


Memory, Address Spaces, Intra-Process-Isolation and Noodles

Doctoral Thesis

Author(s):

Hossle, Nora 

Publication date:

2025

Permanent link:

<https://doi.org/https://doi.org/10.3929/ethz-b-000736056>

Rights / license:

In Copyright - Non-Commercial Use Permitted

DISS. ETH NO. 30834

Memory, Address Spaces, Intra-Process-Isolation and Noodles

A thesis submitted to attain the degree of
DOCTOR OF SCIENCES
(Dr. sc. ETH Zurich)

presented by

NORA ELISABETH HOSSLE
MSc ETH in Computer Science, ETH Zurich
born on 04.08.1992

accepted on the recommendation of

Prof. Dr. Timothy Roscoe
Prof. Dr. Ana Klimovic
Prof. Dr. Ada Gavrilovska
Dr. Matthias Neugschwandtner

2025

This thesis could not have been written, and the work it is based on could not have been completed, without the help, support and suggestions of a great number of people.

First, I would like to thank my advisor Prof. Timothy “Mothy” Roscoe for taking a chance on me and giving me the opportunity to do systems research at ETH Zurich, as part of the Systems Group. Thank you for your continued support, feedback and patience during the last five years!

I would also like to thank the other members of my committee, Prof. Ana Klimovic, Prof. Ada Gavrilovska and especially Dr. Matthias Neugschwandtner who was my supervisor at Oracle Labs Zürich. I was fortunate to intern there and it truly was a great experience.

Furthermore, I’m enormously grateful that over the years I was able to collaborate with so many exceptional people (both at ETH Zurich and at Oracle Labs): Among them Daniel Schwyn, Matt Weingarten, Tom Kuchler, David Cock, Michael Giardino, Reto Achermann, William Blair, Abishek Ramdas, Adam Turowski, Roni Häcki, Lukas Humbel and Roman Meier (whom I want to thank as well for his assistance with benchmarking and IT in general).

A special thank you also to all the current and former members of ETHZ’s System Group, especially to my dear colleagues Anastasiia, Ben, Jasmin, Pengcheng, Zikai, Hidde, and Michal, and to Susie for all her invaluable encouragement. Many thanks also to the Systems Group’s amazing current and former admins Simonetta, Nadia and Macy, and particularly to Natasha for always having time and an open ear when I needed it.

In addition, I’m deeply indebted to my family for all their kind words and assistance over the past years. Thank you all so much.

Finally, I would like to thank my parents for their love and support not just during the last five years but during my whole life. Without you this simply would not have been possible.

Nora Hossle,
Zürich, February 2025

Chapter 2

Contents

2.1 List of contents

1	Acknowledgements	3
2	Contents	5
2.1	List of contents	5
2.2	List of figures	9
2.3	List of tables	11
3	Preface	13
3.1	Abstract	13
3.2	Translation of the abstract	15
4	Introduction	17
4.1	Motivation	19
4.2	Problem statement	22
4.3	Structure of this thesis	23
5	General Background	25
5.1	Inter- and intra-process-isolation	25
5.2	Memory protection keys	27
5.3	Lightweight multi-tenant serverless platforms	29

6	GraalVM and Mistletoe	31
6.1	Background	34
6.1.1	GraalVM [55]	34
6.1.2	How JVMs work internally	36
6.1.3	The <i>graal</i> compiler	37
6.1.4	Snippets	38
6.1.5	GraalVM Native Image	40
6.1.6	Isolates	44
6.2	Motivation	52
6.2.1	Why do systems research on GraalVM	52
6.2.2	Intra-process-isolation in GraalVM	52
6.2.3	Challenges for systems research on GraalVM	53
6.3	Mistletoe	55
6.3.1	Design	55
6.3.2	Applications	61
6.4	Implementation	65
6.4.1	Case study: Per-isolate billing	67
6.5	Evaluation	75
6.5.1	Micro benchmarks	75
6.5.2	DaCapo benchmark suite	81
6.6	Conclusion and future work	85
6.6.1	Future work	85
6.6.2	Conclusion	86
7	Noodles: Fat Threads	89
7.1	Motivation	92
7.2	Design and programming model	95
7.2.1	Targeted use case	95
7.2.2	Design requirements	96
7.2.3	Semantics of <i>Noodles</i>	98
7.2.4	Programming model	101
7.2.5	Managing page tables for <i>Noodles</i>	104
7.3	Linux kernel background	106
7.3.1	Focus: the memory management system	106
7.3.2	Main data structures	107
7.3.3	Internal workings of fork	114
7.3.4	Mapping types and their internal handling	120
7.3.5	No shared mappings	125

7.4	Implementation of Noodles	127
7.4.1	Prior work	127
7.4.2	A patch series	129
7.4.3	Porting the patch series	131
7.4.4	How the patch works and why it doesn't	134
7.4.5	Call graph of the patch series part 10	150
7.4.6	My patch	152
7.4.7	enter_noodle	166
7.4.8	libnoodles	169
7.4.9	Lessons learnt	171
7.5	Evaluation	173
7.5.1	Challenges inherent to benchmarking system calls	174
7.5.2	Benchmarking setup: Machine, benchmarks	175
7.5.3	Baselines	181
7.5.4	Analysis of results	184
7.5.5	Echo server use case	194
7.6	Conclusion and future work	200
7.6.1	What's next	200
8	Related Work	205
8.1	General intra-process-isolation	205
8.1.1	SpaceJMP [19]	205
8.1.2	Light-Weight contexts: An OS abstraction for safety and performance [45]	207
8.1.3	Enforcing Least Privilege Memory Views for Multithreaded Applications [12]	209
8.1.4	Endoprocess: Programmable and Extensible Subprocess Isolation [105]	210
8.2	Isolation using memory protection keys	212
8.2.1	libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK) [72]	212
8.2.2	Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries [24]	213
8.2.3	ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK) [94]	214
8.2.4	VDom: Fast and Unlimited Virtual Domains on Multiple Architecture [106]	214
8.2.5	Comparison to <i>Noodles</i>	216

8.3	Work related to <i>Mistletoe</i>	218
8.3.1	Controlled, systematic, and efficient code replacement for running java programs [49]	218
8.3.2	Debugging and tracing tools	219
9	Future Work	221
10	Conclusion	223
11	Glossary and Acronyms	227
12	References	233

2.2 List of figures

5.1	Intra-process-isolation overview	26
6.1	GraalVM architectural overview	35
6.2	Snippets lifecycle	39
6.3	Native Image build process	41
6.4	Virtual address space layout in the presence of <i>isolates</i>	48
6.5	Example code creating a GraalVM <i>isolate</i>	49
6.6	Example of a method running in an <i>isolate</i>	50
6.7	Conceptual design of <i>Mistletoe</i>	59
6.8	Structural design of <i>Mistletoe</i>	65
6.9	Function signatures from <i>libmistletoe.so</i>	69
6.10	Definition of hooks inside GraalVM	69
6.11	Foreign call implementations	70
6.12	Creating <code>ForeignCallDescriptors</code>	71
6.13	Adding our new <code>ForeignCallDescriptors</code>	71
6.14	Intrinsic nodes	72
6.15	GraalVM codebase, modified <code>setHeapBase</code> function	72
6.16	<i>Mistletoe</i> in action	73
6.17	<i>Mistletoe</i> micro benchmarks overhead overview	77
6.18	<i>Mistletoe SteadyStateSingleIsolate</i> micro benchmark	78
6.19	<i>Mistletoe RunInIsolate</i> micro benchmark	80
6.20	<i>Mistletoe RunInIsolate</i> micro benchmark with different hash table sizes	81
6.21	DaCapo benchmark results	83
7.1	<i>Noodles</i> conceptual overview	98
7.2	<i>Noodles</i> programming model visualization	102
7.3	The Linux kernel's concurrency model	108
7.4	Code snippet from Linux kernel version 5.15.116, <i>include/linux/sched.h</i> , line 77 [84]	110
7.5	The Linux kernel's concurrency model, including <code>mm_structs</code>	111
7.6	Code snippet from Linux kernel version 5.15.116, <i>include/linux/mm_types.h</i> , line 402 [83]	112

7.7	Code snippet from Linux kernel version 5.15.116, <i>include/linux/mm_types.h</i> , line 319 [87]	114
7.8	Internal Linux kernel page tables	117
7.9	Internal Linux kernel page tables after executing <code>fork(2)</code>	118
7.10	State of page tables after thread creation	119
7.11	No shared mappings	124
7.12	VAS API as proposed by the patch series [92]	133
7.13	First class VAS lifecycle	137
7.14	First class VAS attachment overview	140
7.15	Virtual address range collision	142
7.16	Virtual address space preparation before switching back	144
7.17	Conflict on VAS preparation before switch back	146
7.18	Diverging mappings	148
7.19	Callgraph of <i>vas.c</i> , patch series part 10 [92]	151
7.20	Visualization of a basic application's virtual address space	157
7.21	State of page tables after calling <code>enter_noodle()</code>	164
7.22	Call graph visualization of <i>vas.c</i> , my own patch series	165
7.23	Signature of <code>enter_noodle()</code>	166
7.24	Example of calling <code>enter_noodle()</code>	169
7.25	Second convenience wrapper for <code>enter_noodle()</code>	170
7.26	Definition of the <code>struct mapping</code> data type	170
7.27	Excerpt of <i>/proc/self/maps</i>	171
7.28	<i>Noodles</i> micro benchmarks on AArch64, 4KB, baselines only	182
7.29	<i>Noodles</i> micro benchmarks on AArch64, 4KB	184
7.30	<i>Noodles</i> , micro benchmarks on AArch64, 8KB and 16KB	187
7.31	<i>Noodles</i> , micro benchmarks on AArch64, mapping sizes comparisons	189
7.32	<i>Noodles</i> , micro benchmarks, <i>noodle npwr</i> comparison	190
7.33	<i>Noodles</i> micro benchmarks on AArch64, 32KB and 64KB	191
7.34	<i>Noodles</i> micro benchmark results for x86-64	193
7.35	Excerpt of the echo server's <code>per_connection</code> function	197
7.36	<i>Noodles</i> HTTPS server application benchmark results	198

2.3 List of tables

7.1	Overview of mapping types, part 1	121
7.2	Overview of mapping types, part 2	122
7.3	Case distinction for copying <i>private</i> mapping types	161

Chapter 3

Preface

3.1 Abstract

Traditionally, both operating systems and hardware only offered support for *inter-process-isolation*. However, today there exist a number of application types that are also in need of (lightweight) *intra-process-isolation*: Examples are web servers, applications written in a memory-safe language that use native libraries, applications using third-party libraries and applications handling sensitive data (e.g. cryptographic keys). In addition to these applications, new and upcoming lightweight multi-tenant serverless platforms, e.g. GraalOS [53] and Cloudflare Workers [9], also need intra-process-isolation mechanisms.

But even though hardware support for intra-process-isolation is now emerging (e.g. Intel’s MPK [27] and Arm’s PIE/POE [6]) it is not available on all machines yet and usually very limited in the number of protection domains supported (e.g. 16 for MPK and Arm’s PIE/POE). Processes on the other hand are often a too coarse grained abstraction to be an alternative, imposing high performance penalties on each protection domain switch. Threads are – though sufficiently lightweight – not meaningfully isolated.

An additional challenge is posed by the non-trivial complexity of real-life systems, making it hard to retrofit them with a new intra-process-isolation primitive (such as e.g. MPK). In this thesis I investigate this

in the context of Oracle’s GraalVM [55] and present *Mistletoe*, a lean C-runtime enabling a developer to tap into GraalVM’s execution flow with little knowledge of JVM internals. I also analyze GraalVM’s *isolates* [102] and their applicability to intra-process-isolation and present a prototype application built atop *Mistletoe* for per-*isolate* cycle accurate billing. Even though all JVMs (including GraalVM) deliberately abstract the hardware it is possible to directly access and experiment with new hardware features, e.g. make use of MPK, by using *Mistletoe*.

This thesis then presents *Noodles*, a novel intra-process-isolation mechanism that is primarily intended for lightweight multi-tenant serverless platforms. *Noodles* allows a thread to switch to its own virtual address space (while still remaining part of the same process context) – creating a so called fat thread. *Noodles*’s flexible API allows the caller to easily share relevant data with the newly created fat thread while removing access to all unrelated data – by not just disallowing access to the memory in question but by not having the physical pages mapped into the virtual address space altogether.

I implemented *Noodles* as a patch to the recent 5.15 Linux LTS kernel. Benchmark results show that creating a *noodle* is only 10%-11% as expensive as the only other option to create a new virtual address space – `fork(2)` – in relevant scenarios. To demonstrate *Noodles* applicability to real-world applications I present an example HTTPS server application using *Noodles*’s fine grained memory isolation capabilities to prevent the exploitation of a Heartbleed [23] inspired bug.

3.2 Translation of the abstract

Traditionellerweise unterstützen sowohl Betriebssysteme als auch Hardware nur *Inter-Prozess-Isolation*. Heutzutage existieren allerdings einige Applikationen, die zusätzlich auch (performante) *Intra-Prozess-Isolation* benötigen: Beispiele sind Webserver, Applikationen implementiert in einer Programmiersprache, die memory-safe ist, und native Libraries laden, Applikationen, die third-party Libraries brauchen und Applikationen, die sensitive Daten handhaben (beispielsweise kryptographische Schlüssel). Zu diesen Applikationen kommen neue und aufkommende sogenannte lightweight multi-tenant serverlose Plattformen, zum Beispiel GraalOS [53] und Cloudflare Workers [9], welche ebenfalls Intra-Prozess-Isolation benötigen.

Obwohl Hardwareunterstützung für Intra-Prozess-Isolation im Aufkommen ist (bsp. Intels MPK [27] und Arms PIE/POE [6]), steht sie uns noch nicht auf allen aktuellen Maschinen zur Verfügung. Ausserdem ist die Anzahl der unterstützten Schutzdomänen normalerweise sehr beschränkt (bsp. 16 für MPK und PIE/POE). Prozesse sind hingegen häufig eine viel zu grobe Abstraktion, um in Frage zu kommen, da sie per Wechsel zu viel extra Aufwand generieren. Threads sind dafür - obwohl durchaus eine schlankere und dadurch performantere Abstraktion - nicht sinnvoll isoliert.

Ein zusätzliches Problem ist die alles andere als triviale Komplexität von echten Systemen, was es um einiges schwieriger macht, sie im Nachhinein mit Intra-Prozess-Isolation nachzurüsten. In dieser Doktorarbeit untersuche ich dies im Kontext von Oracles GraaVM [55] und präsentiere *Mistletoe*, eine schlanke C-Runtime, die es einem Entwickler erlaubt sich in die Ausführung von GraalVM selbst einzuhaken, ohne zuvor viel über das Innenleben von JVMs wissen zu müssen. Ich analysiere auch GraalVMs *Isolates* [102] und ihre Verwendbarkeit für Intra-Prozess-Isolation und präsentiere eine auf *Mistletoe* aufgebaute Prototypapplikation zum Zweck der Zählung von Zyklen per Isolate. Obwohl alle JVMs (auch GraalVM) absichtlich die Hardware, auf der sie ausgeführt werden, abstrahieren, ist es möglich mit Hilfe von *Mistletoe* direkt auf die Hardware zuzugreifen und mit neuen Features, zum Beispiel MPK, zu experimentieren.

Anschliessend präsentiert diese Doktorarbeit *Noodles*, ein neuartiger Intra-Prozess-Isolationsmechanismus, der primär für schlanke, multi-tenant

Serverlosplattformen gedacht ist. *Noodles* erlaubt es einem Thread, in seinen eigenen virtuellen Adressraum zu wechseln, während er gleichzeitig im gleichen Prozesskontext verbleibt, sodass ein sogenannter fatter Thread entsteht. *Noodles'* flexible API erlaubt es dem Aufrufer auf einfache Art und Weise Daten mit dem neuerlich kreierten fetten Thread zu teilen, während er gleichzeitig den Zugang zu allen irrelevanten Daten entfernt, indem er nicht nur die Zugangsberechtigung auf nicht berechtigt setzt, sondern die Verlinkung zu den physischen Seiten aus dem virtuellen Adressraum entfernt.

Ich habe *Noodles* als einen Patch für den aktuellen 5.15 Linux LTS kernel implementiert. Benchmarkresultate zeigen, dass eine *Noodle* zu erstellen in relevanten Szenarien nur 10%-11% so viel kostet wie die einzige andere Möglichkeit einen neuen virtuellen Adressraum zu kreieren – `fork(2)`. Um *Noodles'* Anwendbarkeit auf reale Systeme zu demonstrieren, präsentierte ich einen Beispiel-HTTPS-Server, der *Noodles'* flexible Speicherisolationfähigkeiten nutzt, um die Ausnutzung einer von Heartbleed [23] inspirierten Sicherheitslücke zu verhindern.

Chapter 4

Introduction

The central topic of this thesis is *intra-process-isolation*, referring to isolating different components which are running in the *same* process from each other. Intra-process-isolation is complementary to *inter-process-isolation*, referring to isolating multiple different processes from each other.

Today, there exist many different usage scenarios in which we not only benefit from but unequivocally need not just inter-process-isolation but also intra-process-isolation: Web servers might simultaneously connect to and communicate with multiple mutually distrusting clients or applications written in a memory safe language might load a native library. Alternatively, a third-party logging library included by our application might be discovered to exhibit a bug from which we need to insulate the rest of our application.

An especially important use case of intra-process-isolation is to control access to particularly sensitive data (e.g. cryptographic keys in a server). If we prevent other, unrelated components of our application from accessing this data we can aim to at least reduce the blast radius of security critical bugs (e.g. Heartbleed [23]). This is in line with the principle of least privilege which demands that no parts of an application should have access to any data other than that which is strictly needed for their correct operation.

Intra-process-isolation also enables novel types of applications: Serverless computing that promises pay-as-you-go together with automatic

scaling is becoming ever more popular. However, to co-locate multiple tenants on the same machine, containers or even whole virtual machines have to be created for security reasons, imposing a significant overhead both in terms of runtime and memory usage. For characteristically small, short running functions this much overhead is unacceptable because it makes serverless platforms expensive to operate, ultimately increasing the price for customers.

Strong but also lightweight isolation primitives promise to make serverless platforms more economically feasible [1]. This allows to build so called *lightweight multi-tenant serverless platforms* [9, 53], a new type of serverless platforms that can run several different workloads in the same process.

While there are more aspects to intra-process-isolation (i.e. how to securely manage other resources) this thesis focuses on the memory aspect: How to partition the virtual address space into different protection domains and thus control access to the physical data pages mapped into it.

Related to developing novel intra-process-isolation primitives is the question how production-grade, often sprawling applications can possibly benefit from such new mechanisms as in real-world scenarios we almost never start with a clean slate. Retro-fitting an existing application so that it can benefit from new isolation primitives (e.g. Intel’s MPK [27] and Arm’s PIE/POE [6]) is often very challenging. This is made worse by the fact that developers working on new isolation primitives might not have application domain specific knowledge and vice versa.

The first part of this thesis investigates the question of how real-life systems can be retro-fitted to benefit from new isolation primitives by studying the example of Oracle’s GraalVM [55], a JDK. I explore how new functionality can be added to a state-of-the-art JVM in an adhoc manner. In addition to this, the first part of this thesis takes a detailed look at *GraalVM’s isolates* [102], an isolation primitive partitioning the heap of a JVM.

The second and main part of this thesis focuses on designing a new isolation primitive to bring intra-process-isolation to the Linux kernel: I propose a novel operating system level abstraction called *Noodles*. *Noodles* are so called *fat threads* – in terms of granularity they sit between

traditional threads and processes as I identified a gap in the design space there.

The second part of this thesis also takes an in-depth look at the already familiar operating systems level abstractions *processes* and *threads* and investigates how they impact intra-process-isolation. As I revisit these familiar abstractions I explore the design space between them by analyzing the Linux kernel implementation of the system calls `fork(2)` [38], `vfork(2)` [40] and `clone(2)` [37]. This then serves as a motivation for my new approach called *Noodles*: I propose a novel intra-process-isolation primitive called a *noodle* which sits in-between processes and threads in terms of granularity and isolation.

4.1 Motivation

Historically, the focus used to be on inter-process-isolation, not intra-process-isolation. That the former is a requirement for any reasonably secure and stable system is plain to see: To give a concrete, straightforward example, think of this classic scenario: Your email client which has just received a suspicious email is running in one process while the pdf viewer displaying sensitive medical records in another window is running in another process. Naturally, the two processes should be isolated from each other.

Until quite recently, the traditional process abstraction as provided in general by all Unix derived operating systems and in particular by the Linux kernel [88] was thought sufficient for whenever any form of isolation was needed. In all use cases where in addition to multiple execution contexts (for e.g. parallelism or to better multiplex certain resources) also privilege separation and different access policies for certain resources were deemed necessary, best practice was considered to simply separate the components in question into two different processes.

Complimentary to the process abstraction is the thread abstraction: If no isolation was needed between two execution contexts and they belonged to the same program, the more lightweight thread abstraction could be used instead of the more heavy-weight process abstraction to execute the two instructions streams in parallel.

Hardening threads in any way was thought unnecessary since if it ran in your own virtual address space, it was obviously code trusted

by you, the developer, in the first place (and probably even written by you anyway). In the (allegedly) unusual case it wasn't, as stated above, you were advised to alternatively just isolate it in a full-blown, separate process instead and use IPC for communication, potentially paying a performance penalty. To summarize, code running in our virtual address space was thought to imply that it was trusted, due to the process context being viewed as a single protection domain.

Unfortunately, this seemingly straightforward argument no longer always applies. Indeed, it may have been based on an oversimplified view of program development almost right from the start: There is virtually no non-trivial program that doesn't load libraries and indeed the use of third-party libraries goes back a long time (the EDSAC developed by Maurice Wilkes and his team is generally regarded as to have been the first computer putting the idea of program libraries into practice in 1949 [46]).

That aside, even code written by yourself (or at least in house) may of course be just as riddled with bugs and vulnerabilities as third-party code is. The assumption that just because something runs in your virtual address space it is automatically trusted code is therefore to be rejected.

An alternative line of argument is that it is simply not feasible to implement intra-process-isolation and that we are therefore *forced* to trust any code running in our process context. This is again to be rejected as not only the work presented in this thesis shows but also an extensive body of prior work on the topic of intra-process-isolation [19, 45, 12, 105, 72, 24, 94, 106] proves its feasibility.

In any case, as applications have ballooned to ever bigger code bases and have started to incorporate ever more functionality, it has become more and more untenable to keep everything in the same, suddenly huge protection domain. To mention a familiar type of user application, in most web servers it is undesirable to parse the input of a potentially hostile client in the same protection domain as, say, the server's cryptographic keys reside. If we have only a single protection domain, any security critical bug in any component – from the logging library to the HTML parser – has potentially catastrophic consequences: A well-known real-life example of this is of course the Heartbleed vulnerability [23].

As stated, the principle of least privilege mandates that no component should have access to more data than necessary for its correct

operation. A thread processing an event related to client A also having access to the session keys for client B at the same time violates this principle at its core. Unfortunately this is exactly how widely used web servers such as Apache [3], nginx [47] etc. function under the hood: Because process creation comes with a heavy performance penalty attached not even Apache running in non-event based mode typically creates a fresh process for every single new connection. Instead, several processes are created which each contain a pool of worker threads which are then continuously reused.

As a consequence of this practice the blast radius of serious vulnerabilities such as Heartbleed [23] can be huge. Even worse such bugs are often inevitable in any non-trivial application – which are often still written in non-memory-safe languages such as C. The question isn't if there will be another one but rather when.

Why isn't intra-process-isolation more widely applied if there is such a strong case for it? The simple answer to this question is that our existing solutions are not yet good enough. They either come with a high performance penalty attached, are too cumbersome to use in practice, require not always available hardware support or don't offer strong enough isolation.

Also, there is no suitable abstraction for intra-process-isolation offered by today's general purpose operating systems: As mentioned, processes are often either too coarse grained and too heavy weight, incurring too much of a performance overhead (e.g. high performance web servers). Or the application's code simply can't be refactored to split up the application into multiple processes within a reasonable amount of time (e.g. JVMs). Threads on the other hand are more lightweight but don't offer any meaningful form of isolation.

