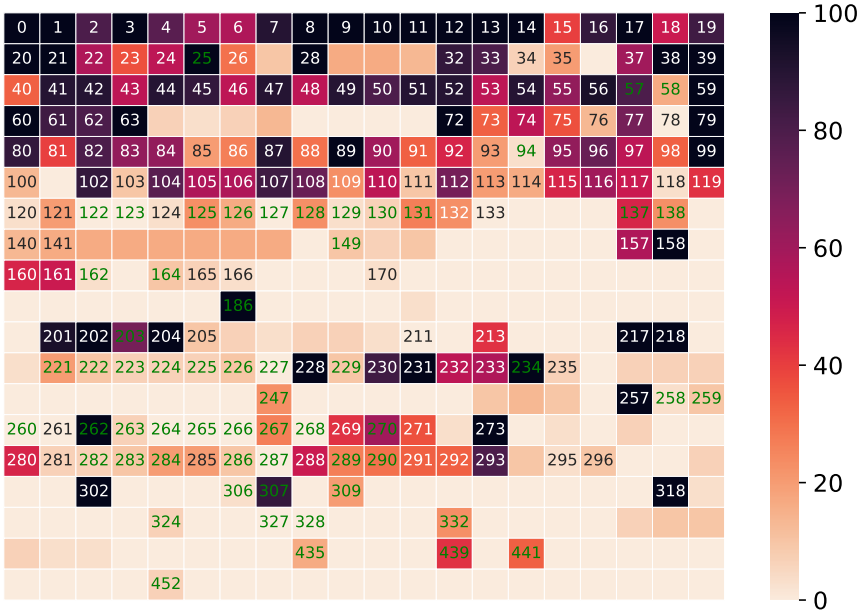


Figure 6.6: *System calls required by 30 server applications compared against system calls supported by Unikraft. Each square corresponds to a Linux system call number (0–467). Colour intensity reflects the proportion of applications requiring the call. White or black numbers denote system calls supported by Unikraft in 2021, while green numbers indicate additional calls supported in 2026. Empty squares denote unsupported calls.*

same analysis in 2026 using the latest version of Unikraft (v0.20.0 Kiviuq). Unikraft now supports 60 additional system calls compared to the 2021 baseline, highlighted in green in Figure 6.6.

Among these additions are several system calls previously identified as compatibility gaps, including `fork` (57), `tgkill` (234), and `newfstatat` (262), which are required by many of the evaluated applications. Other calls, such as `fchmodat` (268) and `preadv2` (327), are not exercised by the evaluated workloads but were added because they could be implemented with relatively little additional effort by reusing existing functionality.

The resulting heatmap reveals that a large proportion of Linux system calls are unused by the popular server applications examined. The additional system calls implemented since the original analysis further confirm that the gap between Unikraft's system call coverage and real-world application requirements is steadily narrowing, and that the tooling developed in this thesis provides a repeatable and quantitative mechanism for tracking this evolution. Among the system calls that remain unimplemented:

- several could be stubbed or faked in a single-application unikernel environment. For instance, the POSIX scheduling family – `sched_setparam` (142), `sched_getparam` (143), etc. – whose semantics are of limited relevance in a single-application unikernel execution model and could plausibly be implemented as no-op stubs returning a constant value;
- others are relatively straightforward to implement since Unikraft already pro-

vides the underlying functionality (e.g., `sem*` family (29–31 and 64–67) for System V IPC, or the `inotify*` family (253–255), given Unikraft's existing VFS support);
- some may not be genuinely required at runtime: static analysis conservatively reports all reachable call sites in a binary, including dead code paths and optional feature probes, which can overestimate the true syscall footprint of an application [117, 51].

Our contribution does not lie in directly extending Unikraft's system call coverage, but in providing tooling that enables the systematic detection and comparison of application system call requirements. This capability allows us to quantify the syscall footprint of different workloads and to assess how well Unikraft's current support aligns with real-world needs.

6.4 Application Support and Porting

With the development of our system call analysis tools, we now turn to the challenge of porting real-world applications to Unikraft. An operating system is only as valuable as the applications it can support, and limited application portability has long been a fundamental obstacle for unikernels, which traditionally require significant manual adaptation to run existing software. This limitation has constrained their adoption despite clear advantages in performance, security, and isolation.

In this section, we examine and compare different application porting strategies for Unikraft, analyzing their trade-offs in terms of developer effort, compatibility, and performance. We investigate a porting approach based on the reuse of externally compiled object files. To support this approach, we extend Unikraft with two key additions: a port of a C standard library (libc) to improve POSIX compliance, and UkTools, a framework that assists developers in integrating applications into unikernels by leveraging the results of our system call analysis.

6.4.1 Porting Strategies

Porting an existing application to Unikraft can follow several paths, each trading off development effort, performance, and compatibility. At one extreme, *source-based porting* involves adapting the application source code to Unikraft's build system and APIs, offering maximum control and optimization potential but requiring extensive developer intervention. At the other extreme, *binary compatibility* aims to execute existing binaries with little or no modification, minimizing porting effort but often at the cost of performance. Between these two extremes lies an intermediate strategy that reuses externally compiled object files. While this approach is implicitly supported by existing build mechanisms, it has received little systematic attention in the context of unikernels and has not been thoroughly characterized as a porting strategy [111, 148, 101, 97].

Our work is motivated by the need to better understand this middle ground and to provide developers with insights into which system calls and libraries are required for a successful port. The remainder of this section examines the design trade-offs of these strategies.

Binary Compatibility

Recent efforts have explored binary compatibility [148, 101], allowing unmodified binaries to execute by translating system calls into the unikernel's internal functionality at runtime. Within Unikraft, achieving such compatibility requires (1) a dedicated loader capable of correctly loading and executing unmodified ELF binaries (statically and dynamically linked), (2) mechanisms to intercept and handle the `syscall` instruction (for `x86_64`), and (3) corresponding implementations for each invoked system call. Although this approach removes the need for manual porting and enables execution without source access, it introduces overhead through additional layers of indirection and risks runtime instability due to subtle Application Binary Interface (ABI) mismatches. At the time of this work (2021), Unikraft provided only minimal and experimental support for binary compatibility [118] (only statically linked binaries can be executed).

Platform	Routine call	#Cycles	nsecs
Linux/KVM	System call	222.0	61.67
	System call (no mitigation)	154.0	42.78
Unikraft/KVM	System call (runtime translation)	84.0	23.33
Both	Function call	4.0	1.11

Table 6.1: *Cost of system calls with and without security mitigations. System calls using runtime translation in Unikraft are 2–3× faster than in Linux but still incur a 21× overhead compared to direct function calls, highlighting the performance impact of binary compatibility.*

To quantify system call overheads, Table 6.1 presents microbenchmark results obtained on an Intel® i7-9700K (3.6 GHz) running Linux 5.11. The benchmarks compare the cost of `no-op` system calls and function calls across Linux and Unikraft. In Linux, system calls incur the cost of a protection-domain transition between user and kernel space, along with associated entry and exit overheads and, when enabled, security mitigations such as Kernel Page Table Isolation (KPTI) [102]. Disabling these mitigations reduces the cost from 222 to 154 cycles. In contrast, Unikraft executes all code within a single address space and therefore avoids protection-domain switches entirely. System calls executed via runtime translation still involve a trap through the `syscall` instruction and subsequent exception handling, but they do not require a transition between user and kernel modes. As a result, binary-compatible system calls in Unikraft are 2–3× faster than their Linux counterparts. Despite this advantage, runtime system call translation remains significantly more expensive than native function calls. As shown in Table 6.1, translated system calls incur a 21× overhead compared to direct function calls. This overhead stems

from the mandatory trap and exception-handling path and makes binary compatibility, as implemented in OSv [101], Rump [97], and HermiTux [148], less suitable for performance-critical workloads.

Sources-based Porting

When source code is available and performance is a primary concern, source-based porting offers the most direct route to optimization. In this mode, applications and their dependent libraries are compiled with Unikraft's modular components, often requiring developers to adapt or rewrite the application's build system. In practice, this is complicated not only by the need to identify and isolate the relevant source files – since many projects include auxiliary sources, test code, or platform-specific fragments [212] – but also by the difficulty of correctly integrating third-party libraries. Each external dependency usually requires a comparable effort: adapting build scripts, inspecting its architecture, and ensuring compatibility with Unikraft's APIs. Moreover, developers must often determine the appropriate compilation flags and occasionally patch source files to resolve incompatibilities. Although this approach can yield highly optimized unikernels, it remains labor-intensive and demands detailed knowledge of the target application, its third-party libraries, and Unikraft's API and build process. Consequently, it can be time-consuming and error-prone, particularly for large or frequently evolving software projects.

Object-Level Reuse (Relying on Native Build System)

To balance the runtime overhead of binary compatibility against the engineering effort of full source porting, we consider an intermediate strategy. This approach leverages the application's native build system to generate relocatable object files, which are then linked directly into Unikraft during the final build stage. By doing so, it reduces both runtime overhead and manual porting effort.

Unikraft provides native support for object-level reuse, but two challenges remain. First, newlib [162], the main libc used in Unikraft, offers limited POSIX compliance. Second, manually managing library dependencies and selecting the appropriate object files remains cumbersome.

6.4.2 Improving Unikraft's Application Porting Workflow

To address these challenges, we extend Unikraft's ecosystem with two complementary contributions: the port of musl as a native Unikraft library – enhancing POSIX compliance – and the development of UkTools, a framework that helps developers automate library resolution and object integration. Together, these components form a more streamlined and reliable workflow for porting applications, reducing manual effort while preserving unikernel efficiency. The following sections describe our contributions in detail.

Musl Integration with Unikraft

Unikraft already supports newlib as a libc library. However, newlib is primarily designed for bare-metal and embedded environments and therefore provides limited POSIX compliance and lacks many features available in more complete libraries such as glibc [68] or musl [61]. To provide broader POSIX support and facilitate the porting of general-purpose Linux applications, we ported musl as a Unikraft library. Musl offers strong compatibility with glibc while remaining lightweight, resource-efficient, and straightforward to integrate into the Unikraft ecosystem.

Porting musl to Unikraft required more than simply linking it as an external library. Musl assumes the presence of a Linux-like execution environment, including a set of POSIX-compliant system calls. To integrate it, we adapted its codebase to Unikraft's build system. To avoid symbol redeclaration during linking, implementations already provided by other Unikraft libraries – such as networking, process management, and signals – were excluded from the build process through refinements to the Makefile rules. Additional glue code was introduced to adapt musl's interfaces to Unikraft's internal APIs, particularly for memory allocation. These adaptations allow musl to serve as the libc for Unikraft with minimal changes to its upstream source code, while preserving strong POSIX compliance and compatibility with existing Linux applications. The initial port has since been incrementally refined, with the complete set of commits available in the public GitHub repository [18].

Beyond build integration, supporting musl requires addressing its reliance on Linux system calls. For this purpose, Unikraft provides a lightweight *syscall shim* layer [19], which maps Linux-style system call numbers to Unikraft's internal handler functions. Each library that implements a system call registers its handler with the shim via a macro, allowing system call requests to be directed to the appropriate implementation. From musl's perspective, the shim behaves like a regular Linux kernel interface, enabling it to issue system calls without modification. These calls are transparently redirected as direct function calls to the corresponding Unikraft libraries. By linking system call implementations directly at build time, Unikraft effectively transforms system calls into ordinary function calls, offering significant performance benefits over traditional Linux system calls.

Porting musl is thus a major step toward building general-purpose unikernels, as it provides the essential system interfaces for POSIX compliance. However, musl support alone does not ensure seamless application portability: missing or mismatched library dependencies can still cause compilation or linking errors unless the appropriate libraries are selected and included in the final unikernel image.

UkTools: Overview

Porting existing applications to a new execution environment often requires careful dependency management. Applications to port often depend on a diverse set of

libraries. Omitting a required library or object file from the external build system may result in linking errors (undefined symbols) or runtime crashes (missing system call implementations). Conversely, including all available libraries and object files can cause linking conflicts (e.g., duplicate symbols) or unnecessarily increase binary size and attack surface. Selecting the correct combination of libraries and object files is therefore crucial for an efficient unikernel build.

Addressing this challenge requires a systematic mechanism to map an application's expected dependencies to the libraries available in Unikraft while correctly integrating the required externally built object files. For this purpose, we developed UkTools [71], an open-source command-line tool designed to assist developers in porting applications to Unikraft while minimizing manual intervention. UkTools analyzes an application's directory and extracts the necessary information through both static and dynamic binary analysis, leveraging StaticBinSys and DynBinSys.