Since the familiar OS-level abstraction of the process and the thread have proven to rigid to provide enough flexibility in case of huge applications such as Apache or JVMs, I believe it is time to explore the design space *between* processes and threads.

Having an abstraction with close to thread granularity that still provides meaningful isolation would be especially beneficial for lightweight multi-tenant serverless platforms. These commonly have a trusted hypervisor component but need to be able to run untrusted user-provided code in process context to keep the overhead as minimal as possible.

As stated, virtual machines and containers are simply too heavy weight, especially as one of the main cost factors in serverless computing is how much DRAM is occupied at a point in time. For economical reasons, lightweight multi-tenant serverless platforms need to be able to scale in such a way that as many workloads as possible can be run on the same hardware at the same time.

4.2 Problem statement

To summarize, the familiar operating systems level abstractions processes and threads are either not fine grained or isolated enough to meaningfully partition today's sprawling applications into different protection domains. Today's enterprise strength applications have grown exponentially in complexity and partitioning them into old fashioned processes is neither doable nor desirable due to the heavy performance penalty associated with creating new processes and IPC.

Still, some form of intra-process-isolation is clearly needed to partition these applications into mutually distrusting protection domains to mitigate risk and control damage in case of a security relevant bug. The root of the problem is the monolithic virtual address space giving all execution contexts, namely, threads running in the same process, not only the same view on memory but also the same access privileges on all data. This implies that a new isolation primitive combining light weight execution contexts with isolation is needed.

I propose that there is a gap in the design space between threads and processes that needs to be filled: What's needed is an abstraction that provides some form of isolation but at the same time remains part of the same process and enables easy, performant sharing with all the others of its kind that are part of the same process. Threads do not fit the bill as they provide no meaningful form of isolation.

Note that while hardware support for inter-process-isolation has been available for a long time (i.e. the MMU) this is not true for intra-process-isolation. Though there is now emerging support in the form of Intel's MPK [27] and Arm's PIE/POE [6], older, still in use hardware does not support it. More unfortunately, due to limited bits available in the Page Table Entries the number of different protection domains is very limited. Clearly, this is not a solution that scales.

Simply changing the permissions in the Page Table Entries via a call to `mprotect(2)` is also not an option because these changes would not be thread local but instead always apply to everything running in the same virtual address space. In addition, having to write page tables each time a protection domain switch occurs is prohibitively expensive [24]. Instead we need a solution that scales and requires no special hardware.

Also, there is the question of how to bring new isolation primitives and intra-process-isolation solutions to big, pre-existing systems. Due to their complexity, it is often challenging to retrofit new solutions to real-life, state-of-the-art applications.

4.3 Structure of this thesis

This thesis is split into two parts: The first part describes the challenges encountered when attempting to bring intra-process-isolation to a real-world system, Oracle’s GraalVM [55], a JDK containing among other things a state-of-the-art JVM. The first part serves as a motivating example for the second part which investigates in depth how to design and then implement a new isolation primitive for intra-process-isolation (with the presented prototype targeting the Linux kernel).

After setting up some shared background for both parts in **Chapter 5**, **Chapter 6** presents the conceptual design of *Mistletoe*, a lean C-runtime enabling the systems programmer to “tap” into GraalVM [55] at specific execution points and potentially set up protection domains for intra-process-isolation. The focus is on investigating how a novel operating system abstraction or hardware extension can be taken advantage of in a production grade system.

The second part contained in **Chapter 7** deals with designing a new isolation primitive and adding it to the Linux kernel in the form of a novel system call.

The red thread that ties both parts together is the need for intra-process-isolation and the challenges that arise when dealing with a real world system. Possible combinations of my work in both parts are to address this are then explored in **Chapter 9**. Finally I conclude in **Chapter 10**.

Chapter 5

General Background

This chapter covers general background information of use for both the first part of this thesis (see chapter 6) and the second part (see chapter 7). Background information specific to part 1 can be found in the upcoming section 6.1, for part 2 see section 7.3.

5.1 Inter- and intra-process-isolation

It has long since been agreed that *inter-process-isolation* is vital for the integrity of any general-purpose operating system. Without it, there would be nothing to stop a malicious application (even if it hasn't managed to escalate its privileges) from siphoning off the data of any other application that just happens to run on the system at the same time. In fact, even if there were no ill intent, programs developed without any knowledge of each other cannot be trusted to fairly share resources between themselves without, e.g., accidentally overwriting each others state.

As a consequence of this, there are many mechanisms used by operating systems to enforce inter-process-isolation. Taking Linux as a concrete example, as is well known, it isolates its processes by having them run in separate virtual address spaces (supported by the MMU).

However, there are many more mechanisms that also serve (at least in part) to isolate processes from each other: For instance cgroups [73, 25]

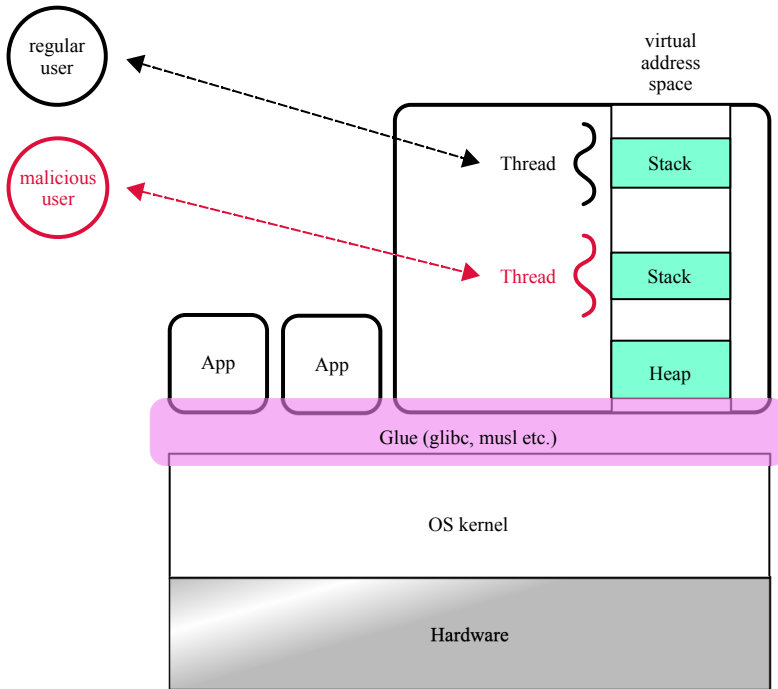


Figure 5.1: Visualization of why we need intra-process-isolation.

allow to isolate whole groups of processes from other groups with regard to different resources, e.g. memory bandwidth, also making this a form of inter-process-isolation. Further examples are permissions and Linux’s capabilities [36]. Files restrict certain resources to specific groups of processes, e.g. they prevent a process from reading data from a file that another process has written. Capabilities restrict certain operations and behaviors which otherwise might allow a process to subvert inter-procession-isolation (e.g. `CAP_SETUID` allows the forging of a process’s UID) to specially privileged processes.

For intra-process-isolation on the other hand there is much less sup-

port available. Recently new hardware support has become available in the form of Intel’s MPK [27] and Arm’s POE/PIE [6]. However, as this is a recent development, memory protection keys are not yet supported on all machines and even if they are available the number of supported protection domains is quite limited (see the next section, section 5.2 for more details).

Please see figure 5.1 for a visualization inspired by the traditional abstraction layer diagrams shown in operating systems textbooks. Note that while the different processes or applications running on top of the operating system are indeed separated from each other (and from the operating system managing them), the same cannot be said about the two different threads running in the same process context in the top right corner. Though each thread has its own stack, the heap is shared – as is the the rest of the virtual address space. Both threads have the same view on the virtual address space and furthermore the *same permissions* for all data mapped into it.

Imagine now that our application is a web server, simultaneously connecting to multiple mutually distrusting clients. If a regular user has sent us data it is of course paramount that our application keeps this data confidential – and does not inadvertently leak it to the malicious user also connected to our web server at the same point in time.

This is not a hypothetical scenario: For performance reasons popular web server’s such as Apache [3] and nginx [47] commonly don’t fork a new process for every single connection. For example, if Apache’s Multi-Processing Module (MPM) *worker* is used [79], each process will operate with pool of threads to serve all clients as quickly as possible.

Web servers are just one of many potential usage scenarios of intra-process-isolation. Other scenarios, aside from those already mentioned in the previous chapter, chapter 4, include (but are not limited to) a shared storage server running user-provided analytics workloads close to the date, a multi-tenant database or a web browser loading third-party resources for a particular website.

5.2 Memory protection keys

As already mentioned earlier, both Intel’s x86-64 and Arm’s AArch64 have started to offer hardware assistance for intra-process-isolation – at

least for the memory side of it. Intel offers Memory Protection Keys [27] (MPK) [27] since Skylake (6th Generation) [100], also referred to as pkeys by the Linux kernel [82]. The idea is that each of these “keys” can be mapped to a specific set of permissions (read-only, read-write, write-only, inaccessible). Each physical page is then associated with one of these keys by writing the number (or ID) of the key directly into the Page Table Entry mapping the physical page into the virtual address space. On a data access (MPK has no effect on instruction fetches) the intersection of the permissions as written in the Page Table Entry and as are mapped to the key stored in the Page Table Entry determines if the access is legal.

What’s most notable is that the register that determines which permissions are mapped to which key (called PKRU) can be read and written to without special privileges, from user space. That means that the permissions for a specific key (or a protection domain depending on usage) can be changed with extremely little overhead (just a couple of cycles). On the flip side this also limits the usefulness of MPK because on its own it cannot be used to enforce intra-process-isolation, it always has to be paired with something else to ensure that no illicit `wrpkru` instructions are issued to write the register. Another limitation of MPK is that because only 4 bits are available to store the key in the Page Table Entry (pte), only ever 16 different keys can exist (in practice there are even less than that available because key 0 is the default context and it is possible that a library loaded by an application also allocates some of the keys for itself). Finally, though MPK appeared first with Skylake, it only shipped with server-class CPUs.

Arm on the other hand introduced Permission Indirection Extension (PIE) and Permission Overlay Extension (POE) in Armv8.9-A/Armv9.4-A [6, 96]. PIE and POE function similarly to Intel’s MPK: Instead of a key, an index is stored in the Translation Table Descriptor (TTD) (the index is stored in the `PIIndex` field of the TTD). This index is then used on memory access to index into a register (`PIR_ELx` with `x` equal to 1, 2 or 3) to determine the permissions for a particular physical page. As a twist, the value found in that register also determines if a *permission overlay* should be applied to further restrict the permissions just read. If this is the case, another field in the TTE (called `POIndex`) is used to index into another register (`POR_ELx` with `x` equal to 0, 1, 2 or 3 depending

on the exception level) to find the mask to further restrict the resulting permissions. Notable is that writing the first of the registers indexed into, (PIR_ELx), is a privileged operation while setting the permissions in POR_ELx is not. The size limitations are the same as with MPK, there can only be 16 different indexes and 16 different masks at the same time.

5.3 Lightweight multi-tenant serverless platforms

To explain serverless in a nutshell, it's key idea is that instead of the user having to set up and maintain a server somewhere themselves (e.g. create a virtual machine on Microsoft's Azure Cloud), the user only supplies the function they wish to run to a so called serverless platform that then does all the heavy lifting for them. Once supplied, the user-provided function is then installed by the serverless platform in as many machines (or servers, the name is misleading as only the user no longer has to manage a server) as necessary to process all incoming requests in a timely fashion – transparently scaling up or down by allocating or de-allocating more instances running the user-provided function.

Serverless has become increasingly popular over the last few years. It's chief selling points are its fee model of pay-as-you-go – you only pay for the amount of time your workload actually runs and not for the time e.g. a virtual machine is running or even worse is just allocated somewhere – and the promise of effortless scaling without the user needing to trouble themselves with server provisioning.

While that sounds very attractive, serverless comes with some downsides: For one, it's of course not straightforward for the serverless platform to know when an input for a specific installed function will arrive. That creates a dilemma since if the platform deallocates all instances the response time will be high (this is referred to as a cold start). If however the platform keeps a few instances of the installed functions around it might be that no input will ever arrive and thus though the instances of occupied precious DRAM, the platform operator doesn't get paid.

Another problem is that it's of course not save to run potentially thousands of functions on the same bare metal machine without any form of isolation between them. The most obvious solution to this would be

to wrap the function's runtime in some kind of container or even virtual machine but that creates a very high overhead, increasing cold start times. Even worse, many serverless workloads consist of functions that run only a very short amount of time.

This is where *lightweight* serverless platforms like *Cloudflare's Workers* [9] become attractive: Instead of isolating each workload in a container or virtual machine they try to use something much more lightweight while still providing effective isolation. Cloudflare does this by using the Javascript Engine V8 [91]'s so called *isolates* to isolate each workload which essentially amounts to running a number of workloads inside the same process (see also chapter 8 for more details).

Chapter 6

GraalVM and Mistletoe

This chapter covers my work on intra-process-isolation in the context of Oracle’s GraalVM [55], essentially a JDK plus¹. I present *Mistletoe*, a lean C-runtime able to tap into GraalVM itself, that I developed while exploring GraalVM’s *isolates* [102] and their applicability to intra-process-isolation. I also discuss several different use cases for *Mistletoe* and present and evaluate a prototype application using it for per *isolate* cycle accurate billing.

Java Virtual Machine (JVM)s deliberately abstract the underlying hardware they are running on. This is one of the Java platform’s greatest strengths as it makes Java code easily portable. However, in the special context of systems programming it can also be a weakness: Java is considered (from the systems programmer’s perspective) to be a *high level language*: e.g. it famously has a garbage collector and the JVM manages many resources like for instance the so called *platform threads* behind the scenes, meaning transparently to the application programmer. This makes it more difficult to employ new hardware features, e.g. utilize memory protection keys such as Intel’s MPK [27] and Arm’s POE/PIE [6]. It also makes it harder to use accelerators meaningfully, debug performance problems caused e.g. by bad cache locality or relate hardware counters correctly to application code without unexpected ac-

¹Note that the work presented in this chapter was performed with GraalVM’s version corresponding to commit `de8da6cee7c0d6efe1f44970518cf02ad7f933ad` of GraalVM’s public GitHub repository [54].

tivity of the JVM distorting measurements.

The just mentioned challenges are all common systems programming tasks. To sidestep any unwelcome interference caused by the JVM it is tempting to instead implement it directly *inside* the JVM itself, in our case GraalVM.

However, this is no easy task as GraalVM is a complex system which it is non-trivial to navigate and which most systems programmers won't have in-depth prior knowledge of. The problem is made even more acute by the complex meta compilation system employed by GraalVM, centered around *snippets* [78]. In addition to this, when operating directly from within a JVM such as GraalVM, a programmer may be restricted by the fact that the environment is not yet completely initialized when their code is called, e.g. any function resulting in a heap allocation might be impossible to call at certain points.

It is still very tempting to implement functionality directly within the GraalVM itself though: It is a state-of-the-art, production grade JVM and implementing any functionality directly *inside* it allows *all* Java programs running on it to benefit from e.g. an improvement in performance when leveraging a hardware accelerator.

As a solution to this problem I propose that we need to break out of the box: I present *Mistletoe* – a lean C runtime that can be called directly and from any point in GraalVM's code (something not possible by simply using JNI). *Mistletoe* provides a clean slate allowing me to implement a prototype of a cycle based billing application.

This chapter is structured as follows: Immediately following this introduction, in section 6.1, I present the reader with a minimum of background information concerning GraalVM itself, *isolates*, JVMs and a special feature of GraalVM called Native Image. In section 6.2 I further elaborate on the motivation of this work. Section 6.3 covers the design of *Mistletoe* including motivating design goals and requirements. It also covers possible applications implemented on top of *Mistletoe*. Section 6.4 complements the previous section on *Mistletoe*'s design with a detailed description of its implementation and the challenges I encountered on the way. It also describes the prototype of a billing application I've implemented on top of *Mistletoe* with the aim of being able to attribute cycles per *isolate*. The section after that, section 6.5, is dedicated to the evaluation of *Mistletoe* and the billing application. Finally, I conclude

this chapter in section 6.6 and present possible future work.

6.1 Background

The aim of this section is to provide the reader with a minimum of necessary background for the later sections in this chapter. The topics covered here are GraalVM [55] itself, some basic information concerning the inner workings of a JVM with a JIT compiler and several special features of GraalVM (*snippets*, *isolates* and *native image*). Note however that covering all aspects of GraalVM in full is well out of scope for this thesis.

6.1.1 GraalVM [55]

The GraalVM project can be seen as essentially a JDK plus. This should be taken to mean that while GraalVM (the actual JVM contained in the project) can be used as a drop-in replacement for the reference JVM implementation, the Java Hotspot VM [69], [64], it also offers users a series of other features. These go well beyond what JDKs traditionally offer:

- With help of GraalVM’s Truffle Language Implementation Framework [70] an interpreter for a language other than Java (or a language compiled to bytecode) can be built. This implementation of an interpreter can then itself be run on top of GraalVM, allowing to run the new language itself on top of GraalVM.
- GraalVM’s Native Image tool, which is discussed in more detail in subsection 6.1.5, allows to compile Java applications Ahead of Time (AOT) to standalone binaries that can then be executed on the chosen platform natively, without the need for having a JVM installed.
- GraalVM’s Polyglot API [68] allows to embed code written in other languages (called guest languages) in Java applications. The API also allows programmers to build applications where values can be passed between parts written in different languages.
- The GraalVM project includes a runtime which is ECMAScript compliant and can execute JavaScript and Node.js applications [58].

- The GraalVM project also includes a LLVM Runtime to execute LLVM based languages such as C/C++ and Fortran [59].

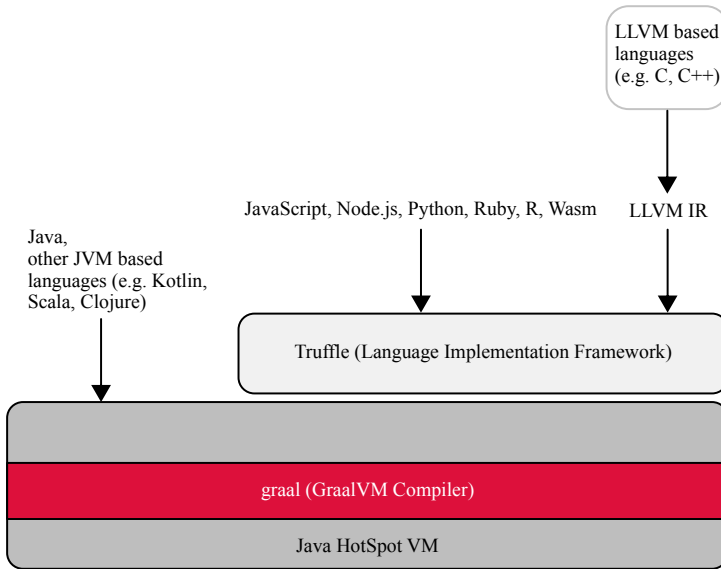


Figure 6.1: GraalVM architectural overview, adapted from figure [57]

GraalVM itself is written (practically completely) in Java, making it a *meta-circular* JVM. As a consequence of this design choice it defines the semantics of high-level Java operations by implementing them as methods written in lower-level Java. [78].

GraalVM is based on the Java Hotspot VM though it has its own JIT compiler, called *graal*, instead of Hotspot’s older client and server compilers, also called *C1* and *C2*. Note that the term compiler here does not refer to the compiler compiling Java files to bytecode, which is called *javac*, but to the compiler used *internally* by the JVM itself to JIT compile bytecode at runtime to machine code instead of interpreting it. (See subsection 6.1.2 for more details on this.)

As its build system GraalVM uses a tool called *mx* [60]. Figure 6.1 shows a conceptual overview of GraalVM.

6.1.2 How JVMs work internally

To understand some of the design considerations inspiring GraalVM Native Image, discussed in subsection 6.1.5, it is important to have a general idea of how most JVMs work internally. In this subsection I therefore briefly discuss the important design concepts of interpretation, JIT and *deoptimization*. I focus on the approach taken by the Java Hotspot VM [69] and GraalVM for obvious reasons but many of the concepts are employed by other JVMs as well.

In a nutshell, a JVM aims to combine the benefits of a bytecode interpreter (execution can start immediately) with the benefits of an optimizing compiler (better performance). At the same time the aim is to mitigate the downsides of both (interpretation is slow compared to the performance of heavily optimized code but producing that heavily optimized machine code takes time). To get the best of both worlds so to speak, bytecode is initially interpreted by the interpreter part of the JVM, guaranteeing that execution can start immediately. As code is executed, the JVM profiles which part of the application code are “hot” (in other words executed often). These parts of the code are then given to the compiler that produces a heavily optimized version of the corresponding machine code.

There is of course a lot of room for fine tuning the basic design concept of combining an interpreter with a JIT compiler: For instance the Java Hotspot VM has two different compilers, the client compiler and the server compiler, also called *C1* and *C2*, to balance compilation time with the level of code optimization: If code is executed repeatedly, it is at first given to the client compiler which takes less time to run than the server compiler but also doesn’t produce as heavily optimized machine code. Only if code continues to remain hot, will it be passed to the server compiler to produce aggressively optimized machine code.

This tiered approach is of course only one of many potentially successful ones, for comparison GraalVM (which as mentioned is also based on the Java Hotspot VM) only has a single compiler, called *graal*. Other degrees of freedom when implementing the design concept described above are the conditions code needs to fulfill to be passed to a compiler and

how aggressively it is optimized.

This brings us to another important design concept employed by JVMs which is called *deoptimization*: Often code can be made much more performant if certain simplifying assumptions are made, e.g. assuming that the receiver of a method call is always of the same runtime type might allow to inline the method, doing away with the function call overhead altogether. But what happens if during execution it suddenly turns out that this assumption is violated and the method call receiver is not in fact always of the same runtime type? The dynamic nature of the Java language means that it's generally not possible to prove that such an assumption always holds, after all (e.g. because of reflection).

This is where *deoptimization* comes in. *Deoptimization* allows an optimizing compiler to aggressively optimize machine code by making certain simplifying assumptions that might be violated at any point. If they are, code is simply *deoptimized*, in other words, execution of it is passed back to the interpreter.

6.1.3 The *graal* compiler

The *graal* compiler which is shown in figure 6.1 interfaces with the Java Hotspot VM [69] using the JVM Compiler Interface (JVMCI). This is a low-level, privileged interface the JVM itself (in this case the Java Hotspot VM) offers to read its metadata, such as the bytecode corresponding to a method. In addition, this interface allows to install machine code directly into the JVM, which is what the *graal* compiler does when it installs the results of its compilation into the JVM's code cache [52]. To summarize JVMCI allows to “plug” a compiler such as *graal* into any JVM that supports JVMCI. This is in fact what allows the GraalVM project to use the *graal* compiler instead of the Java Hotspot VM's traditional tiered compilation scheme employing its client and server compilers. Also, Graal API contains interfaces that provide information about types, methods, fields – these interfaces are implemented by the VM [78].

The *graal* compiler is itself written in Java and uses a graph based intermediate representation internally [18]. The intermediate graph representation can be understood as a superposition of a control flow graph and data flow graph.

The compiler's inputs are the bytecode that is to be compiled and

metadata (e.g. the constant pool) [78]. Any bytecode is first compiled to the intermediate graph representation which was developed with ease of code optimization and transformation in mind. A crucial step is to *lower* the intermediate graph representation of a program: This refers to replacing the nodes corresponding to high level Java concepts such as `instanceof` or `new` with a sub-graph describing their semantics. For this the VM provides so called *snippets* to the compiler, see also the next subsection, subsection 6.1.4. Finally machine code is emitted and loaded into the JVM's code cache. The compiler also outputs reference maps for the garbage collector and metadata for deoptimization (recall subsection 6.1.2) – all of this is passed on to the VM.