The tool supports modular workflows, allowing individual steps – such as static analysis – to be executed independently. Although UkTools guides and streamlines much of the porting process, it is not fully automatic and still requires minimal developer interaction. In particular, the target application must first be built with its native build system to produce the compiled ELF binary and corresponding object files. The tool follows a structured workflow composed of six steps, illustrated in Figure 6.7:

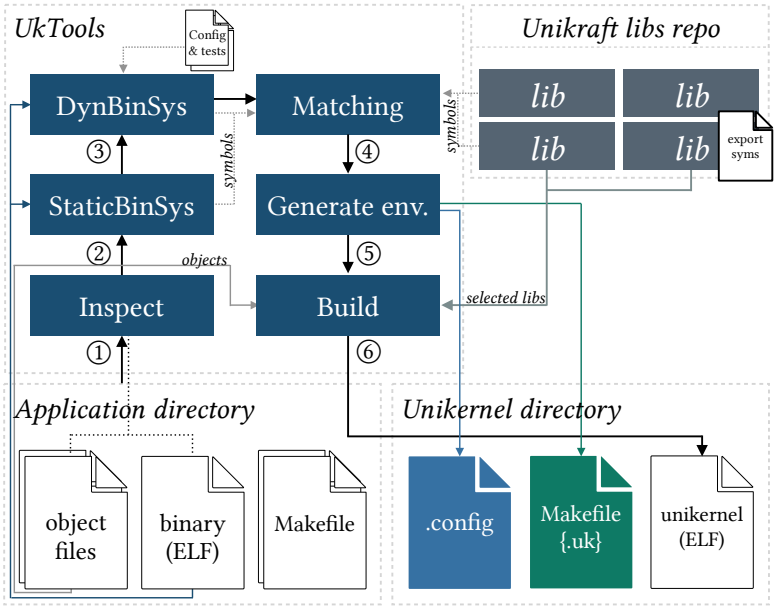


Figure 6.7: *UkTools follows a structured workflow: ① Inspect the application's directory, which must include the final ELF binary along with its corresponding object files. ② Perform static analysis with StaticBinSys on the binary to extract relevant information. ③ Conduct dynamic analysis by running the application with various configurations and workloads using DynBinSys. ④ Match the extracted symbols with Unikraft libraries. ⑤ Generate the build environment. ⑥ Compiles the final unikernel with the required libraries.*

① UkTools first inspects the application's directory, which must contain the final

ELF binary and its corresponding object files. This directory often also contains the source code and build system configuration (e.g., Makefile), which, while not required for analysis, are typically used to produce the binaries processed by UkTools.

② A static analysis is then performed on the ELF binary using an extended version of StaticBinSys. Unlike the baseline StaticBinSys, our extension also extracts library functions in addition to system calls by relying on the ELF symbol table.

③ To complement static analysis, UkTools employs DynBinSys for dynamic analysis in order to capture dependencies that may be missed statically. In addition to using `strace` to trace system calls, we extended DynBinSys with `ltrace` to record dynamic library calls invoked by the running binary. The results of the dynamic analysis are then merged with the static analysis output for subsequent processing.

④ The symbols identified by the analyzers are matched against those exported by Unikraft libraries, as defined in their `exportsyms.uk` files. UkTools applies a simple matching algorithm that compares the application's required symbols against each library's exported symbols. For every library, the tool computes the intersection between the application's symbol set and the library's exported symbols. Libraries with at least one match are retained, and the size of the intersection determines their relevance: the larger the intersection, the stronger the candidate library. If multiple libraries provide the same symbol(s), UkTools prompts the developer to resolve the ambiguity by selecting the appropriate library.

⑤ UkTools then generates the build environment for the application. At this stage, developers are invited to review and, if necessary, refine the selection of object files. The generation process consists of creating a configuration file specifying the required internal libraries (`.config`), a dedicated `Makefile` for external dependencies such as musl or lwIP, and a `Makefile.uk` that integrates externally compiled object files into Unikraft's final linking process.

⑥ Once the environment is generated, UkTools attempts to build the unikernel using the selected libraries and precompiled object files (via the `make` command). Compilation errors are reported on the console. In the case of linking errors (e.g., duplicate symbols), the tool prompts the developer to review and refine the object file selection. Other errors are displayed for manual inspection.

6.4.3 Porting Applications with UkTools through Object-Level Reuse

We then evaluate the effectiveness of our object-level reuse strategy for porting applications to Unikraft. Using UkTools, we streamline the integration of externally compiled object files with Unikraft libraries and assess the resulting compatibility

when building against musl. The results, summarized in Table 6.2, show that this approach successfully builds roughly half of the tested applications. For those that do not, the failures are due to the use of glibc-specific symbols. To address this, we implement a glibc compatibility layer based on a series of musl patches [228] and 20 additional functions implemented by hand, mostly 64-bit versions of file operations such as `pread` and `pwrite`. With this compatibility layer in place, as indicated in Table 6.2 (last column), the combination of UkTools and the glibc compatibility layer enables the successful compilation and linking of all tested libraries and applications.

	Size (MB)	musl (standard)	musl (glibc compatibility layer)
axtls	0.364	✗	✓
bzip2	0.324	✗	✓
c-ares	0.328	✗	✓
duktape	0.756	✓	✓
farmhash	0.256	✓	✓
fft2d	0.364	✓	✓
helloworld	0.248	✓	✓
httpreply	0.252	✓	✓
libucontext	0.248	✓	✓
libunwind	0.248	✓	✓
lighttpd	0.676	✗	✓
memcached	0.536	✗	✓
micropython	0.648	✓	✓
nginx	0.704	✗	✓
open62541	0.252	✓	✓
pcre	0.356	✓	✓
python3	3.1	✗	✓
redis-client	0.660	✗	✓
redis-server	1.3	✗	✓
ruby	5.6	✗	✓
sqlite (amalgamation)	1.4	✗	✓
zlib	0.368	✗	✓
zydis	0.688	✓	✓

Table 6.2: *Results obtained using UkTools and our object-level reuse approach, which links externally built object files against Unikraft with musl. The table reports, for each application, whether the build succeeded with the glibc compatibility layer and without it, as well as the resulting unikernel size.*

6.5 Discussion

This section discusses complementary techniques and design considerations that extend and contextualize our work.

6.5.1 Symbolic Execution

Symbolic execution [33, 100, 92, 103] is a powerful program analysis technique that explores multiple program paths by treating inputs as symbolic values rather than concrete data. It enables reasoning about a program's behavior under diverse input conditions and can, in principle, yield precise information about control-flow decisions, data dependencies, and potential system interactions such as system calls.

In practice, applying symbolic execution to large, real-world programs remains extremely challenging [24, 43, 30, 181]. Complex applications such as Nginx or SQLite include millions of lines of code, depend on many dynamically linked libraries, and exhibit highly variable runtime behaviors [163, 6]. As a result, the number of possible execution paths grows exponentially, a phenomenon known as path explosion, making exhaustive symbolic exploration computationally infeasible.

Nevertheless, symbolic execution can still play a valuable role when coupled with static binary analysis. Rather than attempting to symbolically execute an entire binary, it is possible to integrate targeted symbolic reasoning into specific stages of the analysis pipeline to improve accuracy. For instance, during system call inference, symbolic execution could be selectively applied to identify conditional branches that guard system call invocations or to track inter-procedural propagation of system call arguments. We leave the integration of such targeted symbolic reasoning as a promising direction for future work.

6.5.2 Revisiting Porting Strategies in Light of Unikraft's Evolution

The recent evolution of the Unikraft ecosystem necessitates a re-evaluation of the porting strategies explored in this work. In particular, new mechanisms for dynamic linking and binary execution have reshaped the balance between source-based porting, object-level reuse, and full binary compatibility.

Relying on externally built object files represents an intermediate approach between source-based porting and full binary compatibility, balancing porting effort and performance. At the time our tools were developed, the Unikraft platform provided only limited binary compatibility, as dynamically linked libraries were not yet supported. Recent advancements [21, 20] now enable the execution of dynamically linked binaries, significantly extending Unikraft's compatibility. This improvement is especially important since most applications rely on dynamic linking, with build systems and compilers typically producing dynamically linked rather than static binaries [212]. Today, the Unikraft framework can execute dynamically linked binaries directly (the appropriate shared libraries must be included), thus simplifying the porting of complex applications such as PostgreSQL or MongoDB and removing the need for source code access. In contrast, our current approach still requires developer intervention to validate the automatically included object files and may

involve several refinement cycles to obtain a correct build, particularly for large or complex applications.

These developments naturally motivate an evolution of UkTools. Our framework can be readily extended to support binary compatibility in addition to its current object-level reuse mode. Depending on the developer's intent, UkTools could automatically switch between these modes. To enable binary compatibility, it would integrate Unikraft's ELF loader [21], perform library matching using information extracted by StaticBinSys and DynBinSys, and automatically construct the corresponding unikernel. Nevertheless, this trade-off remains a key design consideration: developers must balance the convenience of binary compatibility against the potential performance benefits offered by object-level reuse or full source-level integration.

6.6 Related Work

In this section, we review research efforts that employ binary analysis to detect system calls, discussing their relevance to the unikernel ecosystem and their relation to our own approach. We also examine the porting process and compare it with other unikernel frameworks.

Detecting system calls. Many existing approaches rely exclusively on dynamic analysis [91, 120, 143, 219], while others are based solely on static analysis [148, 49, 152, 38]. As previously discussed, each method suffers from inherent limitations. In this work, we adopt a hybrid analysis strategy that combines static and dynamic techniques to mitigate the shortcomings of each. Moreover, several prior studies assume full access to the source code of target applications and their dependent libraries [76, 49], an assumption we avoid in designing our tools.

Related efforts such as Sysfilter [51] and Chestnut [40] address different objectives. These systems leverage system call identification to generate seccomp filters [173] aimed at minimizing the kernel attack surface of applications, whereas our work focuses on ensuring application compatibility.

Linux & POSIX APIs studies. Past work has examined the usage of the Linux and POSIX APIs by applications [212, 15]. Tsai et al. [212], for example, employ binary static analysis to characterize the system calls and pseudo-files required by a large set of binaries from the Ubuntu 15.04 archive. While their study provides a broad empirical view of system call usage, it is limited to static analysis and does not integrate dynamic behavior. Consequently, their results may miss runtime-dependent system calls. In addition, their backtracking approach for syscall identification used in their analysis is limited, as it only handles immediate values and does not account for arithmetic operations or register-to-register transfers. Another

related study [15] employs both static and dynamic analysis to measure applications' POSIX API usage; however, their approach appears to analyze entire shared library code without reconstructing an accurate call graph to identify system calls.

Porting with other unikernel frameworks. Recent years have witnessed the emergence of numerous library OS and unikernel projects, many of which target specific applications or programming languages. Examples include runtime.js [175] (JavaScript), IncludeOS [34] (C++), HaLVM [225] (Haskell), Erlang on Xen [45] (Erlang), MiniPython [190] (MicroPython), ClickOS [132] (for network function virtualization), and MiniCache [107] (a Xen-based virtualized content cache). MirageOS [127] is an OCaml-based unikernel emphasizing type safety, but it lacks compatibility with mainstream software. Unlike these solutions, Unikraft is designed to efficiently support a broad range of mainstream applications and runtimes.

Other approaches adopt different strategies for unikernel deployment. For instance, UniK [183] acts as a high-level wrapper for multiple unikernel frameworks. Containers and Kata Containers [160] provide lightweight, Linux-based virtual machines tailored for containerized workloads. Drawbridge [156] is a library OS optimized for running Microsoft applications. The Rump Kernel [97] introduced the "Anykernel" concept, repurposing NetBSD kernel components to run in a single address space while maintaining compatibility with system calls like `fork` and `execve`. However, its reliance on a monolithic kernel limits specialization opportunities. HermiTux [148] achieves binary compatibility through system call rewriting but sacrifices both customizability and performance, drawbacks that Unikraft avoids. Similarly, OSv [101] targets cloud workloads with binary compatibility and multi-language support, but its monolithic design hinders fine-grained optimization.

Recent efforts explore alternative approaches to reducing system call overhead. UKL [161] statically links applications with the Linux kernel, using a shim layer to redirect glibc calls, though this adds significant overhead (45 MB per image). Lupine [109] loads applications directly in kernel mode, eliminating system call costs while leveraging kernel configuration to minimize footprint. However, neither UKL nor Lupine specializes the underlying software stack as effectively as Unikraft. Although not directly related to unikernels, the Yocto [67] project facilitates porting by leveraging the existing Linux ecosystem. While porting applications to Unikraft often requires more porting effort, it leads to significant performance improvements.

6.7 Summary

This chapter addresses the fundamental adoption barrier of unikernels: the complexity of porting existing applications. The Unikraft framework alleviates this challenge by providing a modular, library-based OS that can be tailored to each appli-

cation. However, building a unikernel still requires precise identification of an application's system calls, a task made difficult by the lack of dedicated tooling and methods. Pure static analysis struggles with indirect calls and unresolved control flow, whereas dynamic analysis can miss rarely exercised code paths.