6.1.4 Snippets

One big source of complexity in the GraalVM project and a design concept central to the *graal* compiler (see subsection 6.1.3 above) is called *snippets*: The idea behind *snippets*, described in a paper from 2015 [78], is to use the Java language *itself* to describe the semantics of certain higher level language features such as `instanceof`, `checkcast` or `new`. The aim is to further decouple the JVM from the used JIT compiler (*graal*): While *snippets* are both compiler *and* JVM specific, referencing internal implementation details of both, with exception of the *snippets* provided by the JVM to the compiler, the code of the compiler does not need to reference JVM internals. This allows to use *graal* more easily with other JVMs, such as e.g. the Maxime VM [78].

At first glance, *snippets* may look like ordinary static methods, distinguished only by being annotated with `@Snippet`. However, its crucial to understand that they are not, in fact, methods or even functions: To start with they are never actually *called* at all. Instead, they are *compiled* ahead of time to their corresponding intermediate graph representation (as used by the *graal* compiler). Following this, whenever the compiler encounters bytecode making use of a high level concept such as `instanceof`, the received bytecode is first compiled to its corresponding intermediate graph representation as well. Then the *snippet* graph is *inlined* in the graph representing the received bytecode, replacing there the high level node corresponding to `instanceof`.

See figure 6.2 for a visualization of how *snippets* are processed by the *graal* compiler: We start out with bytecode representing the *snippet* at

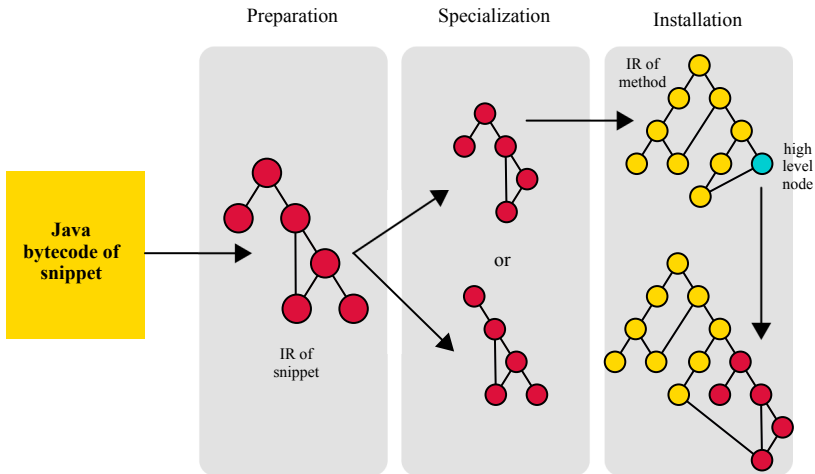


Figure 6.2: Snippets lifecycle, adapted from figure 9 in the *Snippets: Taking the High Road to a Low Level* paper [78]

the left of the figure. This bytecode is then compiled to the intermediate graph representation we encountered already in subsection 6.1.3, just above. This is referred to as *preparation* of the *snippet* and is only done once for each *snippet*.

In a next step, when the compiler receives actual bytecode referencing a specific *snippet*, the resulting intermediate graph representation of the *snippet* from the *preparation* phase is *specialized*: This is done by replacing all constant (known at compile time) parameters of a *snippet* with their actual values. An example for this would be a *snippet* taking profiling information as input telling it if a certain object reference had ever been observed to be `null` before. This information is collected by the interpreter and then passed to the compiler together with bytecode to be compiled, in other words it is already known at compile time.

Due to compiler optimizations such as constant folding and dead code elimination, *specialization* often allows a *snippet*'s original graph

representation to be greatly simplified. This step of *snippet* processing is shown in the middle of figure 6.2, under the header *Specialization*. Notice how the resulting example graphs (depending on the actual values of the constant parameters different optimizations may be possible) have fewer nodes than the original graph shown under *Preparation*.

In a final step, the high level node, e.g. *instanceof*, which is part of the intermediate graph representation of the bytecode to be compiled, is *replaced* with the subgraph computed during the *specialization* phase. This is called *installation*.

6.1.5 GraalVM Native Image

In a nutshell, GraalVM's Native Image makes it possible to compile a program written in Java not just to bytecode (as e.g. done with `javac`) but to a standalone, platform specific executable.

Normally, a Java program is compiled only to bytecode by an application developer. This bytecode, stored in `.class` files, must then be run on top of a JVM by the user later on. As we saw earlier, the JVM either interprets the bytecode produced by `javac` or JIT compiles it to optimized, platform specific machine code, see subsection 6.1.2 for more details.

However, with GraalVM's Native Image it is instead possible to compile our Java program not only to bytecode but directly to a standalone binary, just as for instance a C program would be if compiled with the local toolchain (e.g. `gdb` on a Linux system). The resulting binary, called a *native image*, is of course platform dependent (unlike a program compiled to bytecode) but crucially it can be run on a supported platform without the user having to install a JVM as a prerequisite first.

How it works [103] See figure 6.3 for a schematic overview of the build process of a native image (a standalone, platform specific binary, e.g. an ELF file on Linux).

1. As a first step our `.java` source files still have to be compiled to Java bytecode using e.g. `javac`. This first step is principally no different than compiling to bytecode during normal Java development (see **(1)** in figure 6.3).

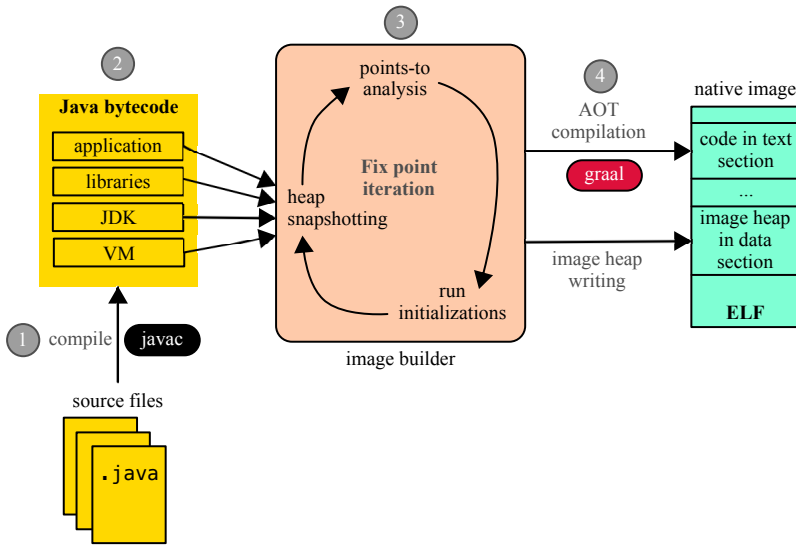


Figure 6.3: The GraalVM Native Image build process, adapted from figure 1 in the *Initialize Once, Start Fast: Application Initialization at Build Time* paper [103]

2. Note that we need to not only compile to bytecode our own (hand-written) source files but also any other code needed by our application, such as is contained in libraries we are including – anything which isn’t already available in bytecode form. This is important because crucially, during the rest of the native image build process, all bytecode will be treated equally: No matter whether it originally came from a source file written by us, belongs to a library we are using or even if it is a class file part of the JDK itself, it will all be processed the same way by the native image build process [103].

This extends even to the bytecode making up the VM our code will eventually run on at runtime (not the Java Hotspot VM but the Substrate VM which is also part of the GraalVM project and can be seen as a stripped down JVM). Note that this VM only provides a rudimentary runtime since much of the usual JVM machinery

(e.g. JIT compilation, profiling etc.) isn't needed for a standalone executable. What is still needed is e.g. memory management and thread scheduling.

All bytecode no matter its origin (see **(2)** in figure 6.3) is passed to the image builder.

3. The image builder (see **(3)** in figure 6.3) performs three important sub-steps: First static analysis is used to determine which code is reachable from the `main` method and therefore needs to be AOT compiled. Then, in a second sub-step the initializers of all classes determined to be reachable are run to compute a sort of “preliminary” heap. Third, this preliminary heap’s object graph is crawled by the image builder to potentially discover newly reachable code.

For the first sub-step, the image builder uses a points-to analysis to determine which classes, methods and fields are “reachable” from the main entry point of the application, meaning actually used by the application. The results of this static analysis then determine which classes’ initializers need to be executed in a next sub-step (also done by the image builder).

After running all initializers, a so called snapshot of the heap is computed as the third sub-step: The algorithm tasked with this takes the result of the points-to analysis as its input. It then starts its exploration of the current object graph with all the values stored in the static fields of the currently reachable classes (which were computed previously when the classes were initialized). The static fields of the currently reachable classes serve as *root pointers* into the object graph. Starting with these, the heap snapshot algorithm iteratively explores the entire object graph, collecting all objects that that it encounters in a set as a sort of *snapshot*.

However, as the algorithm crawls the object graph, it is entirely possible for it to discover new object types: It may find objects with a runtime type corresponding to a class which had not previously been marked as reachable. If this happens, the points-to analysis is updated to take note of this for its next run. The just described three sub-steps – points-to analysis, running the initializers of all reachable classes, computing a snapshot of the heap –

keep repeating until no new classes are discovered by the snapshot sub-step anymore and a fixed point is reached.

4. When finally no new object types have been discovered during the computation of the (last) snapshot of the heap, the resulting set of objects collected by the algorithm in its last run (it starts from scratch every time) is saved as the *image heap*. The image heap is the precomputed part of the heap and written to the *data* section of the resulting binary. The code that has been determined to be reachable and has been AOT compiled is stored in the *text* section of the binary (see (4) in figure 6.3).

Note that to make all of this work, Native Image makes what is called a *closed world* assumption: It assumes that all used classes, methods and fields are known at *compile time*. This is quite remarkable because in Java the assumption is usually that of an *open world* instead: This is embodied by dynamic features such as the Java Reflection API. (Note that these can still be used together with Native Image but configuration files informing the build process about classes it would otherwise be unaware of are required.)

One of the downsides of using Native Image can surely be guessed by the reader at this point: The build process itself can take quite a long time to reach a fixed point (comparatively) and is resource intensive.

Why it's desirable There are several reasons why we might want to give up the Java platforms famous portability and instead directly compile our Java application to a native, platform dependent but standalone executable. First, while Java applications compiled to bytecode and e.g. distributed as a JAR file are platform independent in so much that they can run on MacOS, Linux as well as Windows systems this is of course only true if these systems already have a JVM pre-installed the application can run on. Otherwise your potential customer might first have to hunt down a compatible JVM release and install it or you have to make sure to bundle a suitable JVM release together with your application, resulting in potentially bigger application sizes and more complexity. Thus, depending on the concrete context, it might be desirable to instead compile native executables for all supported platforms

on which the application will then be directly executable, requiring no additional setup.

Aside from not having to bundle our program with a full JVM (or having to rely on the user already having a compatible version installed), the Native Image technology provides a number of other advantages [103]: For one, compiling all the way to optimized machine code means that there is no need for any warm-up period until peak performance is achieved. This is perhaps the most important selling point of Native Image because it promises faster start-up times.

Java programs often warm up slowly because of the interpreter first having to profile the code before it is passed to the JVM's JIT compiler. Native executables containing machine code do not have this characteristic (at least not to this extent though it is of course still possible that hardware caches will warm up over application runtime, resulting in better performance). In addition, as we have just seen, with the Native Image build process it is possible to *precompute* a big part of the heap of a Java application and thus complete much initialization before the application has even started running. This can result in significantly faster start-up.

Another advantage of Native Image is that the binaries are usually smaller and the memory footprint is also smaller [103].

Note that aside from obvious advantages the main motivation behind using GraalVM's Native Image feature for *Mistletoe* is simply that *isolates*, which I set out to instrument, are in fact only supported in this specific context. If not using Native Image to AOT compile to a standalone executable (called a native image in this context), trying to use any *isolates* results in a compilation error.

6.1.6 Isolates

An *isolate*, as described in detail here [102], is a new virtualization primitive that aims to virtualize the JVM itself. This is implemented by creating a new heap context for each new *isolate*, in other words a heap disjoint from the default heap of the application (and that of the other *isolates*). Just like virtual machines (e.g. Virtualbox) virtualize operating systems and allow to run multiple operating systems on the same hardware, isolates virtualize certain aspects of the JVM by allowing multiple heaps to co-exist in the same process.

Why it's desirable *Isolates* are advantageous not just to multiplex the JVM itself, allowing multiple applications or application parts to run on top of the same JVM, but also because they allow for a degree of isolation: For instance, since the heaps of different *isolates* are disjoint, the garbage collector is triggered independently for each *isolate*. In other words, one part of the application won't be affected by another part of the application (running in a separate *isolate*) having triggered the garbage collector – making performance more reliable.

Even better, if an *isolate* is destroyed instead of relying on the garbage collector to eventually reclaim all memory used by a particular, no longer active part of an application, the memory can be freed in one go: Since the memory used by the now inactive part of the application is disjoint from all other memory used by the rest of the application, it can be reclaimed all at once without the garbage collector ever having to run.

But *isolates* also provide another form of isolation: When *isolates* are enabled, to implement the disjoint heaps, all object references are interpreted as *relative* to the current heap. This means that it isn't possible for code running in a particular *isolate* to even address objects stored in another *isolate*, making *isolates* an effective mechanism to prevent accidental leakage of objects. (Note that this heap relativity is implemented by adding object references to the currently active heap's base address which is stored in a register to minimize overhead).

How it works Figure 6.4 depicts schematically the virtual address spaces of a traditional Java application with a single Java heap (on the left) and an application creating an isolate at some point of its execution (on the right). Note that the virtual addresses increase from the bottom of the figure to its top with the kernel space to be imagined somewhere further above. The virtual address spaces are not drawn to scale (due to space constraints). Also note that what's depicted is intentionally simplified to illustrate specific points and that not everything that will be present in a real virtual address space is shown. For instance, both memory allocated by the JVM by actually calling `malloc(3)` and by mapping anonymous memory by calling `mmap(2)` is simply depicted as part of the heap.

The virtual address space to the left of figure 6.4 depicts the layout of a traditional Java application for comparisons sake. The application

is running on the Java Hotspot VM [69] (which is written in C++). Assume the JVM is in turn running on a Linux based operating system on AArch64 hardware. The figure shows the stack, located somewhere at the top of the user's part of the virtual address space, the JVM's heap and the code being run (part of the executable mapped into the virtual address space).

Note that the actual application here as viewed from the point of view of the operating system is in fact not our Java application but the JVM *itself*. Therefore what is mapped into the virtual address space as the code to be executed is not the code making up the Java application that will run on top of the JVM but the code of the JVM itself. The same is true for the heap: The heap starting at *heap start* and ending at *brk* is not the Java application's heap – instead it is the heap of the JVM itself. The Java application's heap is stored there as well but the JVM keeps more state stored there besides the application's heap: It also stores the stacks of application threads and its code cache (containing the JIT compiled application code) there, among other things.

The virtual address space to the right of figure 6.4 shows the virtual address space layout of a native image (a Java application compiled to a standalone executable with GraalVM Native Image, see subsection 6.1.5). The native image creates a single *isolate* at some point of its execution and executes part of its code in this new isolation context before switching back. The additional *isolate* has not yet been destroyed.

The two disjoint heaps right under *Miscellaneous mappings* correspond to one *isolate* each. There are two (and not just one) additional heaps because when *isolates* are enabled a default *isolate* will be created automatically before `main` is even called. It serves as the default context. The second *isolate* (and its heap) are created explicitly on the applications behest when it calls `Isolates.createIsolate(...)`.

Both of these additional heaps are created by calling `mmap(2)`, which allows to map the *image heap* (which is part of the native image binary, recall subsection 6.1.5) right at the start for both of the additional heaps. Note that the *image heap* is mapped with COW semantics. This means that while the backing physical memory is shared for maximum efficiency, if the code running in either isolate writes to this part of their heap at any point, the backing physical memory will be copied by the operating system. This is to avoid one isolate affecting another (and to

avoid changing the native image itself).

To give an example of this, assume a class *MyClass* used by the Java application has a `static` field called *myField*. That field will be initialized during the native image's build process and will have become part of the *image heap* (unless configured otherwise by the application developer). If now, after the Java application has created an additional isolate and code execution has switched to it, code running in the additional isolate writes to this field, this update will not be visible for any code running in the other, default isolate due to COW. Instead the memory storing the field will be duplicated.

Stacks Note that while *isolates* have their own private heap, they do not have their own stack. Instead stacks are associated with threads which are orthogonal to *isolates*: Threads can attach and detach from *isolates*, they have a $n : m$ relation to *isolates*. When a thread switches into an *isolate*, a new stack frame is added to its stack. This stack frame is removed again when the thread switches back. If it instead switches to a third *isolate*, this is again handled by adding another stack frame. The draw back to this is that this means that stack frames belonging to different *isolates* can be present on the same thread stack.

Code Note that the application code (this includes both the AOT compiled code of the Java application and the code of the VM) is shared between *isolates* as well. This is beneficial since it is mapped into the virtual addresses space as execute and read-only.

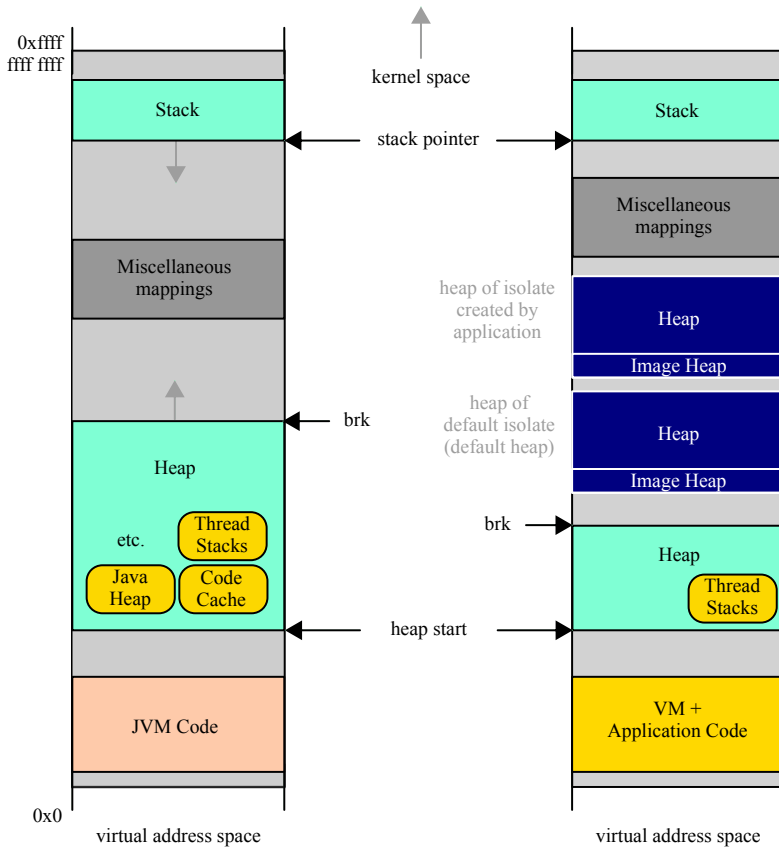


Figure 6.4: Virtual address space layout in the presence of *isolates* [102], [22], [33]: The main feature of *isolates* is their disjoint heaps, as shown in the virtual address space on the right. On the left the virtual address space layout of a traditional Java application is depicted.

Note that this visualization presents a simplified view for practical purposes: It is not to scale and both memory allocations done by calling `malloc(3)` and by installing anonymous memory by calling `mmap(2)` are summarized in the heap space (with the exception of the *isolates*' heaps which are also installed via `mmap(2)`).

Example Let's now discuss an example code snippet showcasing how to create, switch into, run code in, switch back from and destroy an *isolate*. See figure 6.5: First, on line 2, a new *isolate* is created and the current thread is attached to it. This returns a `IsolateThread` which is to be treated as an opaque handle [61]. Next we get the same type of handle for our current context (remember that a default *isolate* is automatically created for us).

```

1 // Create a new isolate
2 IsolateThread newContext = Isolates.createIsolate(Isolates
3           .CreateIsolateParameters
4           .getDefault());
5 IsolateThread currentContext = CurrentIsolate.getCurrentThread();
6
7 // Copy a string into the new isolate
8 String msg = "Hello from my isolate.";
9 ObjectHandle stringHandle = copyString(newContext, msg);
10
11 // Execute the method doSomething in the new isolate
12 ObjectHandle resultHandle =
13     doSomething(newContext, currentContext, stringHandle);
14
15 // Resolve and delete the resultHandle,
16 // now that we are executing in the original isolate again
17 ByteBuffer result = ObjectHandles.getGlobal().get(resultHandle);
18 ObjectHandles.getGlobal().destroy(resultHandle);
19
20 // Tear down the isolate, freeing all temporary objects
21 Isolates.tearDownIsolate(newContext);
22

```

Figure 6.5: Example code creating a GraalVM *isolate*. Partially based on the native-netty-plot demo. [63].

Now we wish to call a special function called `doSomething()` and we want its code to be executed in our newly created *isolate*. See figure 6.6 for the definition of `doSomething()`. Notice that it is annotated with `@EntryPoint` and that it has a special signature: It is a `static` method and it takes as its first parameter the new context we wish the code to

run in.

However, before we can call `doSomething()` in figure 6.5 we have to figure out how we can pass it an argument: We'd like to pass it a `String` named `msg`, defined on line 8, but if we simply make `doSomething()` take a `String` as its third argument and try to call it with `msg` (`doSomething(..., ..., msg)`), this will fail to compile. The reason for this is of course that `msg` actually contains a *reference* to an object stored on the default *isolate*'s heap. Passing `doSomething()` this reference is meaningless since it is relative to our own heap and therefore can only be properly resolved in the context of the default *isolate*.

```

1  @CEntryPoint
2  private static ObjectHandle doSomething(
3      @CEntryPoint.IsolateThreadContext isolateThread newContext,
4      IsolateThread currentContext,
5      ObjectHandle stringToPassIn
6  )
7  {
8      //Resolve and delete the stringHandle,
9      // now that we are executing in the new isolate
10     String msg = ObjectHandles.getGlobal().get(stringToPassIn);
11     ObjectHandles.getGlobal().destroy(stringToPassIn);
12
13     [...]
14
15     return result;
16 }
17

```

Figure 6.6: Excerpt of the static `doSomething` method called in code snippet 6.5

Because of this, on line 9 we call a function called `copyString()` to store a copy of `msg` somewhere where it can be accessed by code running in the newly created *isolate*. This function returns a special handle that we can then pass to our function `doSomething()` on line 13. The details of the `copyString()` function have been omitted for brevity but they are correspondent to what happens after we have called `doSomething()`

on line 13: After we have switched into the new *isolate* we use the same mechanism of getting and setting global `ObjectHandles` [62] on the lines 10 and 11 of figure 6.6 to resolve *msg*.

As we returned from `doSomething()` we automatically switch back to our default *isolate*. On the lines 17 and 18 of figure 6.5 we resolve the returned `ObjectHandle` to a `ByteBuffer` computed by `doSomething`. On line 21 we finally destroy the *isolate* we created which causes its whole heap to be freed all at once.

Implementation When I looked at the implementation of *isolates* in the GraalVM I found that their creation, destruction and switching into another *isolate* is implemented using *snippets*, see subsection 6.1.4. This, combined with the fact that *isolate* operations are already executed during the build process of a native image and at times when not everything has yet been set up made it hard to observe and instrument them: For example, it is not possible to just instrument them with a `System.out.println` call.

6.2 Motivation

In this section I motivate the development of *Mistletoe*, a small C runtime tapping into GraalVM [55]. I explain the choice of GraalVM as a systems research platform and why achieving intra-process-isolation is especially important in the context of Java applications. I also describe the challenges I encountered when looking into the implementation of *isolates* and trying to instrument them.

6.2.1 Why do systems research on GraalVM

GraalVM is a widely used commercial product, a state-of-the-art JVM with production grade performance that – thanks to the Truffle Framework – can run everything from Fortran over Java to Python applications. It is used by both Oracle databases and Oracle Cloud Infrastructure (OCI) as well as companies such as NVIDIA, Goldman Sachs and Alibaba Group [56]. Even when focusing only on its ability to run Java applications (or compile them to standalone binaries), its applicability is still high since Java is still very widespread (e.g. appearing as the number 2 in IEEE Spectrum’s 2023 rankings of programming languages [10]).