To overcome these limitations, we developed a hybrid methodology and designed two complementary tools for more robust system call detection. Using these tools, we analyzed 30 widely used server applications and compared their system call requirements against those implemented in Unikraft, providing an empirical benchmark to guide its ongoing development.

We then systematically evaluate multiple porting strategies, including binary compatibility, manual porting and leveraging the application's native build system. We demonstrate that the most efficient approach leverages the application's native build system in combination with binary analysis tools to detect required system calls and libraries, while relying on musl as the libc alongside a compatibility layer. Together, these contributions form a cohesive pipeline that reduces manual effort and the need for deep system expertise.

6.8 Potential Improvements

Improving static analysis. Our current approach to handling indirect calls tends to overestimate system calls, while some calls are missed due to the limitations of our backtracking analysis. These inaccuracies highlight opportunities for enhancing precision and coverage. Future work could integrate symbolic execution to explore execution paths more comprehensively, enabling more accurate identification of indirect calls. Alternatively, machine learning techniques could be employed to predict likely call targets based on patterns in binary code, reducing both false positives and false negatives. Combining these strategies has the potential to improve the precision of system call analysis in the static analysis.

Automatically building applications with their native build system. A major improvement would be enabling UkTools to automatically configure and build target applications prior to analysis. This would involve handling various build systems such as CMake [9], SCons [184], and Autotools/Automake [65], as well as managing dependencies by fetching and configuring required libraries. Automating this process would reduce manual effort and make the system more accessible for developers looking to port their applications with minimal intervention.

Extending dynamic analysis with fuzzing. Dynamic analysis in our workflow is primarily intended to complement static analysis by capturing system calls that may be missed statically. Because it serves as a complement rather than a comprehensive approach, there is room to improve its coverage. One possible extension is to incorporate fuzzing techniques [234, 174] to automatically explore a wider range

of execution paths within the target application. By generating diverse and unexpected inputs, fuzzing can trigger code paths that are unlikely to be exercised by standard test cases or representative workloads, potentially revealing additional system calls. Careful selection of fuzzing targets and input interfaces would be required to balance coverage improvements against analysis execution time.

Using test suites. To improve coverage in dynamic analysis, existing application test suites could be used instead of custom tests. These suites typically exercise a broader and more representative range of execution paths, providing a more comprehensive view of system call and library usage. However, integrating them with our analysis tools requires additional effort. In particular, test suites often depend on specific build environments or runtime configurations that must be adapted to execute within our controlled analysis framework. Despite this added complexity, incorporating test suites would enhance the accuracy of dynamic analysis.

Using large language models to generate test inputs. A complementary direction is to use Large Language Model (LLM) [36, 52, 214] to automatically generate targeted test inputs for dynamic analysis. LLMs could produce semantically meaningful inputs by leveraging application-specific context such as expected input formats, command-line interfaces, or available source code and documentation. This approach can be integrated at multiple levels of the analysis workflow: at a fine-grained level, LLMs can generate unit tests targeting specific functions or modules, while at a higher level they can construct scenario-based tests that simulate realistic and complex application usage. LLM-generated inputs could also serve as high-quality seeds for coverage-guided fuzzers [234, 174, 236]. A key challenge is ensuring the correctness and relevance of generated inputs, as LLMs provide no formal coverage guarantees and may produce redundant or invalid test cases—making validation and filtering necessary components of such a pipeline. Addressing these limitations represents a promising direction for improving syscall coverage in dynamic analysis workflows.

This page intentionally left blank.

Part IV

Conclusion & Future Work

This page intentionally left blank.

7

CONCLUSION AND FUTURE RESEARCH DIRECTIONS

Over the past decade, cloud computing has undergone a significant transformation in how applications are deployed and managed. Initially dominated by virtual machines, which offered strong isolation but came with considerable overhead, the industry gradually shifted toward containerization to achieve lightweight and more flexible deployment models. Containers reduce resource consumption and startup times, but their shared kernel model weakens isolation by exposing a large attack surface. More recently, unikernels have emerged as a promising alternative: by compiling applications directly with only the necessary operating system components into a single-purpose image, unikernels dramatically reduce attack surfaces, improve boot times, and minimize file size, making them particularly well-suited for modern, scalable cloud environments.

This thesis addressed two limitations of current unikernel systems: (1) the lack of sharing opportunities that restrict the effectiveness of memory deduplication and thus density in cloud environments, and (2) the absence of tooling and systematic workflows to port existing applications, making practical adoption difficult.

To address the first challenge, we established that specialization and compactness in unikernels constitute a primary barrier to page-level deduplication. Guided by this insight, we introduced Spacer and Spacer (ASLR), alignment-driven mechanisms that reorganize and inflate ELF memory layouts to increase cross-instance similarity and improve the effectiveness of scanning-based deduplication mechanisms such as KSM. These techniques reduce memory usage by up to 3× with negligible performance overhead. However, relying solely on runtime scanners introduces overhead and unpredictability. To overcome these limitations, we developed Spacer-SLT to perform memory deduplication at load time, eliminating the dependency on costly background scanning. Spacer-SLT delivered load time memory deduplication and improved startup times – key factors for cloud and edge deployments. Finally, we addressed the challenge of library versioning in statically

linked unikernels. Unlike Spacer and Spacer-SLT, which treat different library versions as separate entities and limit sharing, Spacer-Δ improves memory efficiency by enabling cross-version page sharing. By combining library alignment with a differential representation of library versions, Spacer-Δ preserves memory layout consistency and enables effective page-level sharing across unikernels built against different versions of the same libraries.

To tackle the second challenge, we focused on the lack of systematic workflows for bringing existing software into unikernel environments. We examined the necessity of identifying system calls in traditional applications and the challenges inherent in developing robust system call detection tools. To this end, we designed a hybrid analysis approach that combines static and dynamic binary analysis to infer an application's operating system dependencies, compensating the limitations of each technique when used alone. Moving beyond identification, we considered a novel intermediate porting paradigm that circumvents the high engineering cost of source-level adaptation and the performance overhead of binary compatibility. By integrating a musl libc port and a glibc compatibility layer, we significantly lower the POSIX barrier. We further developed UkTools, a framework that semi-automates the porting workflow by mapping the required interfaces to the appropriate libraries, reusing externally compiled objects, and generating modular build configurations. The resulting approach preserves the performance and specialization benefits of unikernels while significantly lowering the barrier to adoption for real-world software.

In summary, this thesis has provided a multi-faceted contribution to the practical adoption of unikernels. On the one hand, the Spacer family transforms unikernels from poor-sharing specialized images into efficient and scalable cloud deployments, achieving significant memory savings through layout optimization and instantaneous load time deduplication. On the other hand, our porting-related tools reduce the key technical barrier to their use, enabling developers to leverage the benefits of unikernels for existing applications without requiring deep systems expertise. All tools have been released as open-source, inviting adoption, improvement, and extension by both industry and the systems research community. Beyond cost and density improvements, the resulting reduction in memory footprint translates to lower energy consumption and improved resource utilization, through consolidation, representing a critical step toward more sustainable and carbon-efficient cloud infrastructures.

Future Work and Research Directions

In addition to the improvements that we propose in each previous chapter, we conclude this thesis with a discussion that draws future lines of research from unexplored paths and ideas.

Ideal placement for unikernel functions. A promising direction lies in determining the optimal placement of functions in memory to reduce the number of page faults and cache misses during execution. Minimizing these factors can significantly improve runtime performance, particularly for latency-sensitive applications. Ideally, functions could be grouped together in memory based on their usage frequency. However, this strategy becomes more complex when also aiming to optimize memory deduplication across several instances.

To maximize sharing and minimize memory usage, it is necessary to preserve a consistent ordering of functions across all unikernel instances, which can conflict with purely performance-driven placements. Additionally, Dead Code Elimination (DCE) can be employed to remove functions that are never invoked (across all instances), reducing the memory footprint and improving deduplication opportunities. Future work could investigate hybrid strategies that combine the benefits of (global) DCE with function placement optimizations, potentially guided by profile-driven analysis to balance performance and memory efficiency.

Snapshots and fast restore. Another complementary direction involves leveraging memory snapshots to enable rapid application startup. Rather than initializing each unikernel instance from scratch, a single pre-initialized base snapshot could capture the system state immediately after bootstrapping, serving as a foundation for launching multiple application instances with reduced startup latency and improved memory reuse.

In contrast to existing approaches that require one snapshot per application, this method would reuse a single base snapshot across applications. The base snapshot would contain common components, such as core libraries, mapped at fixed virtual addresses to enable memory sharing. A custom loader could then load application-specific code and libraries by constructing appropriate page table mappings, selecting their virtual addresses at load time, and updating the page tables accordingly.

Machine Learning-Assisted System Call Inference. As mentioned in Chapter 6, certain limitations remain inherent to static binary analysis, particularly in the presence of indirect calls, inter-procedural dependencies, and complex control-flow patterns that hinder precise system call identification. A potential direction for future work is to explore Machine Learning (ML) techniques as a complement to traditional static analysis rather than as a replacement. Machine learning models can capture recurring low-level patterns in compiled code, such as instruction sequences, register usage, or control-flow structures. In this context, learning-based methods could be used to infer likely system calls or system call categories from such patterns, especially in cases where static analysis fails to resolve concrete system call identifiers. For instance, models could be trained to associate functions or basic blocks with coarse-grained system call categories (e.g., file I/O, memory management, networking), even when exact system call numbers cannot be stati-

cally determined.

Extending Spacer to the Linux Ecosystem. Another potential direction for future work is applying Spacer to the Linux ecosystem, which can be explored at two levels.

Within the Linux kernel. Applying Spacer-like techniques here targets user-space binaries and their loading process. For statically linked binaries, this would require page-aligned libraries (e.g., via custom linker scripts) and extensions to the ELF loader to support load time deduplication mechanisms similar to Spacer-SLT. This in turn implies kernel-level changes for shared memory mapping across processes. Spacer-Δ could be applied to versioned libraries as in unikernels. For dynamically linked binaries, the architecture differs significantly; while Spacer-SLT may still improve isolation and performance, adopting the full Spacer toolchain would require substantial changes to the Linux ecosystem.

At the kernel level (as a VM guest). Alternatively, Spacer can be applied to the Linux kernel binary itself. The kernel image (e.g., vmlinux) is a statically linked ELF [210] binary where subsystems and modules are compiled into a monolithic artifact. By analogy with Spacer's library model, these subsystems can be treated as libraries, enabling Spacer to identify and share identical components across kernel images. Spacer-SLT-like mechanisms could then maintain shared component pools across VMs, while Spacer-Δ would reduce memory usage by storing only inter-version differences.

Orchestration and hyperscale environments. Spacer could be extended beyond its current single-machine deployment model toward large-scale cloud environments. At present, Spacer relies on an in-memory cache storing all libraries required by co-located unikernels, without any global coordination or orchestration layer. A natural extension would be to design a dedicated orchestration framework [35] capable of managing multiple zones composed of physical machines and unikernels. Such an orchestrator could maintain a global view of library utilization/popularity across the infrastructure, and proactively prefetch frequently accessed libraries ahead of unikernel instantiation or migration events – significantly reducing cold-start latency and improving memory efficiency at scale [142, 69]. Integrating telemetry-driven prediction models or lightweight machine learning techniques could further enable adaptive cache management policies that react dynamically to workload fluctuations, transforming Spacer from a host-local optimization into an infrastructure-wide memory efficiency layer [178, 180].

Extending Spacer techniques beyond cloud deployments. The memory optimization techniques developed in this thesis – Spacer, Spacer-SLT, and Spacer-Δ – were designed and evaluated in the context of cloud deployments, where unikernels execute as guests inside a hypervisor. However, the underlying principles of layout alignment, load time sharing, and delta-based versioning are not inherently cloud-

specific. Edge and fog computing as well as Internet of Things (IoT) represent a natural extension target [3, 44, 42, 80]. In such environments, network bandwidth, storage capacity, and memory availability are more constrained than in centralized cloud datacenters. On platforms capable of running lightweight hypervisors, Spacer techniques could reduce memory usage, improve startup performance, and decrease update transfer sizes for replicated unikernel deployments. Further investigation is nevertheless required for platforms that cannot support a hypervisor layer. Exploring how Spacer techniques could be adapted to these environments remains an important direction for future work.

Adaptation to containers. Future research could focus on adapting Spacer, Spacer-SLT, and Spacer-Δ beyond unikernels toward containers [135, 48]. While limited forms of memory sharing already exist in container environments – such as indirect savings through shared read-only image layers via overlay filesystems [121], and page-level deduplication across eligible memory regions via KSM when pages are marked as mergeable – modern container deployments can still experience memory bloat, particularly due to the increasing use of statically linked languages such as Go [122] and Rust [165]. One possible direction is to design tooling and a specialized container runtime that enables cross-container page sharing and delta-versioned libraries from a centralized host pool, extending Spacer-SLT's library pool concept to container environments.