This breadth of usage makes GraalVM interesting as a research platform due to the chance for research to have real world impact. Also GraalVM’s state-of-the-art performance allows for scientific comparison with other state-of-the-art JVMs, e.g. Java Hotspot VM [69].

6.2.2 Intra-process-isolation in GraalVM

At first sight *isolates* seem ideal for intra-process-isolation, e.g. imagine a server written in Java communicating with multiple mutually distrustful clients. Being able to process each request in a different isolate guarantees that the objects created to serve the request will be placed in mutually disjoint heaps. This then also allows to reclaim that memory all at once without having to rely on the garbage collector by simply destroying the isolate after severing the connection to the client.

Isolates also seem perfect for implementing a lightweight serverless platform in Java, allowing to run each user supplied function in a separate *isolate* but still in the same process.

However, though *isolates* have disjoint heaps as we saw in subsection 6.1.6, the isolation they provide is not yet strong enough: For instance they do not have separate stacks, instead a thread is able to switch from one isolate into another. This means accumulating stack frames belonging to different isolates on one and the same stack.

And though the object references being heap relative makes it impossible to directly refer to an object part of another isolate, the memory containing the heap of the other isolate is in principle still accessible for an errant pointer: Since the operating system has no knowledge whatsoever of isolates and any other thread running in the same process but in the other isolate must be able to access the other heap simultaneously, the virtual memory range containing the other heap cannot simply be marked as inaccessible in the page tables.

The new hardware extensions MPK (by Intel) [27] and POE/PIE by (Arm) [6] seem a promising tool to address this. However, modifying GraalVM's implementation of *isolates* presents some challenges, as I describe in the next subsection, subsection 6.2.3.

6.2.3 Challenges for systems research on GraalVM

As mentioned already in the introduction of this chapter, Java Virtual Machine (JVM)s deliberately abstract away the hardware they are running on. They do this in an attempt to provide applications with as uniform a platform as possible. While often very desirable, this generally goes against the grain of systems programming as systems programmers tend to try to get as close and unobstructed access to the actual hardware as possible.

While it is sometimes an option to “just punch a hole” so to speak in the many layers of abstraction to get access to the hardware by calling for instance a C library (e.g. by using JNI) that's generally a point solution. The JVM is of course not aware of it being subverted, believing itself to be in complete control of resources such as *platform threads* and virtual memory. As a result of this belief clashing with the powerful access the C programming language provides the programmer with, some of its underlying assumptions may end up being broken. In other words the application programmer has to be extremely careful not to inadvertently corrupt the application state past recovery and causing for instance a segmentation fault.

To sidestep this problem it seems best to be able to implement systems programming code that needs unobstructed access to the hardware directly in the JVM *itself*. This would allow the systems programmer to be aware of the JVM's actions, having their code called only at specific points.

However, accomplishing this in the context of GraalVM is no easy task as GraalVM is a complex system – as is any production grade JVM (this is also evidenced by the fact that even today many popular programming languages don't have a high performance runtime implementation as the investment of resources to build one is significant [104]). It is non-trivial to navigate GraalVM's source code (the build process even dynamically pulls in other repositories) and most systems programmers won't have in-depth prior knowledge of JVM's internal workings (see subsection 6.1.2).

The problem is made even more acute by the complex meta compilation system employed by GraalVM, centered around *snippets* [78], see subsection 6.1.4. In addition to this, when operating directly from within a JVM such as GraalVM, a programmer may be restricted by the fact that the environment is not yet completely initialized when their code is called, e.g. any function resulting in a heap allocation might be impossible to call at certain points.

Taking all of this into account, the rabbit hole gets very deep indeed.

6.3 Mistletoe

In this section I present the conceptual design of *Mistletoe*, a lean C-runtime able to “tap” into GraalVM [55] at specific execution points. First, I explore the main design goals motivating the design of *Mistletoe*. After that I discuss the tradeoffs that need to be considered before presenting the final design I arrived at: A lean C-runtime presenting a (system) programmer with a familiar environment that can *hook* into and run code at specific execution points of GraalVM.

Following this, I discuss possible types of applications that can be built on top of this novel (system programming) runtime in subsection 6.3.2.

6.3.1 Design

The design of *Mistletoe* is inspired by the idea of taking a system programmer’s approach to a production grade JVM, specifically GraalVM, as already touched upon in the previous section, section 6.2. The core aim is to “break out of the box”, so to speak, to gain low level hardware access and the ability to observe the operations of the JVM itself with as little overhead as possible. This just described approach as well as several additional requirements give rise to a number of design goals shaping the conceptual design of *Mistletoe*:

Design goals of the *Mistletoe* project

- **Make higher level VM operations, e.g. managing *isolates*, observable:** One of the main goals of the project (and original inspiration) is to investigate GraalVM’s *isolates* (see subsection 6.1.6). This is to be done by making the operations the VM performs to manage *isolates* observable from outside of the VM itself. More generally, the goal is to be able to observe the VM perform high level operations such as are implemented by *snippets* (refer to subsection 6.1.4 for more details).
- **Target the SubstrateVM:** As *isolates* are only supported when building so called native images with GraalVM’s Native Image (see subsection 6.1.5), its a goal of the *Mistletoe* project to specifically

target the SubstrateVM. This VM, part of the GraalVM project, supports native images instead of the usual Java Hotspot VM [69].

- **Run custom code at specific points of the VM’s execution:** A programmer should be able to have their own code called at specific points of the VM’s execution chosen beforehand. Because of the point above, VM in this case refers to the SubstrateVM.
- **Reduce complexity:** With any system of non-trivial size complexity will eventually become an issue: As functionality (and the code base with it) grows, it gets harder and harder to onboard new programmers because of the steeper learning curve. This also makes it difficult to collaborate across different fields of expertise. Case in point, a system programmer is unlikely to be closely familiar with the internal workings of a JVM. Because of this, one of the main aims of the design of *Mistletoe* is to reduce complexity for the (system) programmer, making it unnecessary to know much about GraalVM to inject new functionality into the VM’s execution.
- **Add as little overhead as possible:** *Mistletoe* itself should not add too much overhead, the goal is to build a lightweight way to hook into the VM’s execution that does not come with a huge performance penalty attached. Only then will it be feasible to use *Mistletoe* in practice.
- **Provide a clean slate:** To keep complexity at bay (see above) and to address the fact that at some points of a Java VM’s execution, some operations might be unsafe (e.g. if the Java heap isn’t yet set up or we’re in the process of switching heaps no operation may be performed that will result in a heap allocation), *Mistletoe* aims to provide the programmer with a *clean slate*: To be precise, *Mistletoe* should present the programmer with a familiar, unrestricted execution environment that makes it unnecessary to have any detailed knowledge about the VM’s current internal state.
- **Enable low level access:** The system programmer needs (comparatively to Java) low level access to the system’s resources to be able to perform typical system programming tasks: They need to be able to manage memory directly by calling `malloc(3)` or

`mmap(2)`, write to registers by e.g. accessing the IO space etc. *Mistletoe* must enable this.

- **Break out of the box:** JVMs creates an abstraction of the machine they are running on, hiding the actual hardware and its state from the programmer. As this is incompatible with system programming, *Mistletoe* must provide a way to “break out of this box” so to speak. It must punch a hole in the many levels of abstraction between the Java programmer’s usual view of the machine and the (comparatively) unobstructed view C/C++ provides.
- **Enable experimentation:** *Mistletoe* should enable a system programmer to easily experiment with new hardware features and have Java programs benefit from them without having to rewrite the whole VM/ wait until support becomes available.

Tradeoffs Before presenting the design of *Mistletoe*, which is motivated by the design goals just given above, I provide a short overview of the most important tradeoffs that have to be considered:

- **Safe, easy use vs. capability:** While it might be tempting to design a powerful runtime allowing the programmer to change even the VM’s core functions, this needs to be balanced against the following design goals: reducing complexity for the programmer, providing them with a familiar programming environment and requiring no prior knowledge of VM internals. As powerful tools imply increased responsibility and tend to have a steeper learning curve to use them effectively, it follows that there is a tradeoff between the ease of use of *Mistletoe* and the novel runtime’s capabilities.
- **Functionality vs. overhead** This tradeoff is related to but not identical to the one mentioned just above: When giving a user privileged access and powerful capabilities, for reasons of safety and soundness, this not only requires intimate knowledge of the inner workings of the VM but also of its current *state*. (To avoid accidentally disrupting the VM, e.g. at certain times some operations are not safe to perform.) This in turn implies copying state

out of the VM and into the runtime. Inevitably, this incurs *overhead*. Because of this the functionality achievable with the runtime needs to also be balanced against the need for good performance, not just ease of use.

- **Flexibility vs. short development time:** As with any project, there is also a tradeoff between generality, which implies flexibility, for the user and development time. This being a research project I chose to lean more towards the latter. *Mistletoe* is not meant to be a full-fledged product but simply a prototype inspiring future research.

Mistletoe’s design After having covered both design goals and tradeoffs above, I now present the design I ultimately arrived at: Figure 6.7 shows a conceptual overview. At the top we have the Java application itself which is being run by the VM. Note that even in the case of the Java application having been compiled to a native image by GraalVM Native Image, the application is still a Java application in need of e.g. a garbage collector, threading support etc [66]. That functionality is provided by the Substrate VM the application is running on even if in this case the code of the VM in question has been AOT compiled together with the application itself.

Both the application and the VM as well as *Mistletoe* all run in the same process. Below this process we have the operating system layer and the hardware layer. (This is of course a simplification, not shown is e.g. the so called *glue code* layer or the layer representing the firmware.)

Mistletoe itself is represented by both the rectangular box to the right and the arrows labeled *hooks* pointing to it. This mirrors the fact that there are essentially two parts to *Mistletoe*: the so called *hooks* placed in the VM running the Java application and a dynamically loaded library, called *libmistletoe*, forming the core of the runtime itself.

The *hooks* are placed in advance in relevant parts of the VM that are to be instrumented/ observed or that the programmer wants to react to being executed. The idea is that the *hooks* are minimally invasive, influencing the execution of the VM itself as little as possible. This is because inspired by the tradeoffs explored above, I chose to keep *Mistletoe* a passive observer that isn’t supposed to alter how the VM itself operates.

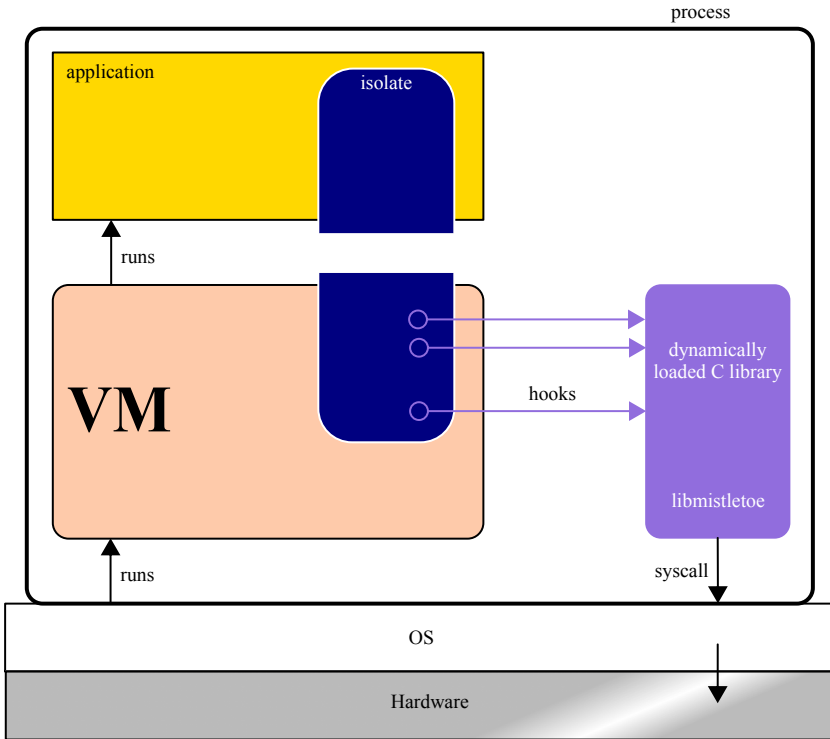


Figure 6.7: Visualization of the design concept of *Mistletoe*: *Mistletoe*, a lean C-runtime taps into GraalVM [55].

The *hooks*, when triggered, will cause functionality present in *libmistletoe* to be invoked at specific points of the execution flow of the VM (depending on where the *hooks* have been placed).

The library itself, called *libmistletoe*, is dynamically loaded and offers the programmer the usual, unrestricted C programming environment. It enables both low level access to and a low level view of the hardware we are running on (comparatively speaking, *libmistletoe* still needs to interact with the hardware via the operating system's system calls). Note that *libmistletoe* is envisioned as a dynamically loaded library to minimize re-

compilation: this makes it easier to change functionality without having to repeat the often time-consuming build process (for either GraalVM or even only the Java application itself as native images may take some time to build).

As for how to make use of *Mistletoe*, the programmer can either place their code directly in the *Mistletoe* library or implement functions in a secondary, also dynamically loaded library which is then loaded and resolved by the *Mistletoe* library at runtime. (This option allows for a more uniform interface.)

Let's walk through a concrete example: We have a Java application that makes use of *isolates* during its lifetime and the programmer wishes to run custom code whenever a new one is created. Assume the appropriate hooks have been placed and the application's corresponding native image has already been built. Now we run it: First, the VM starts executing the Java application. At some point the Java application then makes use of a high-level Java feature, e.g. it checks the runtime type of an object with `instanceof`, creates a new object with `new` or - as in our specific case - creates an *isolate*. These operations are implemented by the *snippets*, discussed in section 6.1.4, which are provided by the VM to the *graal* compiler.

Note that the separation between the application and the VM and the upper usage part and the lower implementation part of the *isolate* shown in figure 6.7 is simply to symbolize the application triggering *isolate* management operations that are then taken care of by the VM.

As e.g. these *isolate* management operations are executed, this will cause the *Mistletoe hooks* to be triggered (in our example they have been placed in *isolate* management code) and will cause execution to jump to code making up *libmistletoe*. Depending on the functionality desired by the programmer, execution might now pass into kernel space (if e.g. a system call is invoked) though this doesn't necessarily have to be the case always. It is also possible to simply perform an operation in user space, e.g. increment a counter to track the number of isolates created.

Finally, after all custom code triggered has been executed, control will first return to the VM and then to the application (as e.g. a function such as `create_isolate` returns).

6.3.2 Applications

In this subsection I touch upon possible usage scenarios for *Mistletoe*. For this I briefly sketch what sort of application could be built on top of *Mistletoe* as potential future work. Note that this subsection references (but isn't equivalent to) work presented in the papers [13, 98], for which I am a co-author, and work done for the the Master's theses [97, 93] which I co-supervised.

Below I first focus on ideas combining memory protection mechanisms such as e.g. Intel's MPK [27] and Arm's POE/PIE [6] with *Mistletoe*. Then I discuss hardware assisted profiling and how to make use of accelerators: In the course of this I outline an option for using *Mistletoe* to implement hardware assisted, zero-overhead profiling for Java applications and I discuss the possibility of using *Mistletoe* together with an accelerator, such as can be implemented on an FPGA.

Hardware assisted memory protection One option to use *Mistletoe* is to combine it with some kind of memory protection mechanism. Then, once hooks have been placed at the right places, access permissions can be changed depending on which part of an application is running.

Obvious candidates for this are so called memory protection keys as provided by Intel's MPK [27] and Arm's POE/PIE [6]. They allow to set permissions *per thread* and once the protection domains have been set up, permissions can be changed with very little overhead (see section 5.2 for more details).

Depending on the use case simply calling `mprotect(2)` might also be an option though extra care would have to be taken since an update to the page table protection bits done via `mprotect(2)` will always affect all threads in a process. Also, as a system call `mprotect(2)` naturally incurs more overhead than a mere write to a user space writable register (comparing to memory protection keys).

Most importantly, *Mistletoe* enables the user to easily try out new experimental memory protection mechanisms: such as e.g. *Noodles*, a project which I cover in the next chapter, chapter 7, and which offers a new system call to limit the virtual address space of a thread to a subset of the virtual address ranges present in the original virtual address space. Or actual hardware capabilities, such as Capability Hardware Enhanced RISC Instructions (CHERI) [95], where a pointer (meaning

an address) is additionally paired with upper and lower bounds which are then enforced by the hardware. (Note that while capabilities can be enforced in software [21], CHERI is a modern implementation of a so called hardware capability.)

Being able to call out into C allows us to use any mechanism that can be accessed from the familiar C environment. All of these mechanism can be added to the VM's execution with very little overhead thanks to *Mistletoe* and called at appropriate moments by placing the right hooks.

An example for this would be placing hooks in the code of the VM that handles switching from one *isolate* into another (by resetting the heap base pointer) such that on each *isolate* switch custom code is called. This would allow to change permissions on virtual memory ranges, either by utilizing memory protection keys or `mprotect(2)` (though we have to keep in mind that all changes to page table permissions as they are made by `mprotect(2)` always apply to all threads running in the same process).

One of the challenges that would have to be addressed for this to work is how to communicate to the *Mistletoe* runtime which data has been placed where. This can potentially be addressed by passing a tiny bit of state via the hooks to the runtime, such as an address and a length argument.

Hardware assisted profiling Another usage case worth exploring is leveraging *Mistletoe* in combination with hardware support for instruction flow tracing to enable very fine grained application profiling. In a nutshell, the idea is that while trace data is being collected and processed *online* (meaning during application runtime), *Mistletoe* is used to place markers in the voluminous stream of trace data to relate it to points in the Java application's control flow (e.g. to mark a switch into a different *isolate*).

The inspiration for this approach is the Enzian system [13]: Enzian is a research hybrid computing platform designed by the Systems Group at ETH Zürich. It combines a server-class ARMv8 CPU (Marvell Cavium ThunderX-1, 48 cores) with a large FPGA (Xilinx XCVU9P Ultrascale+ FPGA) to form an asymmetric NUMA system. The FPGA is connected to the CPU via a high speed interconnect (based on the CPU's native coherent interconnect, with a theoretical maximum bandwidth of 30 GiB/s

in each direction).

Work has been done [97, 93] to explore if this close coupling of FPGA and CPU could be leveraged to take better advantage of Arm’s hardware tracing subsystem called CoreSight [35]. CoreSight exposes an array of tracing and debugging components, including the Embedded Trace Macrocell (ETM) which allows to trace a specific core with zero-overhead. The granularity of the trace data produced is so fine, that it is possible to reconstruct the complete control flow of an application from the combination of the captured trace data and the executed binary at a later point in time.

However, the flip side of this is that CoreSight produces an extremely high volume of data, quickly filling up any but the largest buffers. (According to Weingarten et al. [98], a core may produce up to a 1 GB/s of trace data depending on the behavior of the application being traced!) This large amount of data is not only difficult to store and costly to process but also hard to relate to the high level control flow of an application being traced in any meaningful way.

A solution to this problem could be an end-to-end pipeline system, implemented on Enzian, where produced trace data is not stored and then analyzed at some later point but instead decoded and analyzed as it is being produced, at line rate: As an application is being executed, the trace data produced by the ETM is transferred to the FPGA via the high speed interconnect (instead of filling up some kind of storage medium). On the FPGA the trace data stream is then fed into a decoder able to decode the trace data conforming to the ETMv4 specification at line rate.

Mistletoe in turn is used to send signals encoding high level application knowledge, giving more context, to the FPGA. These signals are then fed into an analyzer, also implemented on the FPGA, together with the output of the decoder and the application binary itself. (Note that with a CPU that supports branch broadcasting the binary itself is not needed to reconstruct the control flow from trace data but the Enzian’s ThunderX-1 CPU does not support it.)

There are many different types of analyzers that could potentially be implemented like this: To give a brief example, imagine an analyzer scanning the instruction flow of an application for disallowed, i.e. privileged instructions, as a form of security verification (guaranteeing

nothing bad has happened). However, as an added complication, the instructions disallowed are disallowed only for specific *parts* of the application being traced. *Mistletoe* could then be used to inform the analyzer, if the application is currently running in a privileged mode and verification should be disabled or if it is switching into a “supervised mode” and verification should thus be enabled.

Note that while work on implementing an end-to-end pipeline as briefly sketched above is currently ongoing [97, 98, 93], this is still future work.

Communicating with accelerators Nowadays there exist a plethora of different hardware accelerators, for a wide array of different use cases: Components that accelerate cryptography, compression or regular expression application, components meant to speed up neural networks (NPU), GPUs, etc. The hardware landscape is fast changing and with the advent of FPGAs even algorithms that don’t warrant/ are impractical to design and build an ASIC for can now be implemented in hardware.

Unfortunately, it is not easy to adopt these benefits for Java applications: Until libraries wrapping functionalities become available or the VM itself is changed – it is cumbersome to communicate with these new devices, potentially requiring setting up JNI (which in turn requires additional configuration for native images [65]).

Instead, the approach proposed here is to use *Mistletoe* to more easily integrate the desired functionality directly into the VM itself, allowing all applications to benefit from the hardware accelerator without any additional overhead for the application programmer.

6.4 Implementation

In this section I describe the implementation of the *Mistletoe* prototype, a small C-runtime making it possible to tap into GraalVM [55] at specific points. I covered the intended design of *Mistletoe* and the design goals that inspired it in the previous section, section 6.3. By contrast this section focuses on the technical implementation details.

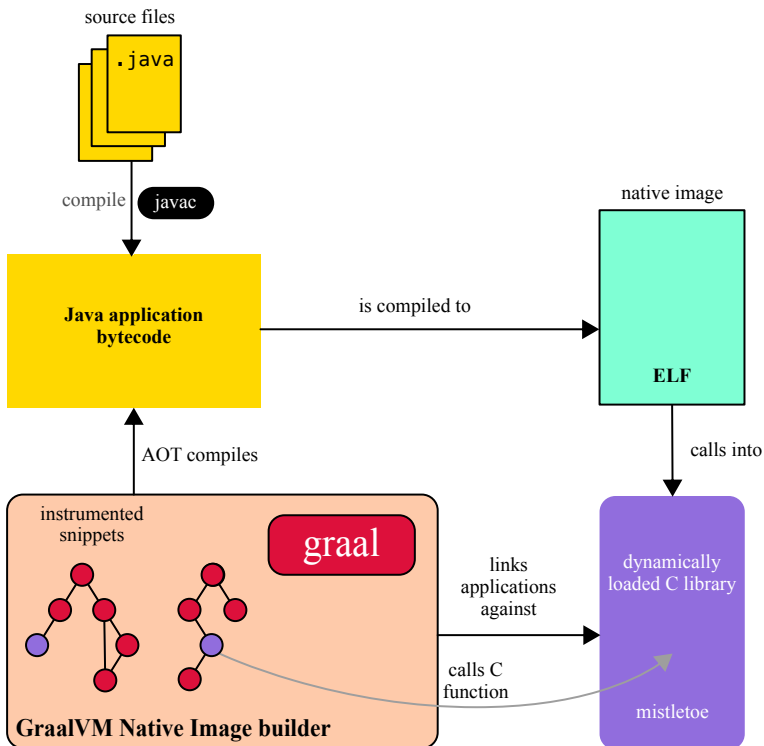


Figure 6.8: Visualization of how the implementation prototype of *Mistletoe* functions: *Mistletoe*, a lean C-runtime taps into GraalVM [55] by instrumenting snippets of interest with calls out to the C-runtime, which lives in a dynamically loaded library.

After outlining how the prototype functions, I explain the finer im-

plementation details by walking the reader through a concrete example application built on top of *Mistletoe*: an *isolate* based billing application (see subsection 6.4.1).

While figure 6.7 presented in the previous section, section 6.3, gave a *conceptual* overview of *Mistletoe*'s design, by contrast figure 6.8 visualizes how the concrete implementation of the *Mistletoe* prototype fits into the bigger picture: At the bottom left of the figure we have the GraalVM Native Image builder. This tool can be built from the open source, community edition GraalVM code base which is available on Github [54].

In figure 6.8 Native Image is used to AOT compile a Java application. It's source files have been compiled to bytecode by `javac` in a previous step (see the top left of figure 6.8). Recall that, as mentioned in subsection 6.1.5, Native Image compiles the Java application to a standalone executable, a *native image*.

On Linux, the resulting executable is an ELF binary (see top right of figure 6.8). This binary has been linked by the Native Image builder against the dynamically loaded library, called *libmistletoe*, forming the core of *Mistletoe* (see the bottom right of figure 6.8).

To achieve this, I made a small modification to the GraalVM code base so that the resulting Native Image builder not only links the native images it builds against `glibc` but also against *libmistletoe.so*. Linking against *libmistletoe.so* makes it possible to call designated C functions defined in *libmistletoe.so* – the so called hooks – directly from within the GraalVM code base.

Note that it is not possible to use JNI instead, as could potentially be done from within the Java application's code. Also note that using a dynamically loaded library makes it possible to change the behavior of the hook functions (though not their signature) without having to recompile the Native Image build tool. In fact, not even a previously created native image binary needs to be recompiled to take advantage of any update to *libmistletoe.so*.

While linking against *libmistletoe.so* allows to call designated C function directly from within the GraalVM code base, *snippets* still present a special challenge: As discussed in subsection 6.1.4, *snippets* are not called as functions are (even though they look like annotated Java functions). To quickly recap, instead *snippets* are compiled to *graal*'s intermediate graph representation and the resulting subgraph is then *inlined* in

a program's corresponding intermediate representation graph, replacing the high level node (e.g. `instanceof`) implemented by the *snippet* in question.

All of this means, that to be able to insert *Mistletoe's* hooks inside a *snippet*, triggering a switch to the *Mistletoe* runtime, some extra steps are needed: Essentially they ensure that the call to the designed C function serving as a hook is defined as special kind of node that can be inlined in the *snippet* subgraph itself. These extra steps are described in detail for the *isolate* billing example application discussed below, in subsection 6.4.1.

6.4.1 Case study: Per-isolate billing

In this subsection, I explain in detail how the *Mistletoe* prototype functions by walking the reader through a how I implemented a per-*isolate* billing application on top of *Mistletoe*. As building an application for another use case would be analogous, this serves as a proof of concept of how to make use of *Mistletoe*.

The goal of the per-*isolate* billing application is to measure how many cycles (as can for instance be measured by reading the TSC register on an x86-64 machine) are spent per *isolate*. This is of interest so that different parts of an application can be appropriately billed.

A potential usage scenario for this would isolating user provided workloads in a serverless setting, though of course the billing application proposed here is just one possible building block: For a serverless platform more than mere heap isolation (as is provided by *isolates*) is needed.

Note that threads are orthogonal to *isolates* (they can attach and detach), so per-*isolate* billing is not equivalent to per-thread cycle counting.

Placing hooks The first step to implement our billing application is to identify target functions or *snippets* in the GraalVM codebase so that we can place suitable hooks: To be able to appropriately count the number of cycles spent per *isolate*, we must be notified each time when we switch *isolates* so that we update our counters. (Note that we assume a single-threaded application scenario.) It is therefore straight forward to target the function that causes the heap base to be (re-)set (recall

that all object references are relative to the current heap base, this is the main feature of *isolates*).

The function that handles setting the heap base can be found in `substratevm/src/com.oracle.svm.core/src/com/oracle/svm/core/graal/snippets/CEntryPointSnippets.java` and is called `setHeapBase`. It is not in itself a *snippet* but because it is used by several *snippets*, any attempt to simply call out into *Mistletoe* from there will cause the compilation of the Native Image build tool to fail.

The reason for this is that all functions that are called by a *snippet* have to be suitable for being inlined by the *graal* compiler. This is not the case for a function defined externally as a C function in a dynamically loaded library and calling such a function in `setHeapBase` transitively makes `setHeapBase` unsuitable for being inlined. Therefore simply calling into *Mistletoe* from there results in a compilation error. Instead we have to perform some additional set-up to transform the call into something that can be inlined in a *snippet*.

Apart from being notified each time the heap base changes we also need to be notified each time an *isolate* is destroyed. This is to avoid running into problems if an *isolate*'s heap is placed at exactly the same location in the virtual address space the heap of a previously destroyed *isolate* occupied. Otherwise we might bill the CPU cycles a later workload consumed to the first *isolate* with its heap starting at a particular virtual address.

Concrete implementation steps Note that the steps discussed in this paragraph were derived by mirroring the patterns employed for other methods in the GraalVM codebase which also call out to C and have to be called by *snippets*.

We start with two C functions, one for each event we wish to be notified of. The signatures we declare in `mistletoe.h`, the corresponding definitions we place in `mistletoe.c`. See figure 6.9 for the signatures. These are the functions we want to call from within GraalVM whenever the heap base gets set or an *isolate* is destroyed.

While `mistletoe_setHeapBase` takes care of calculating how many CPU cycles have passed since an *isolate* was switched into and adding it to the account of the *isolate* in question, `mistletoe_freeIsolate` is responsible for removing an *isolate*'s account (and reporting the number

```

1 void mistletoe_setHeapBase(uint64_t isolate , uint64_t base);
2
3 void mistletoe_freelsolate(uint64_t isolate , uint64_t base);

```

Figure 6.9: The function signatures we declare in *mistletoe.h*, with corresponding definitions being present in *mistletoe.c*. These files are both either compiled directly into the dynamic library called *libmistletoe.so* are dynamically loaded by dispatch code in the library.

of cycles it consumed) once it ceases to exist. For bookkeeping, a simple hash table using the *isolate*'s heap base address as key is used.

Note that we can either directly compile *mistletoe.o* into our dynamically loaded *libmistletoe.so* library or alternatively, if we need a more clearly defined interface for an actual production use case, it is possible to compile our C functions as a separate library. The needed functions can then be resolved and loaded dynamically by *libmistletoe.so* in turn, e.g. by using a configuration file to bind the right functions to stubs.

```

1 @CFunction(value = "mistletoe_setHeapBase",
2           transition = Transition.NO_TRANSITION)
3 public static native void mistletoe_setHeapBase(
4     PointerBase isolate , PointerBase base
5     );
6
7 @CFunction(value = "mistletoe_freelsolate",
8           transition = Transition.NO_TRANSITION)
9 public static native void mistletoe_freelsolate(
10    PointerBase isolate , PointerBase base
11    );
12

```

Figure 6.10: Definition of the hooks as `native` methods inside GraalVM

As a next step, to be able to call our two functions `mistletoe_setHeapBase` and `mistletoe_freeIsolate` from within the GraalVM code base, we define our functions as `native` methods within GraalVM itself, see figure 6.10. Notice the `@CFunction` annotation and that the methods take

the same number of arguments, each with a compatible type (essentially just a pointer), as the C functions.

These definitions allow us to call our C functions from within GraalVM. However, to also be able to integrate the hooks into *snippets* as just discussed (see also subsection 6.1.4 for details on this topic), additional set-up is needed. First, we wrap calling the C functions in normal Java methods that we annotate with `@SubstrateForeignCallTarget`, see figure 6.11. We also annotate them with `@Uninterruptible` because `setHeapBase` is also annotated with this annotation. (The documentation of `@Uninterruptible` informs us that methods annotated with it may only call other methods also marked as `@Uninterruptible`). Note that `VMThreads` is the class where the definitions of the `native` methods live in the prototype.

```

1  @Uninterruptible(reason = "Unknown thread state.")
2  @SubstrateForeignCallTarget(stubCallingConvention = false)
3  private static void mistletoeSetHeapBase(PointerBase base) {
4      VMThreads.singleton().mistletoeSetHeapBase(base);
5  }
6
7  @Uninterruptible(reason = "Unknown thread state.")
8  @SubstrateForeignCallTarget(stubCallingConvention = false)
9  private static void mistletoeFreelsolate(PointerBase base) {
10     VMThreads.singleton().mistletoeFreelsolate(base);
11 }

```

Figure 6.11: The actual implementations of the foreign calls, calling a wrapper for our CFunctions.

Next, we define `ForeignCallDescriptors` to reference the Java methods we have just defined in figure 6.11 to wrap our hooks, see figure 6.12. Note that the `CEntryPointSnippets` class is where we defined the methods in figure 6.11.

These descriptors we store in an array, see figure 6.13 that already contains other `ForeignCallDescriptors` and which will later be used to register the foreign calls.

```

1 public static final
2 SubstrateForeignCallDescriptor MISTLETOE_SET_HEAP_BASE =
3     SnippetRuntime.findForeignCall(
4         CEntryPointSnippets.class,
5         "mistletoeSetHeapBase",
6         false,
7         LocationIdentity.any()
8     );
9
10 public static final
11 SubstrateForeignCallDescriptor MISTLETOE_FREE_ISOLATE =
12     SnippetRuntime.findForeignCall(
13         CEntryPointSnippets.class,
14         "mistletoeFreeIsolate",
15         false,
16         LocationIdentity.any()
17     );
18

```

Figure 6.12: This code creates two `ForeignCallDescriptors`, these are needed to link the foreign call usages to the corresponding implementations.

```

1 public static final
2 SubstrateForeignCallDescriptor [] FOREIGN_CALLS = {
3     ... ,
4     MISTLETOE_SET_HEAP_BASE, MISTLETOE_FREE_ISOLATE,
5     ...
6 };

```

Figure 6.13: This code adds our new `ForeignCallDescriptors` to the array called `FOREIGN_CALLS` which will later on be used to register the foreign calls.

The last step needed is the definition of two *intrinsic nodes*, see figure 6.14. The two methods called `runtimeCallMistletoeSetHeapBase` and `runtimeCallMistletoeFreeIsolate` will be passed the `ForeignCall-`

Descriptors we defined in figure 6.12.

```

1  @NodeIntrinsic(value = ForeignCallNode.class)
2  public static native void runtimeCallMistletoeSetHeapBase(
3      @ConstantNodeParameter ForeignCallDescriptor descriptor,
4      PointerBase base
5  );
6
7  @NodeIntrinsic(value = ForeignCallNode.class)
8  public static native void runtimeCallMistletoeFreelsolate(
9      @ConstantNodeParameter ForeignCallDescriptor descriptor,
10     PointerBase base
11 );

```

Figure 6.14: The definition as *intrinsic nodes*. Note the keyword `native` and the `@NodeIntrinsic` annotation.

```

1  @Uninterruptible(reason = "...",
2  mayBeInlined = true)
3  public static void setHeapBase(PointerBase heapBase) {
4      if (hasHeapBase()) {
5          runtimeCallMistletoeSetHeapBase(
6              MISTLETOE_SET_HEAP_BASE,
7              heapBase
8          );
9          writeCurrentVMHeapBase(heapBase);
10         ...
11     } else {
12         ...
13     }
14 }

```

Figure 6.15: Excerpt of the `setHeapBase` method I modified, taken from the GraalVM codebase [55], `substratevm/src/com.oracle.svm.core/src/com/oracle/svm/core/graal/snippets/CEntryPointSnippets.java`

Now we can include calls out to *Mistletoe* into *snippets*. See figure 6.15 for an example of this: It shows the targeted `setHeapBase` method to which we have added a call to `runtimeCallMistletoeSetHeapBase`. This will result in our C function `mistletoe_setHeapBase` being called each time the heap base is set (this happens on line 9 of figure 6.15). The last steps for `mistletoe_freeIsolate` are analogous.

```

1 App**: main: Started running.
2 App**: We're running on: Substrate VM, 19.0.1, Oracle [...]
3 App**: java.library.path: /usr/lib64:/lib64:/lib:/usr/lib
4 App**: GraalVM ver.: 23.0.0-dev, release: false
5 App**: main: Before creating a new isolate.
6 App**: main: After creating a new isolate.
7 App**: main: Before copying a string into the new isolate.
8 App**: main: After copying a string into the new isolate.
9 App**: main: Before doing something in the new isolate.
10 App**: Isolate: Hello from within the isolate.
11 App**: main: After doing something in the new isolate.
12 App**: main: Before resolving and deleting the resultHandle.
13 App**: main: After resolving and deleting the resultHandle.
14 App**: main: Before tearing down the isolate.
15     base=0x0000e26526600000, cycles=24113916
16 App**: main: After tearing down the isolate.
17 PER ISOLATE ACCOUNTING
18     base=0x0000e26526600000, cycles=0
19     base=0x0000e26527e00000, cycles=336796753
20
21 HEAT MAP
22 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

```

Figure 6.16: Sample console output showing the per-*isolate* billing application built atop *Mistletoe* in action.

For a sense of the kind of output generated by the billing application we just implemented atop of *Mistletoe*, please see figure 6.16. Note that all lines prefixed with `App**` result from print statements present in the Java application which has been compiled to a native image and is now being run. The application first creates an *isolate*, then it copies an input string into it before switching into it and running a function in the

isolate.

Line 15 shows output generated by the billing application after the application has switched back into the default *isolate* and destroyed the other *isolate* (the one it created). Finally on line 16 the default *isolate* is also torn down and below it, the billing application reports its statistics. The heat map shown is for debugging the hash table (which has been set to 16 slots deliberately here to make the output fit on the page).

An in-depth evaluation of the performance overhead incurred by the just described billing application built atop *Mistletoe* can be found in the next section, section 6.5.

6.5 Evaluation

In this section I present my results from benchmarking *Mistletoe*. I evaluate the overhead imposed by *Mistletoe*, specifically by the per-*isolates* billing application. I show that while there is naturally a little bit of an overhead incurred, it is small by comparison and may be considered acceptable depending on the use case.

I evaluated *Mistletoe* with two different benchmark suites: I employed my own, self-developed micro benchmark suite especially tailored to measuring the performance of the operations relating to *isolates* (*isolate* creation, switching into an *isolate*, destruction of an *isolate*). There is - to my knowledge - no other benchmark focusing solely on or even properly exercising *isolates*.

I also used DaCapo's *avro*, *h2*, *lindex*, *lusearch*, *pmd* and *xalan* benchmarks, DaCapo being a well known Java benchmark suite[8], as application benchmarks to simulate a non-isolate-centric application.

6.5.1 Micro benchmarks

In this subsection I describe and analyze the results obtained by running my self-developed micro benchmark suite on both an unmodified version of GraalVM and one extended with the per-*isolate* billing application which I presented in subsection 6.4.1. The experiments were performed on an x86-64 machine with the following specifications:

- 12 × 12th Gen Intel(R) Core(TM) i5-12600
- 64 GB RAM (2 × 32 GB)
- Ubuntu 22.04.5 LTS

Note that as the overhead incurred by *Mistletoe* is entirely dependent on which hooks are placed at which code sites, I chose to evaluate *Mistletoe*'s feasibility via an example application, the per-*isolates* billing application.

The micro benchmark suite is implemented in Java and compiled to a native image using either an unmodified version of GraalVM (hash: *de8da6cee7c0d6efe1f44970518cf02ad7f933ad*) or the same version but extended with the billing application built atop of *Mistletoe*. The resulting native image is then executed on the x86-64 machine to run the

micro benchmark suite. It consists of 7 different micro benchmarks implemented in the form of normal Java classes – all extending a general benchmark class called `Benchmark` providing common functionality. Time is measured by calling `System.nanoTime()`. I describe all different micro benchmarks in the list below.

Note that the micro benchmark suite is designed to do the following for each of its runs: first it executes the operation to be benchmarked x times as a warm-up and then measures how long it takes to execute it y times (the graphs in this subsection all show the normalized results). Depending on how many runs are scheduled for a specific micro benchmark, this is then repeated z times, so that we can take the median of the z runs (with the 95th percentile plotted as error bars). I settled on this procedure because some of the operations to be benchmarked only take a couple of nanoseconds, making exact timing of a single operation vulnerable to interference. This way, jitter is minimized.

- **SteadyStateSingleIsolate** This micro benchmark runs the following workload: First it increments a `private static` counter, then it switches into a pre-created *isolate* where it also increments the same `private static` counter field, then it switches back to the default isolate and decrements the counter again. Note that due to the `private static` field of the `SteadyStateSingleIsolate` class being initialized during the building of the native image and the field thus becoming part of the image heap, each *isolate* (both the one created and the default one) has its own copy of the field (the first write after the first switch into the created *isolate* triggers COW).
- **SteadyStateIsolates** By contrast this micro benchmark measures how long it takes to create a new *isolate*, switch into it, increment a `private static` field, switch back to the default *isolate* and then finally destroy the *isolate* again.
- **CreateIsolates** This micro benchmark is quite straightforward, it simply measures how long it takes to create a new *isolate* (note that it does not switch into it).
- **DestroyIsolates** Just like the micro benchmark above, this micro benchmark simply measures how long it takes to destroy an *isolate*.

- **RunInIsolates-[10, 100, 1000]** This micro benchmark is similar to *SteadyStateSingleIsolate* except that instead of always switching into the same pre-created *isolate* (and then back) it alternates between either 10, 100 or 1000 also pre-created *isolates* in a round-robin fashion.

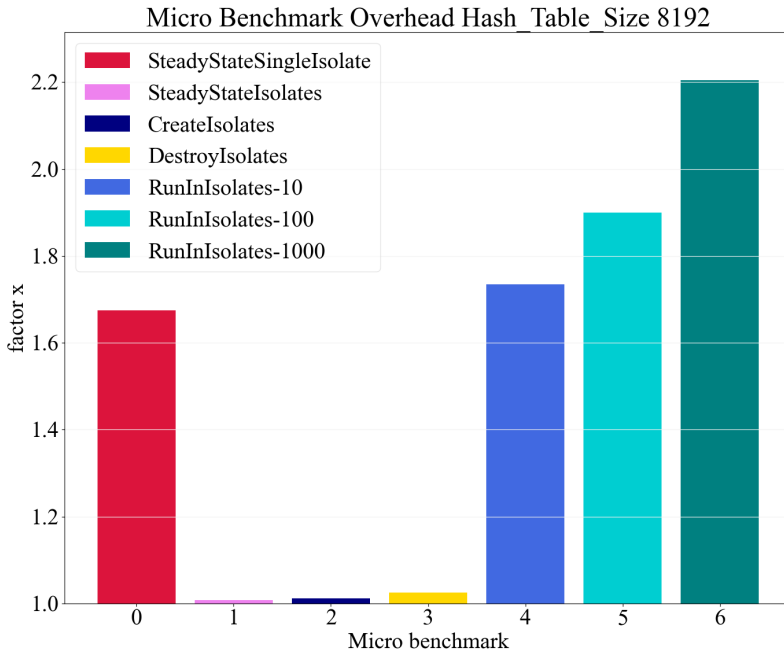


Figure 6.17: **Mistletoe micro benchmarks overhead overview:** This graph depicts the overhead incurred by instrumenting GraalVM with the per-*isolate* billing application. The y-axis depicts the \times times compared to the unmodified version. Hash table size: 8192 slots

Figure 6.17 presents an overview of the results obtained by running the micro benchmark suite executables built with the two different versions of GraalVM (unmodified vs. modified with the billing application).

Note that the hash table used internally by the per-*isolates* billing application was set to contain 8192 slots. The graph depicts the runtime overhead incurred (y-axis) by each of the different micro benchmark types (x-axis).

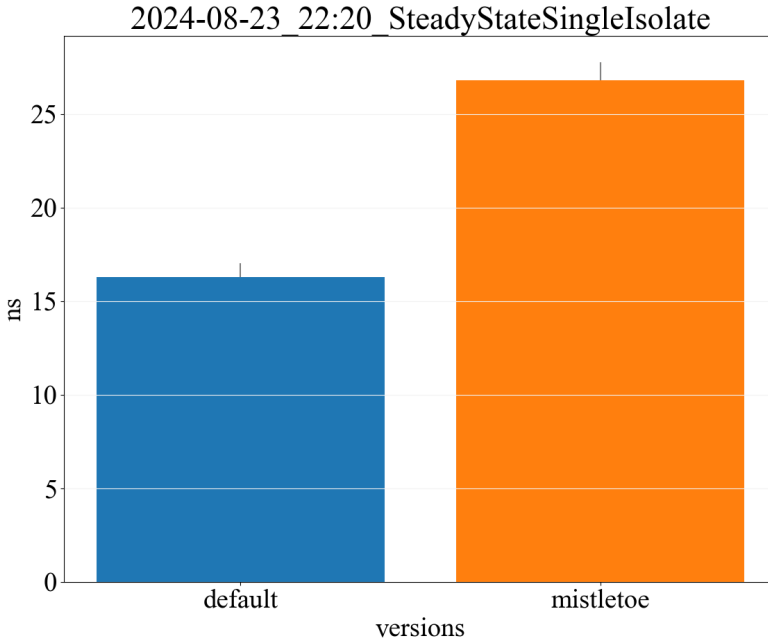


Figure 6.18: **Overhead in nanoseconds:** This shows the results for the *SteadyStateSingleIsolate* benchmark built with both an unmodified version of GraalVM (*default*) and on one modified with the *isolates* application (*mistletoe*). Hash table size: 8192 slots

As can be seen, the overhead incurred is negligible for the micro benchmarks *SteadyStateIsolates*, *CreateIsolates* and *DestroyIsolates*. That is as expected since creating or destroying an *isolate* is an expensive operation. The *Mistletoe* hooks and the work performed by the billing application itself is sufficiently lightweight to hardly make any differ-

ence, creating or destroying an *isolate* is so expensive that it dominates runtime.

As for *SteadyStateSingleIsolate*, *RunInIsolate-10*, *RunInIsolate-100* and *RunInIsolate-1000* we can observe that the overhead increases with the number of active *isolates* that the execution switches between. This makes sense and can be mitigated by choosing a sufficiently big hash table depending on the needs of the actual workload. Note also that as the per-*isolate* application is merely a prototype, the hash table implementation was not the focus of this work and can thus be further improved.

Though the overhead observed for the micro benchmarks investigating how long it takes to switch between *isolates* might seem considerable in figure 6.17, when looking at the concrete numbers it becomes clear that this is also due to the operation in question lasting only tens of nanoseconds.

Figure 6.18 shows a comparison of the concrete runtimes measured for the unmodified version of GraalVM and the one instrumented with the per-*isolate* billing application. The overhead imposed by *Mistletoe* turns out to be about 10 ns in this particular scenario.

When increasing the number of pre-created *isolates* to either 10, 100 or 1000, see figure 6.19, the overhead per operation remains about the same for 10 and 100 – a bit less than 20 ns – but increases for 1000 different *isolates*. This behavior is largely determined by the size (number of slots) chosen for the hash table – as well as the implementation used. As evidence for this see figure 6.20.

Figure 6.20 plots the overhead measured for the three different micro benchmarks *RunInIsolates-1000*, *RunInIsolates-100* and *RunInIsolates-10* by hash table size with the hash table size ranging from 32 to 8192 slots. As can be observed, for the three micro benchmarks in question we reach a point of diminishing returns at 8192 hash table slots. This is also the reason I chose to set the number of hash table slots to 8192 for the experiments plotted in the graphs previously shown in this subsection.

To conclude this subsection, the per-*isolate* billing application does incur an overhead on switching *isolates*, mainly because the operation itself is very fast and only takes a little bit more than 15 ns for the unmodified GraalVM when switching between a single new *isolate* and the default *isolate*. Note however that depending on the use case and how

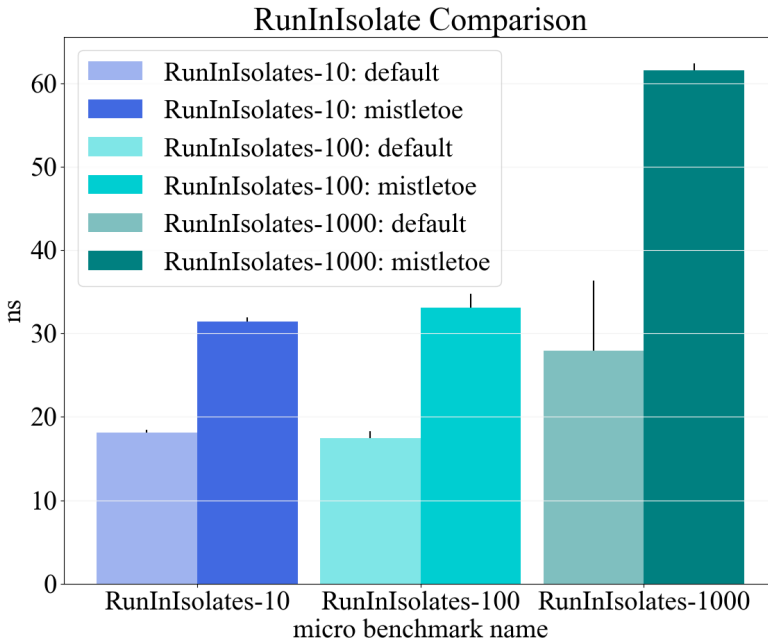


Figure 6.19: **Overhead in nanoseconds for multiple *isolates***: This graph plots the results in nanoseconds obtained for the *RunInIsolate* benchmark (switching between 10, 100 or 1000 different *isolates*) built with both an unmodified version of GraalVM (*default*) and on one modified with the *isolates* application (*mistletoe*). The hash table size was set to 8192 slots.

often switching is required the overhead of about 10 additional nanoseconds (in case of switching between two *isolates*) may be considered acceptable. Note further that as the number of *isolates* we switch between increases, not only the overhead imposed by the billing instrumentation increases (depending on the size of the internal hash table) but also the cost of switching *isolates* as measured for the unmodified version, see figure 6.19.