LLM-Assisted System calls Implementation. An interesting avenue for improving unikernel compatibility is the use of LLM-assisted development [36, 52, 214] to implement missing system calls. A first approach can leverage our existing tools to analyze large sets of applications and identify the most frequently used system calls. An LLM can then generate implementations for the most frequently required syscalls, covering a substantial portion of real-world application needs from the outset. This frequency-driven strategy can be complemented by a dynamic phase. Less common or application-specific syscalls can be stubbed with no-op or best-effort implementations, allowing execution to proceed while a monitoring layer captures their usage, arguments, and effects. These traces can then guide iterative LLM-based synthesis of the remaining syscalls. Together, this two-phase approach offers a practical path toward progressively expanding syscall coverage while reducing manual effort, though correctness and semantic fidelity remain open challenges.

Unikernels and Artificial Intelligence. An interesting topic for future investigation is exploring the suitability of unikernel-based systems for emerging Artificial Intelligence (AI) workloads, including LLM. These workloads are characterized by large memory footprints, complex runtime dependencies, and diverse execution patterns, which make their deployment challenging in cloud environments. While unikernels are traditionally used for lightweight services, their strong isola-

tion properties and minimal runtime may provide a useful foundation for investigating alternative deployment models for AI.

One potential avenue is to study whether memory alignment and deduplication techniques, such as those developed in this thesis, could be applied to portions of AI workloads that are largely read-only, such as model parameters or common runtime components. However, the feasibility of such sharing, its interaction with existing AI frameworks, and its impact on performance remain open questions.

This research direction also raises several challenges that require careful investigation. These include understanding how deduplication interacts with accelerator memory (e.g., GPU or unified CPU/GPU memory), how to handle fine-tuned or partially shared models, and how to preserve isolation guarantees when memory pages are shared across tenants. Exploring these questions would help assess whether unikernels can play a practical role in future AI deployment scenarios and would extend the scope of this work beyond traditional cloud and edge workloads.

Function-Level Unikernels for Fine-Grained Isolation. A further research direction is the exploration of function-level decomposition of applications, where each function or group of functions could be isolated into its own unikernel instance. Such an approach would push isolation beyond library-level boundaries and could substantially reduce the attack surface by confining vulnerabilities to the minimal code required for one or a small group of functions. Designing this model would require new execution and communication primitives, including efficient inter-unikernel invocation and shared abstractions for passing arguments and return values. In particular, the design of a safe and efficient call-frame passing mechanism raises challenging questions related to both security and performance. While this approach has the potential to strengthen security guarantees by limiting the scope of memory-safety violations, it also introduces significant systems and compiler challenges. Investigating these trade-offs would provide valuable insight into the feasibility of ultra-fine-grained isolation in unikernel-based systems.

This page intentionally left blank.

BIBLIOGRAPHY

[1] Open Virtualization Alliance (OVA). *Kernel-based Virtual Machine*. https://linux-kvm.org, accessed 25/11/2024. 2023.

[2] Alexandru Agache, Marc Brooker, Andreea Florescu, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. "Firecracker: lightweight virtualization for serverless applications". In: *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation*. NSDI'20. Santa Clara, CA, USA: USENIX Association, 2020, pp. 419–434. ISBN: 978-1-939133-13-7.

[3] Zanyar Rzgar Ahmed, Shavan Askar, Diana Hayder Hussein, and Media Ali Ibrahim. "Fog Computing Challenges and Opportunities in IoT Networks: A Review". In: *Procedia Computer Science* 259 (2025). Sixth International Conference on Futuristic Trends in Networks and Computing Technologies (FTNCT06), held in Uttarakhand, India, pp. 1749–1764. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2025.04.130.

[4] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. "SAND: towards high-performance serverless computing". In: *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC '18. Boston, MA, USA: USENIX Association, 2018, pp. 923–935. ISBN: 978-1-931971-44-7.

[5] Delwar Alam, Moniruz Zaman, Tanjila Farah, Rummana Rahman, and M.-Shazzad Hosain. "Study of the Dirty Copy on Write, a Linux Kernel memory allocation vulnerability". In: *2017 International Conference on Consumer Electronics and Devices (ICCED)*. New York, NY, USA: IEEE, 2017, pp. 40–45. DOI: 10.1109/ICCED.2017.8019988.

[6] Grant Allen and Mike Owens. *The Definitive Guide to SQLite*. 2nd. Berkeley, CA, USA: Apress, 2010.

[7] Amazon: AWS. *Amazon Web Services (AWS)*. https://aws.amazon.com, accessed 19/05/2025. 2006.

[8] Thomas E. Anderson. "The case for application-specific operating systems". In: *Proceedings Third Workshop on Workstation Operating Systems*. Los Alami-

tos, CA, USA: IEEE Computer Society, Apr. 1992, pp. 92, 93, 94. DOI: 10.1109/WWOS.1992.275682.

[9] Andy Cedilnik, Bill Hoffman, Brad King, Ken Martin, Alexander Neundorf. *cmake*. https://cmake.org, accessed 25/01/2021. 2000.

[10] 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. Rennes, France: Association for Computing Machinery, 2022, pp. 730–746. ISBN: 978-1-4503-9162-7. DOI: 10.1145/3492321.3524270.

[11] Andrea Arcangeli, Izik Eidus, and Chris Wright. "Increasing memory density by using KSM". In: *Proceedings of the 2009 Linux Symposium*. Montréal, Canada: The Linux Kernel Organization, 2009, pp. 19–28.

[12] Andrea Arcangeli, Izik Eidus, and Chris Wright. *The Kernel SamePage Merging (Linux source code)*. https://github.com/torvalds/linux/blob/master/mm/ksm.c, accessed 11/05/2022. 2009.

[13] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. "A view of cloud computing". In: *Commun. ACM* 53.4 (Apr. 2010), pp. 50–58. ISSN: 0001-0782. DOI: 10.1145/1721654.1721672.

[14] Jeff Arnold and M. Frans Kaashoek. "Ksplice: automatic rebootless kernel updates". In: *Proceedings of the 4th ACM European Conference on Computer Systems*. EuroSys '09. Nuremberg, Germany: Association for Computing Machinery, 2009, pp. 187–198. ISBN: 978-1-60558-482-9. DOI: 10.1145/1519065.1519085.

[15] Vaggelis Atlidakis, Jeremy Andrus, Roxana Geambasu, Dimitris Mitropoulos, and Jason Nieh. "POSIX abstractions in modern operating systems: the old, the new, and the missing". In: *Proceedings of the Eleventh European Conference on Computer Systems*. EuroSys '16. London, United Kingdom: Association for Computing Machinery, 2016. ISBN: 978-1-4503-4240-7. DOI: 10.1145/2901318.2901350.

[16] The Capstone Authors. *Capstone The Ultimate Disassembler*. https://www.capstone-engine.org, accessed 11/11/2018. 2013.

[17] The Unikraft Authors. *nolibc: Unikraft's Lightweight Subset of the C Standard Library*. https://github.com/unikraft/unikraft/tree/staging/lib/nolibc, accessed 11/05/2022. 2013.

[18] The Unikraft Authors. *Unikraft - Musl*. https://github.com/unikraft/lib-musl/commits/staging/, accessed 25/01/2021. 2020.

[19] The Unikraft Authors. *Unikraft - Syscall Shim Layer*. https://unikraft.org/docs/internals/syscall-shim, accessed 25/01/2021. 2020.

[20] The Unikraft Authors. *Unikraft: Compatibility*. https://unikraft.org/docs/concepts/compatibility, accessed 16/08/2025. 2021.

[21] The Unikraft Authors. *Unikraft ELF Loader*. https://github.com/unikraft/app-elfloader, accessed 16/08/2025. 2023.

[22] Seungyeon Bae, Taehun Kim, Woomin Lee, and Youngjoo Shin. "Exploiting Memory Page Management in KSM for Remote Memory Deduplication Attack". In: *Information Security Applications: 24th International Conference, WISA 2023, Jeju Island, South Korea, August 23–25, 2023, Revised Selected Papers*. Jeju Island, Korea (Republic of): Springer-Verlag, 2023, pp. 244–256. ISBN: 978-981-99-8023-9. DOI: 10.1007/978-981-99-8024-6_19.

[23] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, and Philippe Suter. "Serverless Computing: Current Trends and Open Problems". In: *Research Advances in Cloud Computing*. Singapore: Springer, 2017, pp. 1–20. ISBN: 978-981-10-5026-8. DOI: 10.1007/978-981-10-5026-8_1.

[24] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. "A Survey of Symbolic Execution Techniques". In: *ACM Comput. Surv.* 51.3 (2018).

[25] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. "Byteweight: learning to recognize functions in binary code". In: *Proceedings of the 23rd USENIX Conference on Security Symposium*. SEC'14. San Diego, CA: USENIX Association, 2014, pp. 845–860. ISBN: 978-1-931971-15-7.

[26] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. "Xen and the art of virtualization". In: *SIGOPS Oper. Syst. Rev.* 37.5 (Oct. 2003), pp. 164–177. ISSN: 0163-5980. DOI: 10.1145/1165389.945462.

[27] Sean Barker, Timothy Wood, Prashant Shenoy, and Ramesh Sitaraman. "An empirical study of memory sharing in virtual machines". In: *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*. USENIX ATC'12. Boston, MA: USENIX Association, 2012, p. 25.

[28] Len Bass, Ingo Weber, and Liming Zhu. *DevOps: A Software Architect's Perspective*. 1st. Addison-Wesley Professional, 2015. ISBN: 978-0-13-404984-7.

[29] Fabrice Bellard. "QEMU, a fast and portable dynamic translator". In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC '05. Anaheim, CA: USENIX Association, 2005, p. 41.

[30] Charles-Henry Bertrand Van Ouytsel, Christophe Crochet, Khanh Huu The Dam, and Axel Legay. "Tool Paper - SEMA: Symbolic Execution Toolchain for Malware Analysis". In: *Risks and Security of Internet and Systems: 17th International Conference, CRiSIS 2022, Sousse, Tunisia, December 7-9, 2022, Revised Selected Papers*. Sousse, Tunisia: Springer-Verlag, 2022, pp. 62–68. ISBN: 978-3-031-31107-9. DOI: 10.1007/978-3-031-31108-6_5.

[31] Rovin Bhandari. *Implementation of a FTP client and server*. https://github.com/rovinbhandari/FTP, accessed 11/05/2022. 2012.

[32] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. "Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector". In: *2016 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2016, pp. 987–1004. DOI: 10.1109/SP.2016.63.

[33] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. "SELECT-a formal system for testing and debugging programs by symbolic execution". In: *SIGPLAN Not.* 10.6 (Apr. 1975), pp. 234–245. ISSN: 0362-1340. DOI: 10.1145/390016.808445.

[34] Alfred Bratterud, Alf-Andre Walla, Hårek Haugerud, Paal E Engelstad, and Kyrre Begnum. "IncludeOS: A minimal, resource efficient unikernel for cloud services". In: *Proceedings of the 7th IEEE International Conference on Cloud Computing Technology and Science*. CloudCom 2015. 2015.

[35] Eric A. Brewer. "Kubernetes and the path to cloud native". In: *Proceedings of the Sixth ACM Symposium on Cloud Computing*. SoCC '15. Kohala Coast, Hawaii: Association for Computing Machinery, 2015, p. 167. ISBN: 9781450336512. DOI: 10.1145/2806777.2809955.

[36] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. "Language models are few-shot learners". In: *Proceedings of the 34th International Conference on Neural Information Processing Systems*. NIPS '20. Vancouver, BC, Canada: Curran Associates Inc., 2020. ISBN: 9781713829546.

[37] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. "Disco: running commodity operating systems on scalable multiprocessors". In: *ACM Transactions on Computer Systems (TOCS)* 15.4 (Nov. 1997), pp. 412–447. ISSN: 0734-2071. DOI: 10.1145/265924.265930.

[38] Alexander Bulekov, Rasoul Jahanshahi, and Manuel Egele. "Saphire: Sandboxing PHP Applications with Tailored System Call Allowlists". In: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 2881–2898. ISBN: 978-1-939133-24-3.

[39] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. "SEUSS: skip redundant paths to make serverless fast". In: *Proceedings of the Fifteenth European Conference on Computer Systems*. EuroSys '20. Heraklion, Greece: Association for Computing Machinery, 2020. ISBN: 978-1-4503-6882-7. DOI: 10.1145/3342195.3392698.

[40] Claudio Canella, Mario Werner, Daniel Gruss, and Michael Schwarz. "Automating Seccomp Filter Generation for Linux Applications". In: *Proceedings of the 2021 on Cloud Computing Security Workshop*. CCSW '21. Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, pp. 139–151. ISBN: 978-1-4503-8653-1. DOI: 10.1145/3474123.3486762.