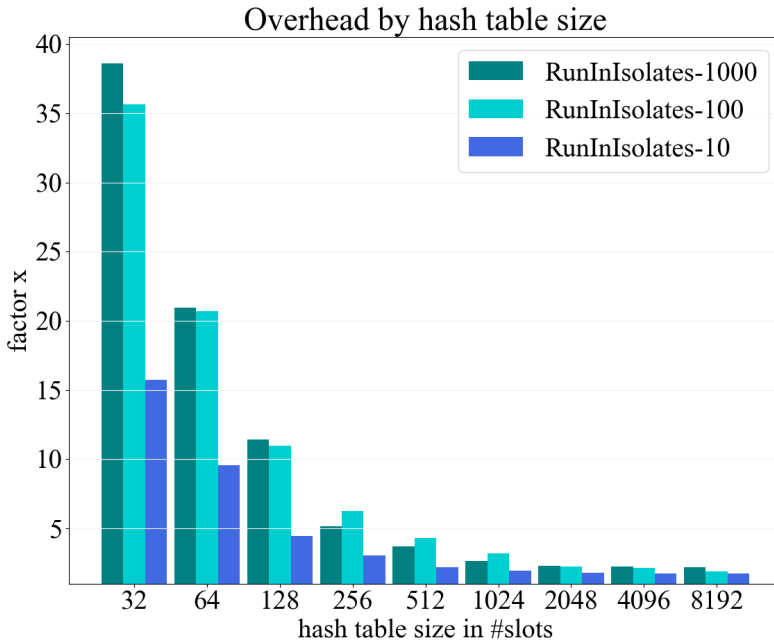


Figure 6.20: **Overhead vs. hash table size for *RunInIsolates*:** This graph depicts the overhead incurred by the *RunInIsolate* benchmark built with the modified version of GraalVM over the *RunInIsolate* benchmark built with the unmodified version of GraalVM. The number of isolates to switch between was set to 10, 100 or 1000.

On the other hand, creating and destroying an *isolate* is such an expensive operation that despite placing a hook such that *Mistletoe* is called each time an *isolate* is destroyed, no additional overhead is noticeable.

6.5.2 DaCapo benchmark suite

To verify that the per-*isolate* billing application built atop *Mistletoe* doesn't impose an unexpected overhead in use cases that are not espe-

cially *isolate* centric (but still create at least a default *isolate*) I evaluated both an unmodified version and one instrumented with the billing application with the standard Java benchmark suite, DaCapo [8].

The DaCapo benchmark suite is an open source benchmark suite especially developed for Java [75]. It contains a series of application centric benchmarks that use non-trivial amounts of memory since memory management (e.g. garbage collection) has a crucial impact on the performance of Java programs.

I chose the DaCapo benchmark suite because `mx`, GraalVM's build system, supports running some of the benchmarks which are part of this benchmark suite out of the box. Note that not all of the benchmarks part of the DaCapo suite are supported by GraalVM's Native Image as built from the source code of the version of GraalVM I used. I simply selected the benchmarks that were supported and that ran without errors. Those that matched those criteria are briefly described below:

- **avroa:** This benchmark was added after the publication of the original DaCapo paper in 2006 [8]. According to the DaCapo Project's Source Forge page `dacapobench.sourceforge.net` [74], it simulates running several different programs on a grid of AVR microcontrollers and was added in 2009.
- **h2:** This benchmark was also added later on, in 2009, and replaces the older `hsqldb` benchmark. It is similar to JDBCbench which executes transactions against an example banking application [74].
- **luindex:** This benchmark runs a tool indexing text according to the original DaCapo paper [8]. The tool in question is Apache Lucene [4] and the document set used consists of Shakespeare works and the King James bible.
- **lusearch:** Also according to the original DaCapo paper, this benchmark runs a tool searching text [8]. The document set worked on is the same as for the `luindex` benchmark above and Apache Lucene [4] is used to search for a range of keywords [74].
- **pmd:** According to the original DaCapo paper [8], a source code analyzer for Java (it analyzes Java classes to find problems [74]).

- **xalan:** : According to the original DaCapo paper [8], this benchmark runs an XSLT processor.

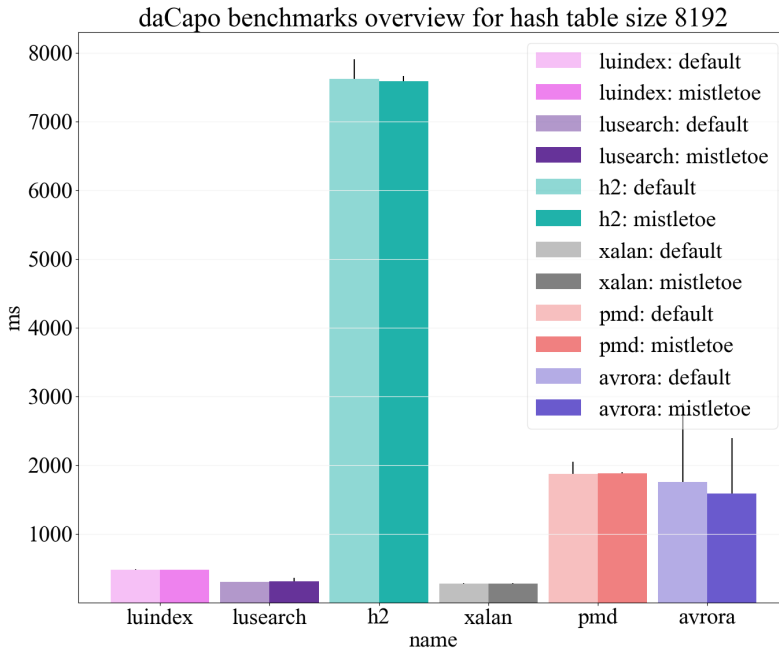


Figure 6.21: This figure plots the runtime of several benchmarks part of the DaCapo benchmark suite [8] in milliseconds. For each benchmark, the bar to the left (drawn in a lighter color) corresponds to running the benchmark on an unmodified version of GraalVM (commit: `de8da6ce e7c0d6efe1f44970518cf02ad7f933ad`) while the darker bar to the right corresponds to running the same benchmark on GraalVM instrumented with the billing application built atop *Mistletoe*. The version of the DaCapo benchmark suite used is `9.12-MR1-git+2baec49`.

Each of the benchmarks chosen was run 3 times on both an unmodified version of GraalVM and one modified with the billing application built atop *Mistletoe*. The hash table of the billing application contained

8192 slots. The machine used to run the benchmarks was the same x86-64 machine as used for the micro benchmarks described in the previous subsection, subsection 6.5.1.

Note that the version of the DaCapo benchmark suite I used is `9.12-MR1-git+2baec49`, the version of GraalVM used is commit `de8da6ce e7c0d6ef e1f44970 518cf02a d7f933ad` (both as the unmodified version and the version to base my modifications on).

The results of running the chosen DaCapo benchmarks are plotted in figure 6.21. Note that for each specific benchmark, the bar to the left (drawn in a lighter color) corresponds to running the benchmark on an unmodified version of GraalVM while the darker variant to the right corresponds to running the same benchmark on GraalVM instrumented with the billing application. Also note that each bar corresponds to the median of 3 different runs with the 95th percentile plotted as error bars.

As can be seen, apart from slight variations due to noise, the addition of the billing application imposes no noticeable overhead. This is for one due to the fact that the DaCapo benchmarks don't properly make use of *isolates* (since they were developed without this GraalVM Native Image specific feature in mind). Secondly, as we have seen when analyzing the micro benchmark results in subsection 6.5.1, the overhead imposed by the billing application is only noticeable if we have very high *isolate* switching rates. Otherwise the cost of other much more expensive operations, e.g. *creating* an *isolate* dominates.

6.6 Conclusion and future work

In this section, after having evaluated the proposed C-runtime *Mistletoe* in the last section, section 6.5, I outline possible future work before concluding this chapter. Note that as possible usage scenarios in which we could leverage *Mistletoe* are outlined in a previous subsection, subsection 6.3.2, the next subsection focuses on the next steps for the runtime, *Mistletoe*, itself.

6.6.1 Future work

Offering utilities One obvious avenue to explore to further improve the proof-of-concept per-*isolate* billing application is to employ a more sophisticated hash table implementation. Instead of developing one, an open source implementation that has already been extensively tested and validated should be chosen. This hints toward *Mistletoe* itself potentially coming with a set of default data structures/ algorithms for the convenience of the user. The the dynamically loaded library forming the core of the *Mistletoe* runtime could easily provide a set of commonly used functions and data types. What exactly to offer the user and what is best left to third-party libraries is one potential direction for future research.

Formalize API On the topic of convenience, the API of *Mistletoe* is currently only roughly sketched for a maximum of flexibility during experimentation but an actual product should of course come with a much more polished, carefully thought out way to interact with its user. The design and implementation of such an API is another possible future direction of research. As already touched upon briefly, one option for an API is to have the user implement a pre-defined set of functions that can then be loaded dynamically by *libmistletoe*.

Passing more information to *Mistletoe* Another topic to look into is how to best share information on data placement in the virtual address space with *Mistletoe* and if that is even desirable (e.g. where the heap of an *isolate* has been placed in the virtual address space). As already touched upon, one design goal of *Mistletoe* is to avoid tight coupling

of the VM and *Mistletoe* itself so that as little knowledge concerning JVMs as possible is required for being able to use *Mistletoe*. This makes it not straightforward to decide if passing information regarding data placement should even be passed to *Mistletoe* as it might add unnecessary overhead and complexity.

Predefined hooks Another idea that could be explored is that if we can call into C when something with regards to *isolates* happens, we could also do that for example for memory management operations or type checks with `instanceof` (among many other possibilities) by placing the appropriate hooks: This would open up all sorts of interesting avenues of future research. To facilitate this it might be desirable for *Mistletoe* to offer the user a set of pre-defined or even pre-installed hooks that could be enabled/ disabled depending on the actual needs of the user.

6.6.2 Conclusion

In this chapter I presented *Mistletoe*, a lean C-runtime making it possible to tap directly into GraalVM itself. I also described and evaluated a proof-of-concept per-*isolates* billing application built atop it and showed that it is feasible to use it in an actual application.

Mistletoe was inspired by my work exploring GraalVM's *isolates* [102] and their applicability to intra-process-isolation. The challenges I encountered in the course of this gave rise to the design goals and requirements shaping *Mistletoe* (as discussed in section 6.3). Most importantly, the idea at the core of *Mistletoe* is to take a system programmer's approach to a production grade JVM and *break out of the abstraction box*: *Mistletoe* allows to place hooks in GraalVM and thus call custom code at specific points of the VM's execution. This allows the system programmer to easily customize the VM itself without having to possess detailed knowledge of JVM internals, facilitating (among other things) experimentation with new hardware features.

In subsection 6.3.2 I discussed several different possible use cases for *Mistletoe*, meaning applications that might be built atop the proposed runtime. I presented the design and implementation of a proof-of-concept application built atop *Mistletoe*, the per-*isolate* billing application. In section 6.5 I investigated how the instrumentation with the

billing application impacts GraalVM's performance and concluded that depending on the concrete usage scenario and *isolate* switching rate, the use of *Mistletoe* is feasible in practice.

To conclude, even though all JVMs (including GraalVM) deliberately abstract the underlying hardware it is possible to directly access and experiment with new hardware features, e.g. communicate with accelerators or make use of MPKs by using *Mistletoe*. It represents an easy and straightforward way for system programmers to implement new functionality that can then be called directly from within GraalVM itself.

Chapter 7

Noodles: Fat Threads

Note: The work presented in this chapter is also described in a related paper called *Noodles: Intra-Process-Isolation with Fat Threads* [26]. It has not yet been published.

As discussed previously, in chapter 5, intra-process-isolation, not just inter-process-isolation, is needed for several different classes of applications: To reiterate, example use cases include lightweight multi-tenant serverless platforms like *Cloudflare's* Workers [9] and *Oracle's* GraalOS [53] but also high performance server applications like the popular web servers Apache [3] and nginx [47] (among many others).

In this chapter I present a novel approach to intra-process-isolation, called *Noodles*, which is primarily intended for lightweight serverless platforms. As part of the design of *Noodles* I propose a new operating system level abstraction called a *noodle* or a “fat thread”, situated in terms of granularity between the isolated but heavyweight process and the lightweight but unisolated thread. The central idea of *Noodles* is to give every thread in a process its very own virtual address space to provide memory isolation where necessary while still making it easy to share memory where desired.

For evaluation purposes I've implemented a prototype as a patch to the Linux kernel version 5.15.116 [14]. The results presented in section 7.5 show that in relevant scenarios *Noodles* incurs only 10%-11% of

the cost of the `fork(2)` system call.

In addition, as a proof of concept, I've protected a simple HTTPS server based on OpenSSL's echo server demo [51] against a Heartbleed [23] inspired memory vulnerability. I've also benchmarked said HTTPS server with Apache benchmark [2] to show it's indeed feasible to deploy *Noodles* in such a setting from a performance point of view.

The structure of this chapter I start this chapter by providing the motivation for this work in section 7.1 (note that the general background and motivation concerning the need for intra-process-isolation is covered in chapter 5). After motivating *Noodles*, this is followed by a description of its design in section 7.2, including a discussion of the semantics of the new operating systems level abstraction called a *noodle*. I also point out the design tradeoffs that inspired the design of *Noodles* and I introduce the programming model I envision to make use of *Noodles* in section 7.2.

After detailing the design and semantics of *Noodles* I provide the reader with some required background information detailing how the Linux kernel [88] works under the hood in section 7.3. After laying down these necessary foundations, I then follow this up with a detailed description of the actual implementation of the *Noodles* prototype in section 7.4. Note that the prototype consists of a kernel patch to the 5.15.116 mainline Linux kernel [14] and a support library running in user space called *libnoodles*.

In the section after that, section 7.5, I then present a numerical performance evaluation of my prototype. I detail the experimental setup and explain the motivation behind the design of my micro benchmarks as well as discussing the results obtained. I also present the graphs showcasing the performance of my prototype. Following the numerical evaluation, I will present my proof of concept HTTPS application based on OpenSSL's echo server [51] in subsection 7.5.5 and the accompanying Apache benchmark results.

Finally, I conclude and sketch necessary future work to turn *Noodles* from a bleeding edge prototype into a fully working, production ready addition to the Linux kernel in the last section of this chapter, section 7.6.

Note that the approach taken in section 7.4, which describes the implementation of *Noodles*, is to look beyond common but general operating system level abstractions. Instead, I discuss and evaluate the ideas

presented in the previous section, section 7.2 detailing *Noodles* design and programming model, with the help of a concrete example, in this case the Linux kernel, version 5.15. The aim of this is to show that it is often non-trivial to implement an abstract design in a concrete, real-world system as new ideas might clash with previously made design decisions.

7.1 Motivation

The main goal for the work presented in this chapter, the *Noodles* project, is to design an intra-process-isolation mechanism which enables the creation of a memory access restricted execution context. Importantly, this new context must still be running *inside* the original process context.

A side goal of the project is to implement this new mechanism specifically for systems running Linux kernel based operating systems.

As already alluded to in chapter 4, the focus of the work presented here is on *memory* isolation though of course there are other, equally important aspects of intra-process-isolation, e.g. making sure that access to the file descriptors is limited to trusted parts of an application. However, in this chapter any form of intra-process-isolation besides memory isolation is considered out of scope.

Main challenge As discussed in section 4.1, lightweight multi-tenant serverless platforms such as Oracle’s GraalOS [53] and *Cloudflare’s* Workers [9] need to be able to create an execution context inside the trusted hypervisor application that has limited access to the virtual address space of the original process context. This means being able to limit access to the virtual address space for only a *specific thread* running in the process instead of for all of them (as would be the case if `mprotect(2)` were used). This specially isolated thread can then be used to run a user-provided workload without risking corruption of the hypervisor application itself or other workloads (provided that access permissions are set up correctly).

Note that for performance reasons and to be able to communicate with the hypervisor application, it is often desirable to designate certain virtual memory ranges to be shared between the untrusted, user-provided workload and the host application. In this case it should also be possible to set different permissions for the trusted host application and the user-provided workload, e.g. make the virtual memory range in question read-only from the point of view of the user-provided workload. In addition, to minimize memory usage, it might further be desirable to be able to share memory not only between the host application and a user-provided workload but also between two different workloads (e.g.

read-only data used by both workloads).

Unfortunately, the current Linux kernel does not provide the outlined functionality without special hardware support (nor any other Unix based operating system to the best of my knowledge). And even if hardware support is indeed available then not at the scale needed for a lightweight multi-tenant serverless platform (see subsection 5.2 for more details regarding Intel’s MPK [27] and Arm’s POE/PIE [6] and their limitations).

Apart from the lightweight multi-tenant serverless platform, as we saw in the chapters 4, 5, there are also other classes of applications that need to be able to restrict access to the virtual address space for some of their threads: Examples include servers assigning a thread per connection, applications making use of third-party libraries, memory safe applications calling into native libraries etc. However, for *Noodles*’s design and programming model the targeted use case is the lightweight multi-tenant serverless application.

Target Linux kernel based operating systems An additional motivation for this work is that some of the prior related work on intra-process-isolation – though undoubtedly of great value – cannot be applied directly in a real world scenario for the very mundane reason that the prototypes developed are not compatible with operating systems commonly used in production. The lwC paper [45] for instance (see also subsection 8.1.2) develops a novel intra-process-isolation solution and then applies this solution to secure two popular server applications (Apache [3] and nginx [47]) but the operating system modified is FreeBSD [76]. However, according to *W3Techs* data of 2021 [101], only up to 0.3% of publicly available web servers were running FreeBSD at the time (the entire market share of BSD systems was 0.3% of which 97.8% were made up by FreeBSD). To compare: More than 75% of servers were thought to be running a version of the Linux kernel.

The *SpaceJMP* paper [19] on the other hand presented implementations for even more niche operating systems: *DragonFly BSD* [17] and *Barrelfish* [21], a research operating system wholly unlike any Unix based system.

In response to this, a side goal of the *Noodles* project is to explore how and with how much difficulty support for intra-process-isolation in

the form of a new operating system level abstraction can be added to a more mainstream operating system: the Linux kernel.

7.2 Design and programming model

In this section, after having further motivated the need for a novel intra-process-isolation mechanism, specifically one targeted at lightweight multi-tenant serverless platforms, in the previous section, section 7.1, I first describe the use case I envisage for the new mechanism in more detail. This then motivates a list of design requirements, see subsection 7.2.2. Following this, I provide a detailed description of the targeted semantics for the proposed new mechanism, *Noodles*, and discuss the programming model to make use of this new intra-process-isolation mechanism.

7.2.1 Targeted use case

The design of the here proposed intra-process-isolation mechanism, called *Noodles*, targets first and foremost a specific use case (though it may also be of interest for other use cases with similar requirements): The targeted use case is that of the lightweight serverless platform running a great number of workloads from mutually distrusting users. Instead of isolating these workloads each in a separate process or even the more heavyweight virtual machine, the goal is to run all concurrently in the same process, minimizing overhead (this includes both runtime and memory footprint) and thus costs.

Imagine a trusted hypervisor application, implemented and controlled by the platform operator. This application runs either in a single process or potentially controls a small set (in the dozens) of “container” processes to run workloads in. Importantly, the number of workloads greatly outnumber the number of hypervisor application processes.

To this hypervisor application user-defined workloads in the form of simple functions are submitted. Characteristically they run only for a very short amount of time. The hypervisor application must safely run these workloads, safely meaning here that the provided workloads must neither be allowed to compromise the hypervisor itself nor any other workloads.

Note that in this usage model the workloads do not actively share any state among each other (withstanding any read-only or COW sharing set up transparently to the user as an optimization). However, the workloads may share some carefully controlled state with the process containing them, the hypervisor application. This application serves in

a similar function as an operating system would for processes running on top of it: That of the resource arbiter and jailor for the workloads submitted by untrusted users.

Note that another important characteristic of the workloads targeted is that they are stateless: after having run to completion, their state can be removed fully from the platform.

7.2.2 Design requirements

From the usage scenario described in the previous subsection, subsection 7.2.1 I derive a list of requirements for the design of the new intra-process-isolation mechanism:

- **No special hardware requirements:**
First, the intra-process-isolation mechanism must not rely on any specialized hardware or not commonly available hardware feature since it is impractical to replace entire fleets of servers to support a single application. Instead it must be supported by common place, of the shelf server hardware, relying only a standard MMU.
- **Scales to thousands of workloads:**
Since the targeted use case is a use case with potentially thousands of clients submitting their workloads concurrently, it is important that the new mechanism is *lightweight*, meaning in this case less resource intensive than the already existing process creation. Only then will the hypervisor application be able to scale to meet the demand.
- **Provides meaningful isolation:**
On the other hand, the new mechanism must strike a balance between being lightweight and yet still providing *meaningful isolation*. For example, threads are lightweight but do not provide any form of isolation and thus don't fulfill this requirement. In the context of memory focused intra-process-isolation meaningful isolation means being able to restrict access to memory.
- **Allows fine grained access control:**
To be able to isolate different workloads from each other but also, vitally, the hypervisor application, it is important that the new

mechanism allows for effective but also flexible partitioning of the virtual address space and to implement different access policies for different components. It is not enough to simply be able to restrict memory access for the whole virtual address space or all components at once, instead we need to be able to restrict memory access for a chosen virtual address space range for a specific component.

- **Allow sharing of data:** At the same time, for reasons of efficiency, it must be possible for a component to selectively share data with another component, e.g. the hypervisor application must be able to share a specific range of virtual memory with an untrusted workload.
- **Enforce isolation with a trusted mechanism:** Intra-process-isolation, since it is so vital for the security of any such lightweight multi-tenant serverless platform, should ultimately be enforced with a well known and trusted hardware mechanism - the MMU.
- **No hidden costs:** For the purpose of predictable performance (especially with regards to an QoS agreements) any costly setup should be done at the startup of a workload. This means it should be avoided as much as possible to “lazily” copy physical pages such as `fork(2)` normally does which might later on result in unpredictable performance loss (because of page faults suffered and the cost of COW being payed at a later point in time).
- **Run on a common operating system:** Just as the new mechanism shouldn’t rely on specialized or new, not yet on every machine fully supported hardware features, it also should be implementable in a commonly used operating system, e.g. the Linux kernel.
- **Operating system integration:** For ease of use, maintainability and adoptability as well as to avoid using more resources than absolutely necessary, the new mechanism should be integrated into the operating system itself. Resource management, access policy enforcement and providing isolation via virtualization are traditionally the responsibility of the operating system. It also avoids certain overheads often created by moving the implementation of functionality of this type into user space, e.g. having to maintain shadow page tables.

7.2.3 Semantics of *Noodles*

After discussing the requirements for the proposed intra-process-isolation mechanism in the previous subsection 7.2.2, this subsection covers its intended semantics.

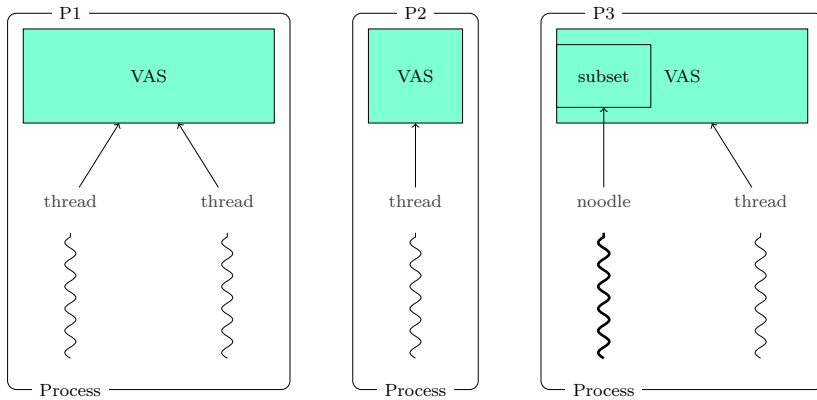


Figure 7.1: Conceptual overview of *Noodles*, inspired by figure 1 in the *Noodles* paper [26] which was created by William Blair

New operating systems primitive *Noodles* proposes a new intra-process-isolation mechanism in the form of an operating system level primitive, called a *noodle*. A *noodle* is a novel abstraction (just like the thread or the process) with support backed into the operating system itself as a design choice. It occupies the middle ground between lightweight but un-isolated threads and isolated but heavyweight processes, combining intra-process-isolation with the ease of use of threads.

Threads with private virtual address spaces The key idea of *Noodles* is to *decouple* process and virtual address space creation. This makes it possible to give a thread its very own, private virtual address space on request: A thread that needs to be isolated in some way from the rest of a process, can be switched to its own private virtual address space at any point in time after its creation. As it switches to its own private

virtual address space, it is transformed into a so called “fat thread” – a *noodle*.

Subset of all virtual address space mappings The principle of least-privilege dictates that an execution context should only have access to the data that is necessary for it to perform its function. *Noodles* implements this by isolating the thread turned *noodle* by giving it access to only a *subset* of the original virtual address space. To be exact, when a thread is turned into a *noodle*, the *noodle*’s private virtual address space contains a *subset* of all the virtual to physical page mappings installed in the original virtual address space.

If a virtual to physical page mapping is installed in both the original virtual address space and the *noodle*’s new virtual address space, in the latter the mapping’s associated permissions must also be a *subset* of the permissions set on the original mapping. An example of this would be to downgrade the permissions on memory mapped into the original virtual address space as read-writeable to read-only. This ensures that access rights of the *noodle* to backing physical memory pages is also limited to a *subset* of the access rights to physical memory pages mapped into the original virtual address space.

Note that it is permissible for both the *noodle* and a thread running in the original virtual address space to install additional mappings or unmap unneeded ones at a later point in time.

Semantics otherwise identical to threads To make *noodles* both intuitive and a possible replacement for threads where intra-process-isolation is desired, the semantics of a *noodle* are kept identical to those of a normal thread – *except* where related to the virtual address space itself and its management.

Conceptual overview of *Noodles* See figure 7.1 for a conceptual overview of how a *noodle* can be understood with relation to the familiar operating systems abstractions thread and process: In the figure there are three different processes depicted, *P1*, *P2* and *P3*. *P1* is a multi-thread process with two threads each sharing the virtual address space (VAS) of *P1*. *P2* by contrast is a single threaded process. Its thread does not share its virtual address space with any other thread. *P3* shows how

noodles fit into this picture: One thread in an originally two-threaded process has been transformed into a *noodle* – a fat thread that via its own private virtual address space has access to a *subset* of the physical pages mapped into the original virtual address space.

The lifecycle of a *noodle* An important factor for both ease of use and adoptability is that the transformation from a normal thread into a *noodle* can be done at any point during a thread’s life time. This makes it possible to have a thread running with full access to the original virtual address space, perform all necessary setup, and then *downgrade* the thread’s access later on, e.g. just before starting the untrusted workload. Once the thread turned *noodle* exits, its private virtual address space and all the resources that are normally released on thread exit, are torn down.

Note that the transformation from a thread to a *noodle* is a one-way transformation: It is only allowed to happen in this direction, undoing it is not supported.

Focus on in-memory data isolation In accordance with focusing on memory isolation in this thesis, the isolation provided by a *noodle* is limited to that gained by either restricting or removing altogether some of the virtual to physical page mappings that form the original virtual address space. This allows to either share physical memory pages mapped into the original virtual address space (if a mapping is kept unchanged) or reduce the *noodle*’s permissions (e.g. downgrade from read-write to read-only). Alternatively we can remove access to certain physical memory pages altogether by not including the corresponding mapping in the *noodle*’s private virtual address space.

Any other state shared by threads – such as file descriptors – is still shared with the remaining threads after one of them has been turned into a *noodle*. The *noodle* also retains the same pid, etc. as in every aspect apart from memory access a *noodle* behaves just as the original thread would have.

Orthogonal to hardware extensions Note that the intended semantics of *Noodles* do not stand at odds with either Intel’s MPK [27] nor Arm’s PIE/POE [6] (see subsection 5.2 for more details on these

hardware extensions). On the contrary, by virtue of having its own private virtual address space, each *noodle* gets its own contingent of memory keys/ protection domains (which are limited to 16 on both Intel and Arm hardware).

Stronger isolation than that provided by MPK and POE/PIE

Note that the memory isolation *Noodles* can provide by not including a virtual to physical page mapping in the *noodle*'s private virtual address space is stronger than that of merely marking a region as inaccessible with the help of MPK or PIE/POE: Due to a physical to virtual page mapping not being present in a virtual address space in the first place, even a successful bit flip by an adversary exploiting e.g. Rowhammer [34] would not be enough to gain access to the physical memory page in question.

Virtual address spaces can be changed independently If a physical to virtual page mapping is removed in either the *noodle*'s private or the original virtual address space *after* a *noodle* has already been created (assuming the mapping is present in both virtual address spaces before being removed), this will not affect the other virtual address space. The same is true if there is a change of permissions set on a mapping. Note though that permissions set on mappings a *noodle* inherited from the original virtual address space may only be downgraded.

7.2.4 Programming model

In this subsection, after describing the semantics of a *noodle* in the previous subsection, subsection 7.2.3, I present the envisioned programming model for *Noodles*. The goal of the model is to efficiently employ *Noodles* to achieve intra-process-isolation in a generic multithreaded application with mutually distrusting components being managed by a trusted host component.

Generic scenario Imagine a scenario where multiple different execution contexts – corresponding to different components of an application – are part of the same process, e.g. a lightweight multi-tenant serverless platform as describe in a earlier subsection, subsection 7.2.1. Crucially,

these execution contexts are not only mutually distrusting (we refer to this as the vertical boundaries of trust) but also untrusted by the application containing them (the host execution context). Though the host execution context itself is trusted, between it and the untrusted execution contexts we have what we refer to as a horizontal boundary of trust.

Example instances of such a scenario include our targeted lightweight multi-tenant serverless platform use case but are not limited to it: A server maintaining connections (managed by different threads) to multiple third-parties at the same time, a shared storage server running untrusted analytics workloads close to the data or a multi-tenant (relational or otherwise) database are other possible examples.

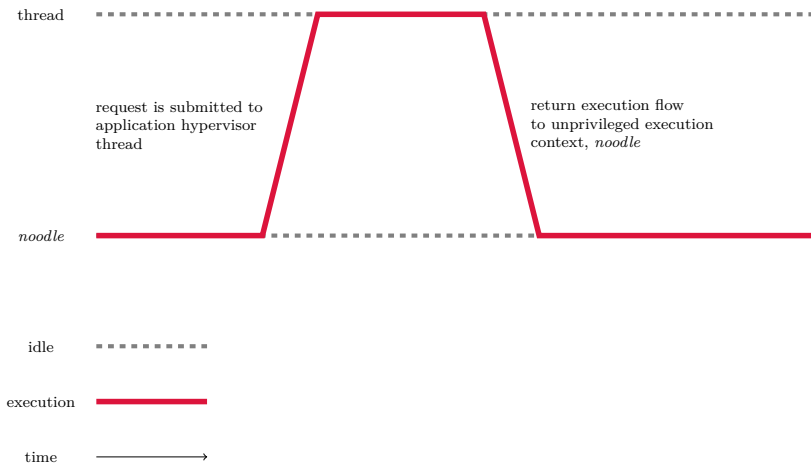


Figure 7.2: Visualization of the programming model, inspired by figure 3 in the *Noodles* paper [26], created by Matthias Neugschwandtner

A rendezvous based programming model Since *noodles* can, once threads have been transformed into them, never be turned back into normal threads, only a rendezvous based programming model is suitable for *Noodles*. A visualization of such a model can be seen in figure 7.2.

The alternative would have been a model where a thread so called

“tunnels through”: The thread would change its state from that of an untrusted execution context to that of a trusted one and back again, potentially multiple times during its life cycle. This would mean executing both trusted host and untrusted client code on the same thread in turn, switching as many times as necessary between the two different states. Implementing this would have required being able to safely both restrict the virtual address space (narrowing access) as well as being able to later *regain* any lost privileges (widening access).

As any mechanism allowing an untrusted execution context to become trusted and thus acquire additional privileges needs to be handled with extreme care, to simplify both design and implementation, *Noodles* only supports the restriction of access for an execution context. As this means that turning a thread into a *noodle* is an operation which cannot later be undone again, the model just described is not a good fit.

Instead, to take advantage of the fact that once an execution context’s access has been restricted this is *irreversible*, the ideal programming model for *Noodles* is a rendezvous based one: With this model the trusted host code (e.g. part of a hypervisor application) executes on dedicated, separate threads. Untrusted code on the other hand is executed by threads that have been turned into *noodles*. Those only have access to a subset of the original virtual address space.

Whenever the *noodles* have to request data not per default accessible to them or need to execute a privileged operation they are not authorized to perform themselves, they will temporarily “pass” execution to the unrestricted, trusted execution context. The trusted context will then perform the operation on their behalf or pass requested data back to them. Data needed to perform the operation in question – such as arguments for a function call – will be placed in a shared, specially designated virtual memory region before signaling the trusted execution context (the same holds for returning data).

In practice this could be implemented by e.g. having a trusted execution context (thread) spin on a queue containing requests that have been submitted by code running in a *noodle*. Each time untrusted code needs to complete an operation for which it lacks sufficient privileges, it would submit requests to the trusted code’s queue and either wait (synchronous communication) or potentially check back later if the request has been processed (asynchronous communication).

Generally, after a “rendezvous”, a point in time when a handover occurs, the trusted execution context will start executing the code to process the request with the passed data as arguments (if there is any). Afterwards, the *noodle* executing the untrusted code will then pick up any data resulting from the processed request and resume execution.

To understand how this works, see also figure 7.2 where we have a privileged thread representing the trusted execution context and a *noodle* representing the restricted execution context running untrusted code. Note that time flows from left to right and that the crimson line represents the flow of execution. At some point in time the *noodle* submits a request to the trusted execution context which then causes the execution flow to switch to the privileged execution context at the top. After the request has been processed, the execution flow is again passed back to the unprivileged *noodle*.

Note that how fast and efficiently the execution flow can be passed from the untrusted execution context to the trusted one above it is entirely dependent on how the actual handover, the rendezvous, is implemented.

Conclusion The programming model just described allows to transform a thread into a *noodle* right at the beginning, after having performed any necessary set-up for the client and before actually starting to run the untrusted code. It then retains only access to a subset of the original virtual address space. To ensure intra-process-isolation, only essential data should be shared between the *noodle* and the trusted execution context. This can be accomplished by storing it in designated virtual memory regions (virtual to physical page mappings) that are part of both virtual address spaces (though potentially with a subset of the permissions in the *noodle*’s case).

7.2.5 Managing page tables for *Noodles*

The design of *Noodles* deliberately does not specify how to manage the page tables giving rise to both the original virtual address space of a process and the new virtual address space private to the *noodle*. This is because there is more than one way to implement *Noodles* and which is best depends to a great degree on how the memory management system

of a concrete operating system is structured. *Noodles* targets the Linux kernel but the design itself as presented in this section is not Linux kernel specific and could feasibly be implemented in another operating system.

The crucial choice when implementing *Noodles* comes down to how much of a *noodle's* page tables are shared with the original virtual address space. Depending on how many physical pages need to be mapped into both the *noodle's* private virtual address space and the original virtual address space, sharing page tables (using COW semantics in case of permission changes etc.) could yield significant benefits.

However, as we will see in the next section, section 7.3, not all operating systems support sharing of physical to virtual page mappings in general and sharing page tables in particular. This is partly due to different abstractions used to manage virtual address spaces, e.g. FreeBSD [76] uses different abstractions than the Linux kernel does.

In operating systems without support for the sharing of physical to virtual page mappings and page tables, we have to fall back copying page tables, potentially taking a performance hit.

7.3 Linux kernel background

This section is all about the inner workings of the Linux kernel's memory management system and how it implements certain central system calls such as `fork(2)` [38], `clone(2)` [37], `mmap(2)` [39] etc. The aim is to look beyond the descriptions and guarantees given in the manual pages and to instead inspect the concrete data structures involved in managing a process's virtual address space, such as e.g. the `vm_area_struct` [87], the `mm_struct` [83] etc.

As part of that we will also examine in detail how mappings installed in a process's virtual address space by `mmap(2)` are represented internally by the Linux kernel and how they are affected by the crucial system calls `fork(2)` and `clone(2)`. This will then form the foundation for the next section, section 7.4, describing in detail how the prototype was implemented.

The Linux kernel is developed as an open-source project by the Linux kernel development community which publishes regular releases under the terms of the GNU GPL version 2 license [88]. These regular releases, which can be found on www.kernel.org [88], are referred to either as *mainline*, *upstream* or *vanilla* Linux kernels.

Note that as the *Noodles* prototype described in the next section, section 7.4 targets the Linux kernel release 5.15.116, this background section always refers to this specific version unless stated otherwise.

7.3.1 Focus: the memory management system

This thesis focuses on one of the most important parts of the Linux kernel, which is the memory management system, often referred to as *mm* for short. Within the memory management system the emphasis will be on the management of the virtual address space belonging to a process as opposed to the management of physical RAM (though of course both of these two topics are tightly interwoven).

To understand how the Linux kernel manages a process's virtual address space, one must first understand its memory management system's main data structures and the abstractions these data structures are based on. Here in this thesis I will focus on the `mm_struct` [83] representing the virtual address space of a process and the `vm_area_struct` [87] representing a mapping or region in this virtual address space. I will also

touch on the `task_struct` [84] which isn't just central to the memory system but the whole Linux kernel.

Note that many of Linux kernel's data structures have been present in it for decades or even since the very beginning. Inevitably, over the years the data structures' often originally quite simple and straightforward designs have been obscured by the addition of new features, the handling of edge cases and hardware quirks, performance optimization etc. To give an example, the `task_struct` can be found in the Linux kernel's very first version, version 0.01 [85]. Since then its definition has grown greatly in size: It has ballooned from a mere 29 lines of code to over 700 lines in version 5.15.116!

As an added complication, there is sometimes little documentation available that has been kept up to date; often the only descriptions to be found when searching for documentation for specific components are years, sometimes even decades out of date. All of this means that understanding how the Linux kernel's code functions can sometimes amount to a little bit of detective work, on occasion even requiring going back in time, carefully studying old versions, commit messages, additions and deletions leading up to the current state.

Back to memory management and how it is implemented in the Linux kernel: The files most relevant to the memory management system can be found in the directories `include/linux` (for the header files such as `include/linux/mm.h` and `include/linux/mm_types.h`) and `mm` [14]. A quick glance at both these directories will reveal to the interested reader that over the years the Linux kernel's memory management system has grown to sprawling size. Complete coverage of every aspect of the memory management system is therefore out of scope for this thesis, instead the aim here is to give the reader the necessary background information as it pertains to the use case we are interested in here: Lightweight intra-process-isolation.

7.3.2 Main data structures

The most important data structures used in the Linux kernel's memory management system are the three following: The struct representing the task, called `task_struct`, the struct representing the virtual address space, called `mm_struct`, and the struct representing a mapping (of virtual to physical pages), called `vm_area_struct` (short for virtual

memory area).

I will discuss each of these briefly in the following paragraphs.

Tasks not processes

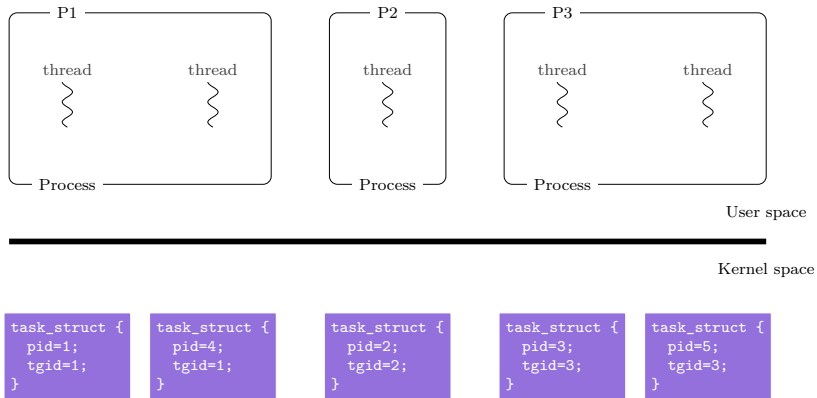


Figure 7.3: The Linux kernel's concurrency model

The crucial `task_struct` is both the Linux kernel's unit of scheduling and the abstraction it uses to represent an execution context, not the well-known *process* or, as an alternative, the *thread* as one might otherwise suspect. In fact, the abstraction of the process, which is so prevalent in user space, is not directly represented by any data structure in the Linux kernel at all. Instead it represents each thread running in user space - no matter if they are the implicitly created thread of a single-threaded process or an explicitly created thread part of a multi-threaded process - as a unique `task_struct` in kernel space. Processes are consequently represented only *implicitly*, as group of `task_structs`.

To make our discussion of the `task_struct` more concrete, here are two examples illustrating this: If a single-threaded process, *P2*, is created, it will be represented by a single `task_struct` in kernel space while a process *P1* with 2 threads will be represented by 2 different `task_structs` in kernel space. Note that just as a process with 2 threads is represented by 2 `task_structs` so are 2 single-threaded processes.

Figure 7.3 gives a schematic overview of the Linux kernel’s concurrency model and how it maps to user space.

The kernel space is represented by the lower half of figure 7.3, the user space by the upper half. The rectangles at the bottom of the diagram each represent a `task_struct` present in kernel space and encapsulating the state of an execution context. In the middle of the top half we have process *P2*, while the box with the rounded corners to its left refers to process *P1* (as seen by user space).

What ties together the two `task_structs` representing process *P1* as opposed to two `task_structs` part of two different processes is the kernel space’s Thread Group ID (`tgid`) which in user space shows up as the Process ID (`pid`) [33]. (Note that the `task_struct`’s field called `pid` will show up as the Thread ID (`tid`) in user space. Also note that for the *thread group leader* `tgid` equals `pid` holds.)

It can justifiably be said that the `task_struct` is one of the most important data types in the whole Linux kernel. It is defined in the file `include/linux/sched.h` [84] and has, as mentioned above, been part of the Linux kernel since its very first release. Since then it has grown too sprawling to be reproduced here in full, as it spawns a staggering 787 lines in version 5.15.116, but the for us most relevant fields are as follows:

As figure 7.4 shows, the `task_struct` has a field called `mm`. This is a pointer to another crucial data structure we will discuss in the very next paragraph, 7.3.2, the `mm_struct` [83] representing a whole process’s virtual address space.

Note also the fields `pid` and `tgid` which we have already encountered in figure 7.3. The inconsistency in their naming (compared to the names under which these same values show up in user space) is due to the current Linux kernel being the result of continuous development taking place over many years.

As an aside, it is worth mentioning here for clarity’s sake that when referring to a *thread*, I’m always referring to what is sometimes called a *kernel thread* when discussing *N:M* scheduling schemes or *platform thread* in some contexts. These terms are used for a thread that the kernel itself is aware of and therefore can preempt and schedule as opposed to a pure userland thread (also called *green thread* sometimes). I will not use the term *kernel thread* here as in the context of the Linux kernel

```

1  struct task_struct {
2
3      /* ... */
4
5      /* represents the virtual address space */
6      struct mm_struct *mm;
7
8      /* ... */
9
10     pid_t pid; /* shows up as tid in user space */
11     pid_t tgid; /* shows up as pid in user space */
12
13     /* ... */
14
15 };

```

Figure 7.4: Linux kernel version 5.15.116, *include/linux/sched.h*, line 77 [84], additional comments by the author

(and following this, also in this thesis), the term *kernel thread* is instead used to refer to something else entirely: A thread which runs purely in kernel space, usually to perform some sort of background work, e.g. *kswapd*, the kernel swap daemon.

Virtual address spaces and their management in the Linux kernel

Another central data structure of the Linux kernel is the `mm_struct` [83], defined in the file *include/linux/mm_types.h*. It represents the virtual address space of a user space process in the Linux kernel and is used by the memory management system to manage it. As we have seen in figure 7.4, showing a shortened version of the definition of the `task_struct`, all `task_structs` have a pointer to an `mm_struct`.

See figure 7.5 for a version of figure 7.3 that also shows the corresponding `mm_structs`. Observe that the number of `mm_structs` is equal to the number of processes as seen from user space and that `task_structs` representing user space threads running in the same process share their `mm_struct`. This reflects the fact that threads part of

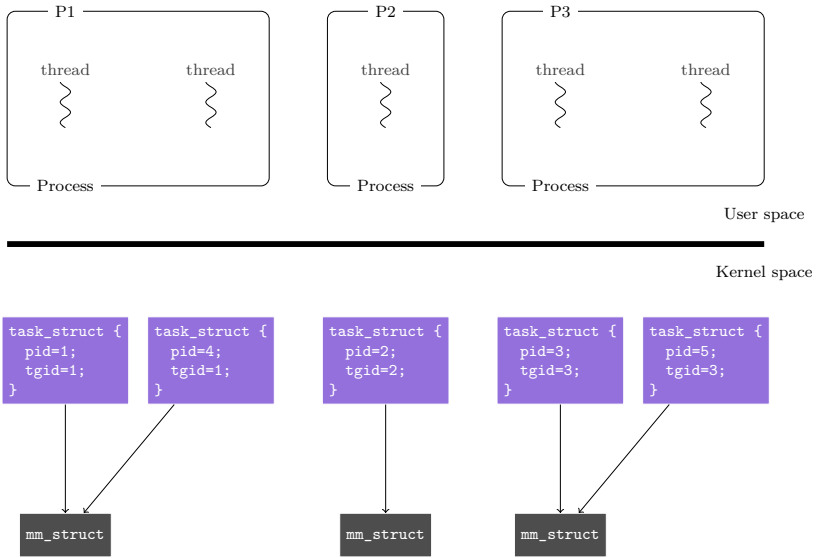


Figure 7.5: The Linux kernel’s concurrency model, also showing the `mm_struct`s representing the virtual address spaces

the same process share their virtual address space.

Figure 7.6 shows the (shortened) definition of the `mm_struct`. Note that the first part of the definition is wrapped in an inner struct to randomize the layout of these fields.

The field called `pgd` on line 7 points to the root of the page tables, the Page Global Directory (`pgd`), which we will see in more detail in subsection 7.3.3 further below.

The fields defined on the lines 19 – 21 are address pointers giving the start and end of specific sections of the virtual address space. To give an example, `start_code` and `end_code` can be used to find the code segment of a process.