[41] Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. "The rise of serverless computing". In: *Commun. ACM* 62.12 (Nov. 2019), pp. 44–54. ISSN: 0001-0782. DOI: 10.1145/3368454.

[42] Shichao Chen and Mengchu Zhou. "Evolving Container to Unikernel for Edge Computing and Applications in Process Industry". In: *Processes* 9.2 (2021). ISSN: 2227-9717. DOI: 10.3390/pr9020351.

[43] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. "S2E: a platform for in-vivo multi-path analysis of software systems". In: *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XVI. Newport Beach, California, USA: Association for Computing Machinery, 2011, pp. 265–278. ISBN: 978-1-4503-0266-1. DOI: 10.1145/1950365.1950396.

[44] Siddharth Choudhuri. "A Case for Unikernels in IoT: Enhancing Security and Performance". In: *Internet of Things: Enabling Technologies, Security and Social Implications*. Ed. by Santosh Kumar Pani and Manjusha Pandey. Singapore: Springer Singapore, 2021, pp. 85–91. ISBN: 978-981-15-8621-7. DOI: 10.1007/978-981-15-8621-7_7.

[45] Cloudozer Organization. *Erlang on Xen*. https://github.com/cloudozer/ling, accessed 25/01/2021. 2015.

[46] Christian Collberg, John H. Hartman, Sridivya Babu, and Sharath K. Udupa. "Slinky: static linking reloaded". In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC '05. Anaheim, CA: USENIX Association, 2005, p. 34.

[47] The kernel development community. *Page table lock - The Linux Kernel documentation*. https://www.kernel.org/doc/html/v5.7/vm/split_page_table_lock.html, accessed 12/11/2025. 2011.

[48] Daniel Lezcano, Serge Hallyn, Stéphane Graber. *Linux Container*. https://linuxcontainers.org, accessed 10/04/2024. 2018.

[49] Wagner David and Drew Dean. "Intrusion detection via static analysis". In: *Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001*. 2001, pp. 156–168. DOI: 10.1109/SECPRI.2001.924296.

[50] Debian. *Debian Popularity Contest*. https://popcon.debian.org/, accessed 25/01/2021. 2015.

[51] Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P. Kemerlis. "sysfilter: Automated System Call Filtering for Commodity Software". In: *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. San Sebastian: USENIX Association, Oct. 2020, pp. 459–474. ISBN: 978-1-939133-18-2.

[52] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2019. arXiv: 1810.04805 [cs.CL]. URL: https://arxiv.org/abs/1810.04805.

[53] Will Dietz and Vikram Adve. "Software multiplexing: share your libraries and statically link them too". In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA (Oct. 2018). DOI: 10.1145/3276524.

[54] *dlclose, dlopen, dlmopen - open and close a shared object*. 6.03. https://man7.org/linux/man-pages/man3/dlopen.3.html, accessed 16/10/2025. Feb. 2023.

[55] Ulrich Drepper. "How To Write Shared Libraries". In: *Structure* 16 (2006), p. 2009.

[56] Aaron Drew. *A lightweight DNS-to-HTTPS proxy for the RFC 8484 DNS-over-HTTPS standard*. https://github.com/aarond10/https_dns_proxy, accessed 11/05/2022. 2016.

[57] Dong Du, Zhichao Hua, Yubin Xia, Binyu Zang, and Haibo Chen. "XPC: architectural support for secure and efficient cross process call". In: *Proceedings of the 46th International Symposium on Computer Architecture*. ISCA '19.

Phoenix, Arizona: Association for Computing Machinery, 2019, pp. 671–684. ISBN: 978-1-4503-6669-4. DOI: 10.1145/3307650.3322218.

[58] Jon Dugan, Seth Elliott, Bruce A. Mah, Jeff Poskanzer, and Kaustubh Prabhu. *iPerf - The ultimate speed test tool for TCP, UDP and SCTP*. https://iperf.fr/iperf-doc.php, accessed 08/01/2023. 2014.

[59] Bob Duncan, Andreas Happe, and Alfred Bratterud. "Cloud Cyber Security: Finding an Effective Approach with Unikernels". In: *Advances in Security in Computing and Communications*. Rijeka: IntechOpen, 2017. Chap. 2. DOI: 10.5772/67801.

[60] Dawson R. Engler, M. F. Kaashoek, and J. O'Toole. "Exokernel: an operating system architecture for application-level resource management". In: *SIGOPS Oper. Syst. Rev.* 29.5 (Dec. 1995), pp. 251–266. ISSN: 0163-5980. DOI: 10.1145/224057.224076.

[61] Rich Felker. *musl libc*. https://www.musl-libc.org, accessed 25/01/2021. 2011.

[62] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. "An updated performance comparison of virtual machines and Linux containers". In: *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2015, pp. 171–172. DOI: 10.1109/ISPASS.2015.7095802.

[63] Brad Fitzpatrick. *memcached*. http://memcached.org, accessed 25/01/2021. 2009.

[64] Apache Software Foundation. *Apache HTTP Server*. https://httpd.apache.org, accessed 11/05/2022. 1997.

[65] Free Software Foundation. *GNU Autotools*. https://www.gnu.org/software/automake/manual/html_node/Autotools-Introduction.html, accessed 25/01/2021. 1997.

[66] Free Software Foundation. *GNU linker LD (GNU Binutils)*. https://sourceware.org/binutils/docs/ld/, accessed 11/05/2022. 1998.

[67] The Linux Foundation. *The Yocto Project*. https://www.yoctoproject.org, accessed 25/01/2021. 2010.

[68] Free Software Foundation. *GNU C Library*. https://www.gnu.org/software/libc/, accessed 25/01/2021. 1988.

[69] Alexander Fuerst and Prateek Sharma. "FaasCache: keeping serverless computing alive with greedy-dual caching". In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '21. Virtual, USA: Association for Computing

Machinery, 2021, pp. 386–400. ISBN: 9781450383172. DOI: 10.1145/3445814.3446757.

[70] Gaulthier Gain. *Unikraft Static Binary Analysis Tool* (*Part of the Loupe Artifact*). https://github.com/unikraft/loupe/tree/staging/src/static-binary-analyser, accessed 17/08/2023. 2023.

[71] Gaulthier Gain. *Unikraft Tools*. https://github.com/gaulthiergain/tools, accessed 14/10/2023. 2023.

[72] Gaulthier Gain. *Custom Loader based on Firecracker* (*patches*). https://github.com/gaulthiergain/firecracker_custom_loader, accessed 13/08/2025. 2025.

[73] Gaulthier Gain, Cyril Soldani, Felipe Huici, and Laurent Mathy. "Want More Unikernels? Inflate Them!" In: *Proceedings of the 13th Symposium on Cloud Computing*. SoCC '22. San Francisco, California: Association for Computing Machinery, 2022, pp. 510–525. ISBN: 978-1-4503-9414-7. DOI: 10.1145/3542929.3563473.

[74] Jonathan Ganz and Sean Peisert. "ASLR: How Robust Is the Randomness?" In: *2017 IEEE Cybersecurity Development* (*SecDev*). 2017, pp. 34–41. DOI: 10.1109/SecDev.2017.19.

[75] Xing Gao, Zhongshu Gu, Mehmet Kayaalp, Dimitrios Pendarakis, and Haining Wang. "ContainerLeaks: Emerging Security Threats of Information Leakages in Container Clouds". In: *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (*DSN*). 2017, pp. 237–248. DOI: 10.1109/DSN.2017.49.

[76] Seyedhamed Ghavamnia, Tapti Palit, Azzedine Benameur, and Michalis Polychronakis. "Confine: Automated system call policy generation for container attack surface reduction". In: *Proceedings of the 23rd International Symposium on Research in Attacks, Intrusions and Defenses*. RAID'20. 2020.

[77] Git. *Git Basics - Tagging*. https://git-scm.com/book/en/v2/Git-Basics-Tagging, accessed 18/08/2025. 2005.

[78] Github. *Github*. https://github.com, accessed 12/05/2025. 2008.

[79] Will Glozer. *wrk - a HTTP benchmarking tool*. https://github.com/wg/wrk, accessed 08/01/2023. 2012.

[80] Tom Goethals, Merlijn Sebrechts, Mays Al-Naday, Bruno Volckaert, and Filip De Turck. "A Functional and Performance Benchmark of Lightweight Virtualization Platforms for Edge Computing". In: *2022 IEEE International Conference on Edge Computing and Communications* (*EDGE*). 2022, pp. 60–68. DOI: 10.1109/EDGE55608.2022.00020.

[81] Google. *Protocol Buffers Documentation*. https://protobuf.dev/, accessed 29/05/2025. 2001.

[82] Google. *Google Cloud*. https://cloud.google.com/, accessed 19/05/2025. 2011.

[83] Google LLC. *gVisor: Container Runtime Sandbox*. https://gvisor.dev. Accessed: 2026-04-23. 2019.

[84] Aaron Grattafiori. *Understanding and Hardening Linux Containers*. https://tinyurl.com/nccgroup2016, accessed 11/05/2022. 2016.

[85] Brendan Gregg. *Performance analysis tools based on Linux perf_events (aka perf) and ftrace*. https://github.com/brendangregg/perf-tools, accessed 23/06/2025. 2014.

[86] Richard Grisenthwaite, Graeme Barnes, Robert N. M. Watson, Simon W. Moore, Peter Sewell, and Jonathan Woodruff. "The Arm Morello Evaluation Platform-Validating CHERI-Based Security in a High-Performance System". In: *IEEE Micro* 43.3 (2023), pp. 50–57. DOI: 10.1109/MM.2023.3264676.

[87] Daniel Gruss, David Bidner, and Stefan Mangard. "Practical Memory Deduplication Attacks in Sandboxed Javascript". In: *Computer Security – ESORICS 2015: 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21-25, 2015, Proceedings, Part I*. Vienna, Austria: Springer, 2015, pp. 108–122. ISBN: 978-3-319-24173-9. DOI: 10.1007/978-3-319-24174-6_6.

[88] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. "Bringing the web up to speed with WebAssembly". In: *SIGPLAN Not.* 52.6 (June 2017), pp. 185–200. ISSN: 0362-1340. DOI: 10.1145/3140587.3062363.

[89] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. "The performance of μ-kernel-based systems". In: *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*. SOSP '97. Saint Malo, France: Association for Computing Machinery, 1997, pp. 66–77. ISBN: 0897919165. DOI: 10.1145/268998.266660.

[90] Philip Hazel and Zoltan Herczeg. *PCRE - Perl Compatible Regular Expressions*. https://github.com/gaulthiergain/lib-pcre, accessed 19/05/2025. 2025.

[91] Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji. "Intrusion detection using sequences of system calls". In: *Journal of Computer Security* 6.3 (Aug. 1998), pp. 151–180. ISSN: 0926-227X.

[92] W.E. Howden. "Symbolic Testing and the DISSECT Symbolic Evaluation System". In: *IEEE Transactions on Software Engineering* SE-3.4 (1977), pp. 266–278. DOI: 10.1109/TSE.1977.231144.

[93] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3B*. Intel Corporation. May 2019.

[94] Charles Jacobsen, Muktesh Khole, Sarah Spall, Scotty Bauer, and Anton Burtsev. "Lightweight capability domains: towards decomposing the Linux kernel". In: *Proceedings of the 8th Workshop on Programming Languages and Operating Systems*. PLOS '15. Monterey, California: Association for Computing Machinery, 2015, pp. 8–14. ISBN: 978-1-4503-3942-1. DOI: 10.1145/2818302.2818307.

[95] Omar Jarkas, Ryan Ko, Naipeng Dong, and Redowan Mahmud. "A Container Security Survey: Exploits, Attacks, and Defenses". In: *ACM Comput. Surv.* 57.7 (Feb. 2025). ISSN: 0360-0300. DOI: 10.1145/3715001.

[96] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, João Carreira, Karl Krauth, Neeraja Jayant Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. "Cloud Programming Simplified: A Berkeley View on Serverless Computing". In: *CoRR* abs/1902.03383 (2019). arXiv: 1902.03383. URL: http://arxiv.org/abs/1902.03383.

[97] Antti Kantee and Justin Cormack. "Rump Kernels No OS? No Problem!" In: *USENIX; login: magazine* (2014).

[98] Max Kellermann. *The Dirty Pipe Vulnerability*. https://dirtypipe.cm4all.com, accessed 11/05/2022. 2022.

[99] Mazen Kharbutli, Xiaowei Jiang, Yan Solihin, Guru Venkataramani, and Milos Prvulovic. "Comprehensively and Efficiently Protecting the Heap". In: *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XII. San Jose, California, USA: Association for Computing Machinery, 2006, pp. 207–218. DOI: 10.1145/1168857.1168884.

[100] James C. King. "Symbolic execution and program testing". In: *Communications of the ACM* 19.7 (July 1976), pp. 385–394. ISSN: 0001-0782. DOI: 10.1145/360248.360252.

[101] Avi Kivity, Dor Laor Glauber Costa, and Pekka Enberg. "OSv - Optimizing the Operating System for Virtual Machines". In: *Proceedings of the 2014 USENIX Annual Technical Conference*. ATC'14. 2014.

[102] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. "Spectre attacks: exploiting speculative execution". In: *Commun. ACM* 63.7 (June 2020), pp. 93–101. ISSN: 0001-0782. DOI: 10.1145/3399742.

[103] Daniel Kroening, Natasha Sharygina, Stefano Tonetta, Aliaksei Tsitovich, and Christoph M. Wintersteiger. "Loop Summarization Using Abstract Transformers". In: *Automated Technology for Verification and Analysis*. Ed. by Sungdeok (Steve) Cha, Jin-Young Choi, Moonzoo Kim, Insup Lee, and Mahesh Viswanathan. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 111–125. ISBN: 978-3-540-88387-6.

[104] KubeVirt Project. *KSM Management*. https://kubevirt.io/user-guide/cluster_admin/ksm/, accessed 24/04/2026. 2017.

[105] Simon Kuenzer, Vlad-Andrei Bădoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Ştefan Teodorescu, Costi Răducanu, Cristian Banu, Laurent Mathy, Răzvan Deaconescu, Costin Raiciu, and Felipe Huici. "Unikraft: fast, specialized unikernels the easy way". In: *Proceedings of the Sixteenth European Conference on Computer Systems*. EuroSys '21. Online Event, United Kingdom: Association for Computing Machinery, 2021, pp. 376–394. ISBN: 978-1-4503-8334-9. DOI: 10.1145/3447786.3456248.

[106] Simon Kuenzer, Anton Ivanov, Filipe Manco, Jose Mendes, Yuri Volchkov, Florian Schmidt, Kenichi Yasukata, Michio Honda, and Felipe Huici. "Unikernels Everywhere: The Case for Elastic CDNs". In: *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE'17. 2017.

[107] Simon Kuenzer, Joao Martins, Mohamed Ahmed, and Felipe Huici. "Towards Minimalistic, Virtualized Content Caches with Minicache". In: *Proceedings of the 2013 ACM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*. HotMiddlebox'13. ACM, 2013, pp. 13–18. ISBN: 978-1-4503-2574-5. DOI: 10.1145/2535828.2535832.

[108] Raula Gaikovina Kula, Daniel M. German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. "Do developers update their library dependencies?" In: *Empirical Software Engineering* 23.1 (Feb. 2018), pp. 384–417. ISSN: 1382-3256. DOI: 10.1007/s10664-017-9521-5.

[109] Hsuan-Chi Kuo, Dan Williams, Ricardo Koller, and Sibin Mohan. "A Linux in Unikernel Clothing". In: *Proceedings of the 15th European Conference on Computer Systems*. EuroSys'20. 2020.

[110] Serge Lamikhov. *ELFIO - C++ library for reading and generating ELF files*. https://github.com/serge1/ELFIO. 2012.

[111] Stefan Lankes, Simon Pickartz, and Jens Breitbart. "HermitCore: A Unikernel for Extreme Scale Computing". In: *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers*. ROSS '16. Kyoto, Japan: Association for Computing Machinery, 2016. ISBN: 978-1-4503-4387-9. DOI: 10.1145/2931088.2931093.

[112] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*. CGO '04. Palo Alto, California: IEEE Computer Society, 2004, pp. 75–86. DOI: 10.1109/CGO.2004.1281665.

[113] *ld.so, ld-linux.so - dynamic linker/loader*. 6.15. https://man7.org/linux/man-pages/man8/ld.so.8.html, accessed 16/10/2025. July 2025.

[114] *ldd - print shared object dependencies*. 6.03. https://man7.org/linux/man-pages/man1/ldd.1.html, accessed 16/10/2025. Feb. 2023.

[115] Chung-Chieh Lee and Der Tsai Lee. "A simple on-line bin-packing algorithm". In: *J. ACM* 32.3 (July 1985), pp. 562–572. ISSN: 0004-5411. DOI: 10.1145/3828.3833.

[116] Hugo Lefeuvre, Vlad-Andrei Bădoiu, Alexander Jung, Stefan Lucian Teodorescu, Sebastian Rauch, Felipe Huici, Costin Raiciu, and Pierre Olivier. "FlexOS: towards flexible OS isolation". In: *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '22. Lausanne, Switzerland: Association for Computing Machinery, 2022, pp. 467–482. ISBN: 978-1-4503-9205-1. DOI: 10.1145/3503222.3507759.

[117] Hugo Lefeuvre, Gaulthier Gain, Vlad-Andrei Bădoiu, Daniel Dinca, Vlad-Radu Schiller, Costin Raiciu, Felipe Huici, and Pierre Olivier. "Loupe: Driving the Development of OS Compatibility Layers". In: *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. ASPLOS '24. La Jolla, CA, USA: Association for Computing Machinery, 2024, pp. 249–267. ISBN: 979-8-4007-0372-0. DOI: 10.1145/3617232.3624861.

[118] Hugo Lefeuvre, Gaulthier Gain, Daniel Dinca, Alexander Jung, Simon Kuenzer, Vlad-Andrei Badoiu, Razvan Deaconescu, Laurent Mathy, Costin Raiciu, Pierre Olivier, and Felipe Huici. "Unikraft and the Coming of Age of Unikernels". In: *USENIX; login* (2021).

[119] Ian M. Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, and Eoin Hyden. "The design and implementation of an operating system to support distributed multimedia applications". In: *IEEE J.Sel. A. Commun.* 14.7 (Sept. 1996), pp. 1280–1297. ISSN: 0733-8716. DOI: 10.1109/49.536480.

[120] Yihua Liao and V. Rao Vemuri. "Using Text Categorization Techniques for Intrusion Detection". In: *Proceedings of the 11th USENIX Security Symposium*. USA: USENIX Association, 2002, pp. 51–59.

[121] Linux Kernel Documentation. *Overlay Filesystem Documentation*. https://docs.kernel.org/filesystems/overlayfs.html, accessed 05/05/2026. 2011.

[122] Google LLC. *The Go Programming Language*. https://go.dev, accessed 05/05/2026. 2009.

[123] LLVM Developer Group. *CLANG Compiler User's Manual*. https://clang.llvm.org/docs/UsersManual.html, accessed 15/08/2025. 2007.

[124] Hong jiu Lu, Michael Matz, Milind Girkar, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. *System V Application Binary Interface – AMD64 Architecture Processor Supplement (With LP64 and ILP32 Programming Models)*. 2018. URL: https://github.com/hjl-tools/x86-psABI/wiki/x86-64-psABI-1.0.pdf.

[125] Costin Lupu, Andrei Albisoru, Radu Nichita, Doru-Florin Blânzeanu, Mihai Pogonaru, Răzvan Deaconescu, and Costin Raiciu. "Nephele: Extending Virtualization Environments for Cloning Unikernel-Based VMs". In: *Proceedings of the Eighteenth European Conference on Computer Systems*. EuroSys '23. Rome, Italy: Association for Computing Machinery, 2023, pp. 574–589. ISBN: 978-1-4503-9487-1. DOI: 10.1145/3552326.3587454.

[126] Theo Lynn, Pierangelo Rosati, Arnaud Lejeune, and Vincent Emeakaroha. "A Preliminary Review of Enterprise Serverless Cloud Computing (FaaS) Platforms". In: *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. New York, NY, USA: IEEE, 2017, pp. 162–169. DOI: 10.1109/CloudCom.2017.15.

[127] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. "Unikernels: library operating systems for the cloud". In: *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '13. Houston, Texas, USA: Association for Computing Machinery, 2013, pp. 461–472. ISBN: 978-1-4503-1870-9. DOI: 10.1145/2451116.2451167.

[128] Anil Madhavapeddy and David J. Scott. "Unikernels: the rise of the virtual library operating system". In: *Commun. ACM* 57.1 (Jan. 2014), pp. 61–69. ISSN: 0001-0782. DOI: 10.1145/2541883.2541895.

[129] *madvise - give advice about use of memory*. 6.03. https://man7.org/linux/man-pages/man2/madvise.2.html, accessed 08/01/2023. July 2023.

[130] Daniel J. Magenheimer, Chris Mason, Dave McCracken, Kurt Hackel, and Oracle Corp. *Transcendent Memory and Linux*. https://www.kernel.org/doc/ols/2009/ols2009-pages-191-200.pdf, Online. 2009.

[131] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. "My VM is Lighter (and Safer) than your Container". In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP '17. Shanghai, China: Association for Computing Machinery, 2017, pp. 218–233. ISBN: 978-1-4503-5085-3. DOI: 10.1145/3132747.3132763.

[132] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. "ClickOS and the art of network function virtualization". In: *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. NSDI'14. Seattle, WA: USENIX Association, 2014, pp. 459–473. ISBN: 978-1-931971-09-6.

[133] Mark McGranaghan and Eli Bendersky. *gobyexample*. https://gobyexample.com, accessed 12/05/2023. 2012.

[134] Garrett McGrath and Paul R. Brenner. "Serverless Computing: Design, Implementation, and Performance". In: *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. New York, NY, USA: IEEE, 2017, pp. 405–410. DOI: 10.1109/ICDCSW.2017.36.

[135] Dirk Merkel. "Docker: lightweight Linux containers for consistent development and deployment". In: *Linux J.* 2014.239 (Mar. 2014). ISSN: 1075-3583.

[136] Spencer Michaels and Jeff Dileo. *NCC Group Whitepaper Assessing Unikernel Security*. https://people.cs.pitt.edu/~babay/courses/cs3551/projects/SCADA_Unikernel/NCC_Group-Assessing_Unikernel_Security.pdf, accessed 29/05/2022. 2019.

[137] David S. Miller. *Cache and TLB Flushing Under Linux*. https://docs.kernel.org/core-api/cachetlb.html, accessed 30/05/2025. 2025.

[138] Konrad Miller, Fabian Franz, Thorsten Groeninger, Marc Rittinghaus, Marius Hillenbrand, and Frank Bellosa. "KSM++: Using I/O-based hints to make memory-deduplication scanners more efficient". In: *Proceedings of the AS-*

PLOS Workshop on Runtime Environments, Systems, Layering and Virtualized Environments (RESoLVE'12). 2012.

[139] Konrad Miller, Fabian Franz, Marc Rittinghaus, Marius Hillenbrand, and Frank Bellosa. "XLH: more effective memory deduplication scanners through cross-layer hints". In: *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*. USENIX ATC'13. San Jose, CA: USENIX Association, 2013, pp. 279–290.

[140] Grzegorz Miłós, Derek G. Murray, Steven Hand, and Michael A. Fetterman. "Satori: enlightened page sharing". In: *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*. USENIX'09. San Diego, California: USENIX Association, 2009, p. 1.

[141] Masanori Misono, Peter Okelmann, Charalampos Mainas, and Pramod Bhatotia. "uIO: Lightweight and Extensible Unikernels". In: *Proceedings of the 2024 ACM Symposium on Cloud Computing*. SoCC '24. Redmond, WA, USA: Association for Computing Machinery, 2024, pp. 580–599. ISBN: 979-8-4007-1286-9. DOI: 10.1145/3698038.3698518.

[142] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. "Agile cold starts for scalable serverless". In: *Proceedings of the 11th USENIX Conference on Hot Topics in Cloud Computing*. HotCloud'19. Renton, WA, USA: USENIX Association, 2019, p. 21.

[143] Darren Mutz, Fredrik Valeur, Giovanni Vigna, and Christopher Kruegel. "Anomalous system call detection". In: *ACM Trans. Inf. Syst. Secur.* 9.1 (Feb. 2006), pp. 61–93. ISSN: 1094-9224. DOI: 10.1145/1127345.1127348.

[144] Djob Mvondo, Mathieu Bacou, Kevin Nguetchouang, Lucien Ngale, Stéphane Pouget, Josiane Kouam, Renaud Lachaize, Jinho Hwang, Tim Wood, Daniel Hagimont, Noël De Palma, Bernabé Batchakui, and Alain Tchana. "OFC: an opportunistic caching system for FaaS platforms". In: *Proceedings of the Sixteenth European Conference on Computer Systems*. EuroSys '21. Online Event, United Kingdom: Association for Computing Machinery, 2021, pp. 228–244. ISBN: 978-1-4503-8334-9. DOI: 10.1145/3447786.3456239.

[145] Vivek Narasayya and Surajit Chaudhuri. "Multi-Tenant Cloud Data Services: State-of-the-Art, Challenges and Opportunities". In: *Proceedings of the 2022 International Conference on Management of Data*. SIGMOD '22. Philadelphia, PA, USA: Association for Computing Machinery, 2022, pp. 2465–2473. ISBN: 9781450392495. DOI: 10.1145/3514221.3522566.

[146] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. "SOCK: Rapid Task Provisioning with Serverless-Optimized Containers". In: *2018 USENIX Annual*

Technical Conference (*USENIX ATC 18*). Boston, MA: USENIX Association, July 2018, pp. 57–70. ISBN: 978-1-931971-44-7.

[147] Phil Oester. *Dirty COW (CVE-2016-5195)*. https://access.redhat.com/security/cve/cve-2016-5195, accessed 11/05/2022. 2016.

[148] Pierre Olivier, Daniel Chiba, Stefan Lankes, Changwoo Min, and Binoy Ravindran. "A binary-compatible unikernel". In: *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE 2019. Providence, RI, USA: Association for Computing Machinery, 2019, pp. 59–73. ISBN: 978-1-4503-6020-3. DOI: 10.1145/3313808.3313817.

[149] Inc. OpenSSL Foundation. *OpenSSL: Cryptography and SSL/TLS Toolkit*. https://www.openssl.org, accessed 29/05/2024. 1998.

[150] Samuel Ortiz. *Measuring Firecracker boot time*. https://gist.github.com/sameo/0647d6aaa36e73e6b536b51c29db1ead, accessed 30/05/2025. 2019.

[151] Rodney Owens and Weichao Wang. " Non-interactive OS finger-printing through memory de-duplication technique in virtual machines ". In: *IEEE International Performance Computing and Communications Conference*. Los Alamitos, CA, USA: IEEE Computer Society, Nov. 2011, pp. 1–8. DOI: 10.1109/PCCC.2011.6108094.

[152] Shankara Pailoor, Xinyu Wang, Hovav Shacham, and Isil Dillig. "Automated policy synthesis for system call sandboxing". In: *Proceedings of the ACM on Programming Languages* 4.OOPSLA (Nov. 2020). DOI: 10.1145/3428203.

[153] Ivan Pashchenko, Duc-Ly Vu, and Fabio Massacci. "A Qualitative Study of Dependency Management and Its Security Implications". In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. CCS '20. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 1513–1531. ISBN: 978-1-4503-7089-9. DOI: 10.1145/3372297.3417232.

[154] Luís Pina, Anastasios Andronidis, Michael Hicks, and Cristian Cadar. "MVEDSUA: Higher Availability Dynamic Software Updates via Multi-Version Execution". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '19. Providence, RI, USA: Association for Computing Machinery, 2019, pp. 573–585. ISBN: 978-1-4503-6240-5. DOI: 10.1145/3297858.3304063.

[155] Max Plauth, Lena Feinbube, and Andreas Polze. "A Performance Survey of Lightweight Virtualization Techniques". In: *Proceedings of the 6th European Conference on Service-Oriented and Cloud Computing*. ICN 2017. Springer. 2017.

[156] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. "Rethinking the library OS from the top down". In: *Proceed-*

ings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems. ASPLOS XVI. Newport Beach, California, USA: Association for Computing Machinery, 2011, pp. 291–304. ISBN: 978-1-4503-0266-1. DOI: 10.1145/1950365.1950399.

[157] Proxmox. *Kernel Same-page Merging (KSM)*. https://pve.proxmox.com/wiki/Kernel_Samepage_Merging_(KSM), accessed 24/04/2026. 2008.

[158] Wei Qiu, Marcin Copik, Yun Wang, Alexandru Calotoiu, and Torsten Hoefler. " User-guided Page Merging for Memory Deduplication in Serverless Systems ". In: *2023 IEEE International Conference on Big Data (BigData)*. Los Alamitos, CA, USA: IEEE Computer Society, Dec. 2023, pp. 159–169. DOI: 10.1109/BigData59044.2023.10386487.

[159] Shashank Rachamalla, Debadatta Mishra, and Purushottam Kulkarni. "Share-o-meter: An empirical analysis of KSM based memory sharing in virtualized systems". In: *20th Annual International Conference on High Performance Computing*. 2013, pp. 59–68. DOI: 10.1109/HiPC.2013.6799096.

[160] Alessandro Randazzo and Ilenia Tinnirello. "Kata Containers: An Emerging Architecture for Enabling MEC Services in Fast and Secure Way". In: *Proceedings of the 6th International Conference on Internet of Things: Systems, Management and Security*. IOTSMS'19. IEEE. 2019, pp. 209–214.

[161] Ali Raza, Thomas Unger, Matthew Boyd, Eric B Munson, Parul Sohal, Ulrich Drepper, Richard Jones, Daniel Bristot De Oliveira, Larry Woodman, Renato Mancuso, Jonathan Appavoo, and Orran Krieger. "Unikernel Linux (UKL)". In: *Proceedings of the 18th European Conference on Computer Systems*. EuroSys'23. 2023.

[162] RedHat. *Newlib: a C library intended for use on embedded systems*. https://sourceware.org/newlib/, accessed 12/12/2021. 2004.

[163] Will Reese. "Nginx: the high-performance web server and reverse proxy". In: *Linux J.* 2008.173 (Sept. 2008). ISSN: 1075-3583.

[164] Steven Rostedt. *ftrace - Function Tracer*. https://www.kernel.org/doc/html/v5.0/trace/ftrace.html, accessed 29/05/2025. 2008.

[165] Rust Foundation. *The Rust Programming Language*. https://www.rust-lang.org, accessed 05/05/2026. 2006.

[166] Jyotiprakash Sahoo, Subasish Mohapatra, and Radha Lath. "Virtualization: A Survey on Concepts, Taxonomy and Associated Security Issues". In: *2010 Second International Conference on Computer and Network Technology*. 2010, pp. 222–226. DOI: 10.1109/ICCNT.2010.49.

[167] Salvatore Sanfilippo. *redis*. https://redis.io/, accessed 25/01/2021. 2009.

[168] Vasily A. Sartakov, Lluís Vilanova, Munir Geden, David Eyers, Takahiro Shinagawa, and Peter Pietzuch. "ORC: Increasing Cloud Memory Density via Object Reuse with Capabilities". In: *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. Boston, MA: USENIX Association, July 2023, pp. 573–587. ISBN: 978-1-939133-34-2.

[169] Vasily A. Sartakov, Lluís Vilanova, and Peter Pietzuch. "CubicleOS: A Library OS with Software Componentisation for Practical Isolation". In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '21. Virtual, USA: Association for Computing Machinery, 2021, pp. 546–558. ISBN: 978-1-4503-8317-2. DOI: 10.1145/3445814.3446731.

[170] Savannah. *LwIP - A Lightweight TCP/IP stack*. https://savannah.nongnu.org/projects/lwip/, accessed 12/12/2017. 2002.

[171] Divyanshu Saxena, Tao Ji, Arjun Singhvi, Junaid Khalid, and Aditya Akella. "Memory Deduplication for Serverless Computing with Medes". In: *Proceedings of the Seventeenth European Conference on Computer Systems*. EuroSys '22. Rennes, France: Association for Computing Machinery, 2022, pp. 714–729. ISBN: 978-1-4503-9162-7. DOI: 10.1145/3492321.3524272.

[172] David Seal. *ARM Architecture Reference Manual*. 2nd. USA: Addison-Wesley Longman Publishing Co., Inc., 2000. ISBN: 978-0-201-73719-6.

[173] *seccomp - operate on Secure Computing state of the process*. 6.03. https://man7.org/linux/man-pages/man2/seccomp.2.html, accessed 31/07/2023. Feb. 2023.

[174] Kosta Serebryany. "Continuous Fuzzing with libFuzzer and AddressSanitizer". In: *2016 IEEE Cybersecurity Development (SecDev)*. 2016, pp. 157–157. DOI: 10.1109/SecDev.2016.043.

[175] Sergii Iefremov, David Björklund, Ariel Abreu. *Javascript Library Operating System For The Cloud*. http://runtimejs.org, accessed 25/01/2021. 2014.

[176] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. "On the Effectiveness of Address-Space Randomization". In: *Proceedings of CCS 2004*. Ed. by Birgit Pfitzmann and Peng Liu. ACM Press, Oct. 2004, pp. 298–307.

[177] 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. Columbus, OH, USA: Association for Computing Machinery, 2019, pp. 1063–1075. ISBN: 978-1-4503-6938-1. DOI: 10.1145/3352460.3358296.

[178] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. "Serverless in the wild: characterizing and optimizing the serverless workload at a large cloud provider". In: *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC'20. USA: USENIX Association, 2020. ISBN: 978-1-939133-14-4.

[179] Simon Shillaker and Peter Pietzuch. "FAASM: lightweight isolation for efficient stateful serverless computing". In: *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC'20. USA: USENIX Association, 2020. ISBN: 978-1-939133-14-4.

[180] Youngjoo Shin, Dongyoung Koo, and Junbeom Hur. "A Survey of Secure Data Deduplication Schemes for Cloud Storage Systems". In: *ACM Comput. Surv.* 49.4 (Jan. 2017). ISSN: 0360-0300. DOI: 10.1145/3017428.

[181] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis". In: *IEEE Symposium on Security and Privacy*. 2016.

[182] *The Solo5 Unikernel*. https://github.com/Solo5/solo5, accessed 25/11/2019. 2017.

[183] *The Unikernel and MicroVM Compilation and Deployment Platform*. https://github.com/solo-io/unik, accessed 27/11/2017. 2017.

[184] Steven Knight. *SCons*. https://scons.org, accessed 25/01/2021. 2001.

[185] *strace - Linux syscall tracer*. 6.1. https://man7.org/linux/man-pages/man1/strace.1.html, accessed 16/10/2022. July 2022.

[186] Suriya Subramanian, Michael Hicks, and Kathryn S. McKinley. "Dynamic software updates: a VM-centric approach". In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '09. Dublin, Ireland: Association for Computing Machinery, 2009, pp. 1–12. ISBN: 978-1-60558-392-1. DOI: 10.1145/1542476.1542478.

[187] Sari Sultan, Imtiaz Ahmad, and Tassos Dimitriou. "Container Security: Issues, Challenges, and the Road Ahead". In: *IEEE Access* 7 (2019), pp. 52976–52996. DOI: 10.1109/ACCESS.2019.2911732.

[188] Kuniyasu Suzaki, Kengo Iijima, Toshiki Yagi, and Cyrille Artho. "Memory Deduplication as a Threat to the Guest OS". In: *Proceedings of the Fourth European Workshop on System Security*. EUROSEC '11. Salzburg, Austria: Associa-

tion for Computing Machinery, 2011. ISBN: 978-1-4503-0613-3. DOI: 10.1145/1972551.1972552.

[189] Kuniyasu Suzaki, Kengo Iijima, Toshiki Yagi, and Cyrille Artho. "Implementation of a memory disclosure attack on memory deduplication of virtual machines". In: *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* 96.1 (2013), pp. 215–224.

[190] Systems and Machine Learning group (NECLAB). *A Micropython Unikernel*. https://github.com/sysml/minipython, accessed 25/01/2021. 2015.

[191] Byungchul Tak, Canturk Isci, Sastry Duri, Nilton Bila, Shripad Nadgowda, and James Doran. "Understanding Security Implications of Using Containers in the Cloud". In: *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, July 2017, pp. 313–319. ISBN: 978-1-931971-38-6.

[192] Bo Tan, Haikun Liu, Jia Rao, Xiaofei Liao, Hai Jin, and Yu Zhang. "Towards Lightweight Serverless Computing via Unikernel as a Function". In: *2020 IEEE/ACM 28th International Symposium on Quality of Service, IWQoS 2020* (2020). DOI: 10.1109/IWQoS49365.2020.9213020.

[193] Willy Tarreau. *Haproxy, The Reliable, High Performance TCP/HTTP Load Balancer*. https://www.haproxy.org, accessed 11/05/2022. 2001.

[194] The SQLite Development Team. *SQLite Speedtest*. https://github.com/sqlite/sqlite/blob/3d24637325188c1ed9db46e5bb23ab5d747ad29f/test/speedtest1.c, accessed 29/05/2025. 2024.

[195] Selome Kostentinos Tesfatsion, Cristian Klein, and Johan Tordsson. "Virtualization Techniques Compared: Performance, Resource, and Power Usage Overheads in Clouds". In: *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*. ICPE '18. Berlin, Germany: Association for Computing Machinery, 2018, pp. 145–156. ISBN: 9781450350952. DOI: 10.1145/3184407.3184414.

[196] The GNU Project. *Optimize Options (Using the GNU Compiler Collection (GCC))*. https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html, accessed 19/05/2025. 2023.

[197] The Go Authors. *The Go Programming Language*. https://go.dev, accessed 19/05/2025. 2009.

[198] The NanoVMS Authors. *NanoVMs*. https://nanovms.com, accessed 19/05/2025. 2023.

[199] The SQLite Consortium. *SQLite*. https://www.sqlite.org, accessed 11/07/2022. 2007.

[200] The Unikraft Authors. *ukboot: Unikraft bootstrapping*. https://github.com/unikraft/unikraft/tree/staging/lib/ukboot, accessed 24/10/2023. 2017.

[201] The Unikraft Authors. *Unikraft application repository: Applications supported by the Unikraft libOS*. https://github.com/orgs/unikraft/repositories, accessed 08/01/2023. 2017.

[202] The Unikraft Authors. *Unikraft port of pthread-embedded, an embedded pthread library*. https://github.com/unikraft/lib-pthread-embedded, accessed 19/05/2025. 2019.

[203] The Unikraft Authors. *Unikraft port of Python 3*. https://github.com/unikraft/lib-python3/, accessed 19/05/2025. 2019.

[204] The Unikraft Authors. *UnikraftCloud: True Serverless*. https://unikraft.cloud, accessed 29/05/2025. 2024.

[205] Romain Thomas. *LIEF - Library to Instrument Executable Formats*. https://lief-project.github.io, accessed 11/05/2022. 2017.

[206] TIS Committee. *Tool Interface Standard (TIS) Formats Specification for Windows (Version 1.0)*. https://openwatcom.org/ftp/devel/docs/pe_and_symbols.pdf, accessed 15/10/2025. 1993.

[207] TIS Committee. *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification (Version 1.2)*. May. TIS Committee. 1995, p. 106. ISBN: 978-1-4503-0261-6.

[208] Salvatore Tomaselli. *Minimal HTTP server to share your files*. https://ltworf.codeberg.page/weborf/, accessed 11/05/2022. 2008.

[209] Andrei Topala. *Unikraft (Release v0.8.0 Enceladus) with firecracker-mmio-0.8 support*. https://github.com/Krechals/unikraft/tree/firecracker-mmio-0.8, accessed 11/05/2022. 2022.

[210] Linus Torvalds. *The Linux Kernel Archives*. https://www.kernel.org, accessed 24/10/2023. 2011.

[211] Sam Trenholme. *MaraDNS: A small open-source DNS server*. https://github.com/samboy/MaraDNS/, accessed 11/05/2022. 2014.

[212] Chia-Che Tsai, Bhushan Jain, Nafees Ahmed Abdul, and Donald E. Porter. "A study of modern Linux API usage and compatibility: what to support when you're supporting". In: *Proceedings of the Eleventh European Conference on Computer Systems*. EuroSys '16. London, United Kingdom: Association for Computing Machinery, 2016. ISBN: 978-1-4503-4240-7. DOI: 10.1145/2901318.2901341.

[213] Chia-Che Tsai, Donald E. Porter, and Mona Vij. "Graphene-SGX: a practical library OS for unmodified applications on SGX". In: *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC '17. Santa Clara, CA, USA: USENIX Association, 2017, pp. 645–658. ISBN: 978-1-931971-38-6.

[214] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. "Attention is all you need". In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. NIPS'17. Long Beach, California, USA: Curran Associates Inc., 2017, pp. 6000–6010. ISBN: 9781510860964.

[215] Anthony Velte and Toby Velte. *Microsoft Virtualization with Hyper-V*. 1st ed. USA: McGraw-Hill, Inc., 2009. ISBN: 978-0-07-161403-0.

[216] VMware. *Fusion and Workstation | VMware*. https://www.vmware.com/products/desktop-hypervisor/workstation-and-fusion, accessed 25/11/2024. 2020.

[217] William Von Hagen. *The Definitive Guide to GCC, Second Edition*. USA: Apress, 2006. ISBN: 978-1-59059-585-5.

[218] Carl A. Waldspurger. "Memory Resource Management in VMware ESX Server". In: *SIGOPS Oper. Syst. Rev.* 36.SI (Dec. 2003), pp. 181–194. ISSN: 0163-5980. DOI: 10.1145/844128.844146.

[219] Zhiyuan Wan, David Lo, Xin Xia, and Liang Cai. "Practical and effective sandboxing for Linux containers". English. In: *Empirical Software Engineering* 24.6 (Dec. 2019), pp. 4034–4070. ISSN: 1382-3256. DOI: 10.1007/s10664-019-09737-2.

[220] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. "Peeking Behind the Curtains of Serverless Platforms". In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, July 2018, pp. 133–146. ISBN: 978-1-939133-01-4.

[221] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. "Ramblr: Making Reassembly Great Again". English (US). In: *24th Annual Network and Distributed System Security Symposium, NDSS 2017*. 24th Annual Network and Distributed System Security Symposium, NDSS 2017. The Internet Society, 2017. DOI: 10.14722/ndss.2017.23225.

[222] Richard Wartell, Yan Zhou, Kevin W. Hamlen, Murat Kantarcioglu, and Bhavani Thuraisingham. "Differentiating code from data in x86 binaries". In: *Proceedings of the 2011 European Conference on Machine Learning and Knowl-*

edge Discovery in Databases - Volume Part III. ECML PKDD'11. Athens, Greece: Springer-Verlag, 2011, pp. 522–536. ISBN: 978-3-642-23807-9.

[223] Andrew Waterman, Yunsup Lee, Rimas Avizienis, Henry Cook, David Patterson, and Krste Asanovic. "The RISC-V instruction set". In: *2013 IEEE Hot Chips 25 Symposium (HCS)*. 2013, pp. 1–1. DOI: 10.1109/HOTCHIPS.2013.7478332.

[224] Matthias Wenzl, Georg Merzdovnik, Johanna Ullrich, and Edgar Weippl. "From Hack to Elaborate Technique-A Survey on Binary Rewriting". In: *ACM Comput. Surv.* 52.3 (June 2019). ISSN: 0360-0300. DOI: 10.1145/3316415.

[225] Adam Wick and Amir Chaudhry. *CyberChaff: HaLVM unikernels protecting corporate networks*. http://unikernel.org/blog/2016/halvm-cyberchaff, accessed 14/10/2023. 2016.

[226] Dan Williams, Ricardo Koller, Martin Lucina, and Nikhil Prakash. "Unikernels As Processes". In: *Proceedings of the ACM Symposium on Cloud Computing*. SoCC '18. Carlsbad, CA, USA: ACM, 2018, pp. 199–211. ISBN: 978-1-4503-6011-1. DOI: 10.1145/3267809.3267845.

[227] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. "The CHERI capability model: revisiting RISC in an age of risk". In: *SIGARCH Comput. Archit. News* 42.3 (June 2014), pp. 457–468. ISSN: 0163-5964. DOI: 10.1145/2678373.2665740.

[228] Josiah Worcester. *Implement glibc *_chk interfaces for ABI compatibility*. https://musl.openwall.narkive.com/sdsJ7uUe/patch-implement-glibc-chk-interfaces-for-abi-compatibility, accessed 25/01/2021. 2015.

[229] Bruno Xavier, Tiago Ferreto, and Luis Jersak. "Time provisioning Evaluation of KVM, Docker and Unikernels in a Cloud Platform". In: *Proceedings of the 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. CCGRID 2016. IEEE. 2016.

[230] Nai Xia, Chen Tian, Yan Luo, Hang Liu, and Xiaoliang Wang. "UKSM: Swift Memory Deduplication via Hierarchical and Adaptive Memory Region Distilling". In: *16th USENIX Conference on File and Storage Technologies (FAST 18)*. Oakland, CA: USENIX Association, Feb. 2018, pp. 325–340. ISBN: 978-1-931971-42-3.

[231] Jidong Xiao, Zhang Xu, Hai Huang, and Haining Wang. "A covert channel construction in a virtualized environment". In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. CCS '12. Raleigh, North Carolina, USA: Association for Computing Machinery, 2012, pp. 1040–1042. ISBN: 978-1-4503-1651-4. DOI: 10.1145/2382196.2382318.

[232] Jidong Xiao, Zhang Xu, Hai Huang, and Haining Wang. "Security implications of memory deduplication in a virtualized environment". In: *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. Los Alamitos, CA, USA: IEEE Computer Society, 2013, pp. 1–12. DOI: 10.1109/DSN.2013.6575349.

[233] Fei Xu, Fangming Liu, Hai Jin, and Athanasios V. Vasilakos. "Managing Performance Overhead of Virtual Machines in Cloud Computing: A Survey, State of the Art, and Future Directions". In: *Proceedings of the IEEE* 102.1 (2014), pp. 11–31. DOI: 10.1109/JPROC.2013.2287711.

[234] Michał Zalewski. *American Fuzzy Lop (AFL) – Technical "whitepaper" for afl-fuzz*. https://lcamtuf.coredump.cx/afl/technical_details.txt, accessed 06/01/2026. 2013.

[235] Lukasz Zarnowiecki. *The Ultra Kernel Samepage Merging (UKSM)*. https://github.com/dolohow/uksm, accessed 19/05/2025. 2016.

[236] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2024. URL: https://www.fuzzingbook.org/ (visited on 2024-07-01).

[237] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. "The Demikernel Datapath OS Architecture for Microsecond-scale Datacenter Systems". In: *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. SOSP '21. Virtual Event, Germany: Association for Computing Machinery, 2021, pp. 195–211. ISBN: 978-1-4503-8709-5. DOI: 10.1145/3477132.3483569.

[238] Yiming Zhang, Jon Crowcroft, Dongsheng Li, Chengfei Zhang, Huiba Li, Yaozheng Wang, Kai Yu, Yongqiang Xiong, and Guihai Chen. "KylinX: A Dynamic Library Operating System for Simplified and Efficient Cloud Virtualization". In: *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC '18. Boston, MA, USA: USENIX Association, 2018, pp. 173–185. ISBN: 978-1-931971-44-7.

[239] Till Zimmermann and Emanuele Acri. *A stand-alone DHCP server*. https://github.com/crossbowerbt/dhcpserver, accessed 11/05/2022. 2014.

This page intentionally left blank.

Appendices

This page intentionally left blank.

A

APPLICATIONS PORTED AS UNIKERNELS

Unikernel & Sources	Description
DHCP server	A stand-alone DHCP server.
DNS server	An authoritative DNS server ported as unikernel. We modified it to stop just before the `accept()` function to simulate ephemeral unikernels.
FTP server	A FTP server ported as unikernel.
Hanoi resolver	A simple hanoi resolver that uses recursion.
Haproxy	The Haproxy server ported as unikernel (community edition).
Helloworld	A simple "Hello World!" unikernel used for testing and demonstration purposes.
Iperf3	The Iperf3 performance network tool server ported as unikernel.
Mandelbrot-perf	A Mandelbrot generator saving a 1280x720 image with 10000 iterations.
Nginx	The Nginx web server ported as unikernel. We modified it to stop just before the `accept()` function to simulate ephemeral unikernels.
Proxy server	A lightweight DNS-to-HTTPS proxy for the RFC 8484 DNS-over-HTTPS standard.
SQLite	The SQLite shell ported as unikernel.
SQLite-bench	SQLite benchmark.
Weborf	The Weborf web server ported as unikernel.

Table A.1: *Unikernels and Sources used in Chapter 3.*

Unikernel & Sources	Description
DHCP server	A stand-alone DHCP server.
DNS server	An authoritative DNS server ported as unikernel. We modified it to stop just before the `accept()` function to simulate ephemeral unikernels.
FTP server	A FTP server ported as unikernel.
Haproxy	The Haproxy server ported as unikernel (community edition).
Helloworld	A simple "Hello World!" unikernel used for testing and demonstration purposes.
Httpreply	A minimal HTTP server, useful for testing basic web server.
Iperf3	The Iperf3 performance network tool server ported as unikernel.
Lambda-perf	Reads an image, performs a 2x up-scaling transformation, and writes the result.
Mandelbrot-perf	A Mandelbrot generator saving a 1280x720 image with 10000 iterations.
Matrix-perf	A parallel 2000x2000 matrix multiplier saving the result to a file.
Nginx	The Nginx web server ported as unikernel. We modified it to stop just before the `accept()` function to simulate ephemeral unikernels.
NTP server	A NTP server ported as unikernel.
Proxy server	A lightweight DNS-to-HTTPS proxy for the RFC 8484 DNS-over-HTTPS standard.
Scamper-uradargun	Scamper driver to run radargun on a list of candidate aliases.
Scamper-uspeedtrap	Scamper driver to resolve aliases for a set of IPv6 interfaces.
Scamper-utnt	Scamper driver to reveal MPLS tunnels to a destination.
SQLite	The SQLite shell ported as unikernel.
SQLite-perf	The SQLite speed test ported as unikernel.
Weborf	The Weborf web server ported as unikernel.
Zlib-perf	A zlib-based unikernel that inflates and deflates a 10 MiB file.

Table A.2: *Unikernels and Sources used in Chapter 4.*

This page intentionally left blank.