Field `mmap_lock` corresponds to the first field called `mmap` and is used to synchronize access to it. `mmap` is a pointer to a linked list of virtual address space ranges, which are covered next. `mmap_lock` must be taken in either read or write mode to read or even modify this linked

list, which is significant for page fault handling (among other things).

```
1 struct mm_struct {
2     struct {
3         struct vm_area_struct *mmap; /* list of VMAs */
4
5         /* ... */
6
7         pgd_t * pgd;
8
9         /* ... */
10
11        int map_count; /* number of VMAs */
12
13        /* ... */
14
15        struct rw_semaphore mmap_lock;
16
17        /* ... */
18
19        unsigned long start_code, end_code, start_data, end_data;
20        unsigned long start_brk, brk, start_stack;
21        unsigned long arg_start, arg_end, env_start, env_end;
22
23        /* ... */
24
25    } __randomize_layout;
26
27    /* ... */
28
29 };
30
```

Figure 7.6: Linux kernel version 5.15.116,
include/linux/mm_types.h, line 402 [83]

A linked list of mappings

While the page tables of a process define its virtual address space for the hardware Linux is running on (specifically, the MMU), the linked list called *mmap* is how the kernel itself manages a virtual address space: Each struct part of this linked list corresponds to a contiguous range of virtual addresses in the virtual address space. That's why the struct is named, fittingly, *vm_area_struct* (or just VMA for short).

They are also referred to as *mappings* since each time `mmap(2)` is called to either map a file into the virtual address space or an anonymous region of memory, internally a *vm_area_struct* is created¹. In consequence, the VMAs forming *mmap* only cover the user's part of the virtual address space and only those parts that have already been mapped.

A VMA covering a certain range of the virtual address space being present in the *mmap* list does not, however, guarantee that any physical memory pages have actually been allocated: Whenever a page fault occurs, the VMA the virtual address falls into is instead consulted to decide how the page fault should be handled.

Note that while the *mmap* list is conceptually important, since scanning a linked list comes with a non-negligible cost, all *vm_area_struct*s are also part of a red-black-tree. This enables faster look-ups.

Figure 7.7 gives the (shortened) definition of the *vm_area_struct*: The fields *vm_start* and *vm_end* gives the range of virtual addresses the VMA covers. The fields *vm_next* and *vm_prev* can be used to traverse the linked list connecting all VMAs belonging to the same virtual address space. *vm_mm* is a back-pointer to the *mm_struct* representing this virtual address space. The next two fields, *vm_page_prot* and *vm_flags* refer to what type of mapping the VMA represents (*shared* or *private*, *file-backed* or *anonymous*) and how it may be accessed. *vm_ops* bundles together several function pointers that determine (among other things) how a page fault in the range of the VMA will be handled. Finally, the last two fields given, *vm_pgoff* and *vm_file* are only meaningful if the mapping is file-backed.

¹There are exceptions to this, e.g. when a new mapping is instead merged with a pre-existing, adjacent mapping with the same permissions.

```

1  struct vm_area_struct {
2
3     /* ... */
4
5     unsigned long vm_start; /* Our start address within vm_mm. */
6     unsigned long vm_end; /* The first byte after our end address
7                            within vm_mm. */
8
9     /* linked list of VM areas per task, sorted by address */
10    struct vm_area_struct *vm_next, *vm_prev;
11
12    /* ... */
13
14    struct mm_struct *vm_mm; /* The address space we belong to. */
15
16    /* Access permissions of this VMA. [...] */
17    pgprot_t vm_page_prot;
18    unsigned long vm_flags; /* Flags, see mm.h. */
19
20    /* ... */
21
22    /* Function pointers to deal with this struct. */
23    const struct vm_operations_struct *vm_ops;
24
25    /* Information about our backing store: */
26    unsigned long vm_pgoff; /* Offset (within vm_file) in PAGE_SIZE
27                            units */
28    struct file * vm_file; /* File we map to (can be NULL). */
29
30    /* ... */
31
32 } __randomize_layout;
33

```

Figure 7.7: Linux kernel version 5.15.116, *include/linux/mm_types.h*, line 319 [87]

7.3.3 Internal workings of fork

In this subsection we continue our exploration of the Linux kernel’s memory management system by examining the Linux kernel’s implementation

of what can justifiably be called *Unix*'s defining system call: `fork(2)`.

Forking a process – essentially dividing the program execution flow into two streams with a shared history – instead of creating a new process from scratch is how all *Unix* derived operating systems and indeed also the Linux kernel create all new processes. This implies that the implementation of `fork(2)` is an integral part of how the Linux kernel functions.

Before we can talk about `fork(2)` we need to clarify to which of its different incarnations we are referring: Most of the time when we call system calls we don't actually invoke them directly but instead call a library's wrapper function for reasons of ease and convenience which then does the heavy lifting – the system call invocation – for us. `glibc` offers one such wrapper for forking a process, called `fork`, however, behind the scenes this no longer actually calls the Linux kernel's system call also named `fork`, `sys_fork`, but instead invokes its much more general system call called `sys_clone`. To maintain backwards compatibility, `glibc`'s wrapper calls the Linux kernel's `sys_clone` with a specific set of flags to instruct it to behave just like the older, traditional `sys_fork` would have done had it been called instead. This can be verified on `fork(2)`'s own man page [38] or by using the `strace` utility. Tracing a simple program executing `fork(2)` will print something like the following snippet to the console:

```
clone(  
    child_stack=NULL,  
    flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD,  
    child_tidptr=0xe0354f23df30  
) = 96673
```

(96673 being the created child's PID as seen from user space.)

The Linux kernel's actual system call `sys_fork` on the other hand (which does still exist and can be called by passing its system call number to `syscall(2)`) is simply one more way to call the Linux kernel's `kernel_clone` function. This function is ultimately called by all system calls creating a new process: That includes `sys_clone`, its newer variant `sys_clone3` and, as just stated, `sys_fork` but also `sys_vfork`.

All of this is to say that to study the Linux kernel's implementation of `fork(2)`, the function that needs to be examined is actually `kernel_clone` [81]. This function, due to its many possible sets of flags,

serves simultaneously as implementation for a whole range of system calls: `fork(2)`, `clone(2)`, `clone3(2)` and `vfork(2)`.

Despite all of these nuances, the semantics of creating a new process remain the same: Forking creates a *child process* by first *duplicating* the calling process, termed the *parent process*. Following the duplication, the newly created process is then put on the run queue before we return from the system call and control is passed back to user space.

Since copying all of the caller's resources (all physical memory backing its virtual address space, its page tables, its registers and its kernel state) is prohibitively expensive, mechanisms were developed to alleviate this cost: One of the mechanism is Copy-On-Write (COW). It exploits the fact that much of the data stored in the parent's virtual address space before forking might never be accessed by the child, e.g. because the call to `fork(2)` is immediately followed by a call to `execve(2)` in the child. As a short reminder, using COW, the physical memory pages backing the private parts of the parent's virtual address space, will be marked as read-only instead of copied. They will only be duplicated once either the parent or the child attempts to modify them.

Another mechanism invented to alleviate the cost of fork is its variation called `vfork(2)` [40], which can be thought of as a sort of poor man's fork. The idea here is that instead of duplicating all resources of the parent, they are temporarily *loaned* to the child while the parent is paused. The parent will only resume execution once the child has either terminated (normally or abnormally) or executed `execve(2)`. Since the child borrowing the parent's resources in such a way comes with certain restrictions for the child (e.g. the child may not exit the function), COW is the more popular of these two mechanisms.

However, on Linux when creating a child by calling `fork(2)` COW is only used for the physical pages mapped *into* a process's virtual address space, it is not used for the actual pages tables *themselves*. This means, that whenever a child is created by forking, the parent's entire page tables are duplicated in full.

Over the years there were several attempts [16, 31, 15], [107] to introduce COW for the page tables themselves but those were never upstreamed into the mainline of the Linux kernel to the best of my knowledge. It was argued that the additional complexity introduced wasn't worth the gained performance in specific cases [11].

In 2005 an optimization [31] to address the cost of forking was indeed added to the Linux kernel but it concerns only *shared mappings* and does not involve COW or truly sharing page tables: Instead of copying the page tables mapping e.g. a file (as shared) into a virtual address space, copying them for the child process is simply skipped during forking and the page tables are re-built when the child page faults. Unfortunately this only works for specific cases (cases in which it is certain that rebuilding the page tables will lead to the same result). It might also lead to performance degradation at a later point as the page tables will still be copied if the child accesses the mapping in question.

The consequence of all this is that as the number of mappings in the virtual address space of a process and with it the amount of memory needed to store its page tables grows, so does the cost to fork that process. (See the upcoming section 7.5 for benchmark results illustrating this.)

The effect of `fork(2)` on page tables can be visualized as follows:

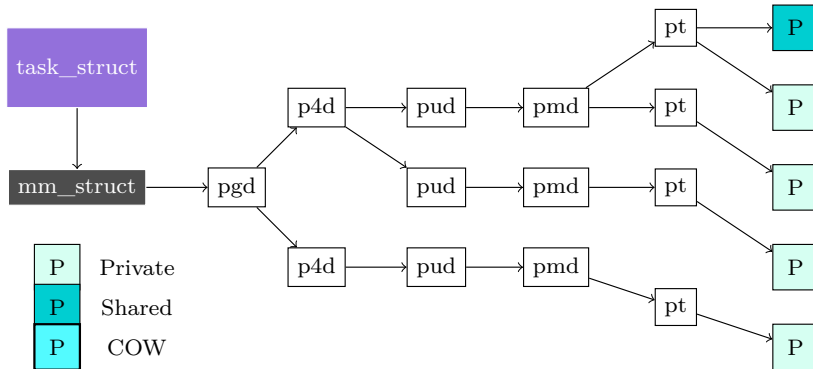


Figure 7.8: Page tables as represented internally by the Linux kernel before any forking takes place.

Figure 7.8 shows a simplified representation of the page table tree backing a single threaded process's virtual address space (as represented by the Linux kernel internally). At the top left, the `task_struct` representing an executable context in kernel space can be seen. It contains a pointer called `mm` which gives the location of the `mm_struct` represent-

ing its virtual address space. This struct in turn has a field pointing to the root of the page table tree: the Page Global Directory (pgd).

At this point I want to emphasize that the Linux kernel always represents a process's page table tree like shown in figure 7.8, even when running on CPU architectures that do not structure their page tables like this. An example would be that not all architectures even have 5 levels as shown (the 5th level was introduced to the Linux kernel in version 4.11-rc2 [32]). This is made possible by some clever compiler magic and architecture dependent implementations of specific functions.

The level below the pgd is the level containing the p4d (p4d)s, which are pointed to by the entries of the pgd. The entries of the p4ds in turn point to the Page Upper Directory (pud)s whose entries point to the Page Middle Directory (pmd)s. Their entries, finally, point to the actual Page Table (pt)s containing the Page Table Entry (pte)s. The ptes specify the physical pages backing a process's virtual address space – the Linux kernel refers to them commonly as *frames*.

In figures 7.8 the frames or physical pages are represented by the boxes in different shades of blue containing a *P*.

If we now fork the process we've been studying, we will get what figure 7.9 demonstrates schematically:

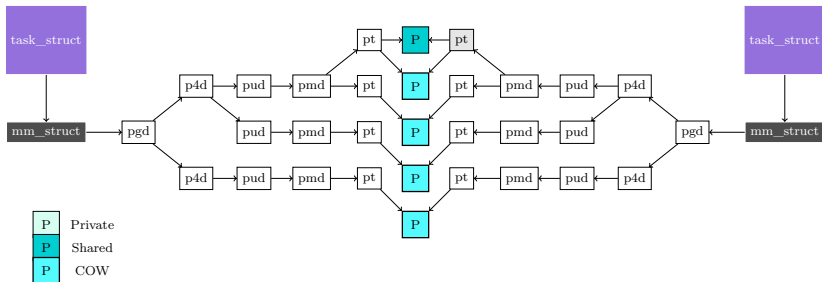


Figure 7.9: After executing `fork(2)`

A new `task_struct` has been created as well as a new `mm_struct` to represent the also single threaded child process and its virtual address space. This makes sense since we created a fully fledged new process and not just a new thread, sharing its virtual address space with its siblings part of the same process. The physical pages themselves have not been

duplicated. Instead, provided they are not part of a *shared* mapping as e.g. the frame at the top of figure 7.9 is, they have been marked as *COW* and will only be duplicated if at a later point either the parent or the newly created child happens to perform a write to the frame. As already mentioned, the same hasn't happened to the page table tree itself: Every single level of the page table tree has been duplicated in full, requiring the Linux kernel to copy the physical memory holding the page table tree ².

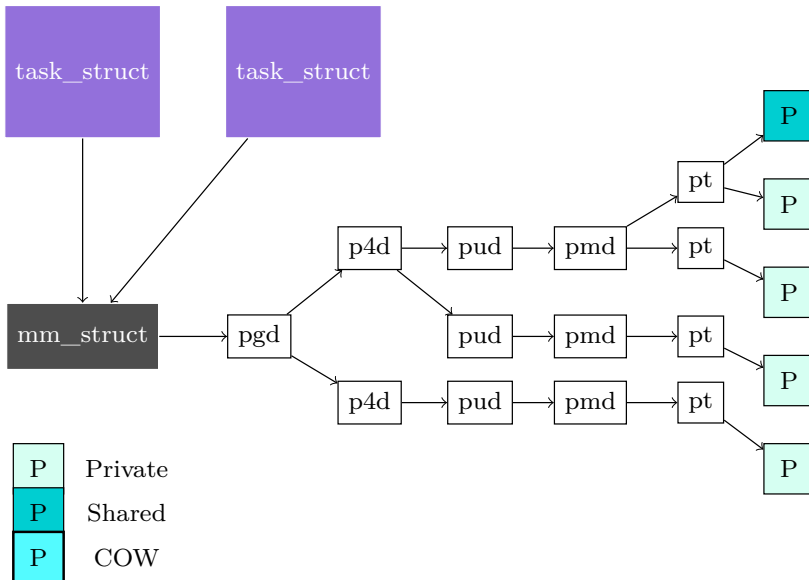


Figure 7.10: State of the page tables after creating a new thread

To conclude this subsection and for comparison, observe in figure 7.10 how the kernel's internal data structures and the page tables are affected if we had created a thread instead of forking our process: Notice that not only have the page tables themselves not been duplicated but that the new `task_struct` shares its `mm_struct` with the original `task_struct`.

²Caveat: An exception to this are the page tables corresponding to shared mappings as discussed earlier, notice the greyed out `pt`.

7.3.4 Mapping types and their internal handling

When discussing mappings and how they are handled by the Linux kernel, it is important to be aware that even though all mappings are represented by `vm_area_structs` internally, there are 4 different mapping types. This type depends entirely on the flags passed to `mmap(2)` when a mapping is originally installed (by calling `mmap(2)`) and it determines how the Linux kernel manages a `vm_area_struct`.

A mapping can be either *private* or *shared* and it may be installed either as an anonymous or a file backed mapping. This leaves us with 4 different possible combinations: *File backed shared*, *anonymous shared*, *file backed private* and *anonymous private*.

See table 7.1 for an overview of the different mapping types and their semantics:

Mapping types [39]

File backed shared	<p>This type of mapping is used to map a region of a file, starting at a page boundary aligned offset, into a process's virtual address space.</p> <p>This region can be mapped with different permissions (any combination of the flags <code>PROT_READ</code>, <code>PROT_WRITE</code> and <code>PROT_EXEC</code> or <code>PROT_NONE</code> that is a subset of the permissions the caller has opened the corresponding file descriptor with is allowed).</p> <p>Crucially, if the resulting mapping is later written to, any modifications will be visible to any other process mapping the same region and will be carried through to the backing file, no matter if it exists in RAM only or if it is stored on some kind of device, e.g. a disk.</p> <p>Flags: <code>MMAP_SHARED</code></p>
File backed private	<p>In contrast to <i>file backed shared</i> this allows a process to have their own private copy of a region of a file. This region is then mapped into the process's virtual address space under the same conditions as detailed above. However, modifications made to the region are not reflected back to the original file and are also not visible to any other process that may have the same file region mapped in their virtual address space at the same time. Examples of such mappings are for instance the mappings created for the segments of the ELF binary itself.</p> <p>Flags: <code>MMAP_PRIVATE</code></p>

Table 7.1: An overview of the different mapping types and their semantics, part 1, file backed mappings

Anonymous shared	<p>This type of mapping behaves the same as the <i>file backed shared</i> mappings, except that the region of memory mapped into the virtual address space does not correspond to any file. Instead a zeroed out block of <i>anonymous</i> memory of sufficient length is mapped into the caller's virtual address space.</p> <p>Any data stored in this region can, depending on the permissions with which a process has mapped the region into its virtual address space, be accessed and modified by all processes that are sharing the region. However, all data stored in the region will be lost once the last process sharing the mapping unmaps it from its virtual address space since it is not file backed and therefore not persistent.</p> <p>Flags: MMAP_SHARED MMAP_ANONYMOUS</p>
Anonymous private	<p>Finally, this forth type of mapping also provides a caller with a region of zeroed out, non-file backed memory of sufficient length. However, in contrast to <i>anonymous shared</i> mappings the region can neither be accessed by any other process nor can it even be mapped into any other virtual address space.</p> <p>The data stored in such a mapping is also not persistent.</p> <p>Examples of such mappings are the virtual address space regions containing a process's stack and heap.</p> <p>Flags: MMAP_PRIVATE MMAP_ANONYMOUS</p>

Table 7.2: An overview of the different mapping types and their semantics, part 2, anonymous mappings

As we have seen when studying the figure 7.8 and comparing it to figure 7.9, not all mappings, respectively `vm_area_structs`, are treated the same when forking a process: All shared mappings (both those anonymous and backed by a file) will neither be duplicated nor be marked as COW when forking. Instead their backing physical memory (which is either anonymous memory or memory caching a file's pages) will be mapped into both the parent's and the child's virtual address spaces, leading them to share the backing physical pages.

At this point, it would be natural to assume that internally private anonymous mappings and shared anonymous mappings are handled by the same code paths inside the Linux kernel – at least up to a point – however this is not the case: The type of the shared anonymous mapping is instead a special case of the shared, file-backed mapping and is handled differently than private anonymous mappings.

This is made possible by having the Linux kernel mount a RAM-only filesystem transparently to the user, storing any anonymous shared mapping inside this special filesystem. The filesystem in question is called *tmpfs* and can be exposed to the user with the right kernel configuration though it is always mounted internally, even if the corresponding configuration option, `CONFIG_TMPFS`, is not set [86]. In this case *tmpfs* is a *kernel internal* mount.

In effect this means that there are three types of file backed mappings supported by the Linux kernel, private and shared file backed as well as shared anonymous mappings. This allows the Linux kernel to reuse the code paths handling file backed mappings for the case of the shared anonymous mapping. In fact, the manual entry for `mmap(2)` [39] reveals that “the odd one out”, the case of the private anonymous mapping, is only supported since version 2.4 of the Linux kernel.

To summarize this subsection, contrary to appearances, the cases of the private anonymous mapping and the shared anonymous mapping are not handled symmetrically by the Linux kernel. Instead, three of the four mapping types are file backed mapping types, while the private anonymous mapping is a special case.

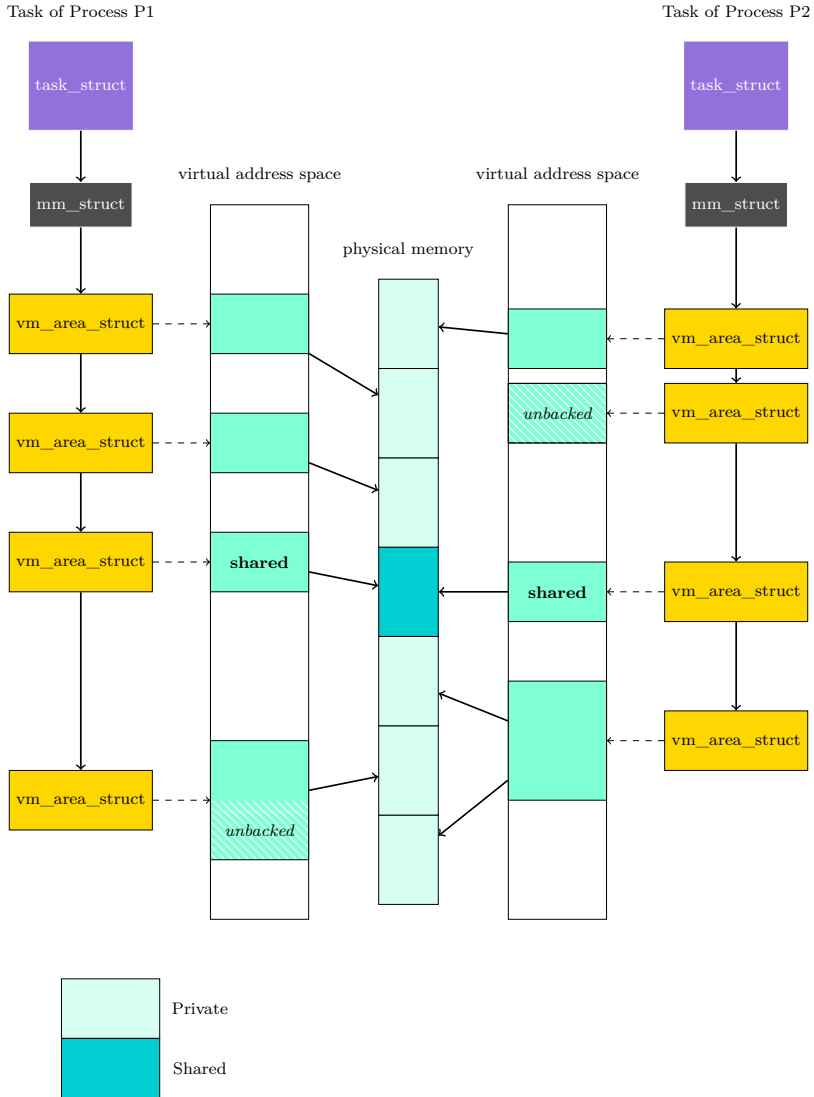


Figure 7.11: Mappings, even for shared memory are defined separately for each virtual address space

7.3.5 No shared mappings

In this subsection, to follow up on the previous subsection 7.3.4, we will finally have a closer look at one of the most important consequences of the memory management system's design: The inability to truly share virtual address space ranges or regions between two different virtual address spaces instead of just sharing the backing physical memory.

Imagine we have two user space processes running on the same machine, on top of the same operating system, each of them single threaded. They each have their own, separate virtual address space as provided per default by the Linux kernel, mapping their own private memory, except for a single mapping with a size of 4KB (one page). This mapping is said to be *shared* between the two processes (for our discussion here it is irrelevant if it is backed by a normal file or if it is an anonymous shared mapping as in this case it will simply be backed by a file in *tmpfs*).

See figure 7.11 for a schematic representation of the two processes. Each has four distinct mappings present in their virtual address space though only one of these mappings is shared between them. The three other mappings are private. In the middle of figure 7.11, the system's RAM is represented as an array of physical pages.

For the three *private* mappings, the physical memory mapped is only ever mapped into one of the two virtual address spaces. In contrast, for the shared mapping, the same physical memory is mapped into both virtual address spaces. Note that even though the same physical memory is mapped (via each process's page tables) for the shared mapping, each process has its *own* `vm_area_struct` governing the shared mapping. Since they belong to two separate virtual address spaces, the two different `vm_area_structs` are not part of the same `mmap` list (accessible via the `mm_struct` representing the respective virtual address space). The locks synchronizing access and modification of those two linked lists is also located on their respective `mm_structs` (field `mmap_lock`, see subsection 7.3.2, figure 7.6). In consequence, these two `vm_area_structs` can be accessed and changed *independently* of each other. This includes the page tables that they govern, which are also not shared between the two virtual address spaces but defined separately.

Another important point to note here is that not each mapping present in a process's virtual address space has to be backed by physical memory just yet. A mapping's physical pages will often be allocated

lazily, on the first access, which then triggers a page fault. It is even possible (see the 8KB mapping part of the left virtual address space, right at the bottom) for a mapping to be only *partially* paged in at a moment in time.

When a page fault is triggered on access to a virtual address that is part of a range represented by a `vm_area_struct`, after taking the `mmap_lock` lock located on the associated `mm_struct`, the page fault is handled by the handler function belonging to the `vm_area_struct` in question. It determines how to resolve the page fault (if possible) and, after taking the necessary locks in write mode, the page tables corresponding to the `vm_area_struct` are updated.

At this point the reader may be wondering how page fault handling works in the case of a shared mapping since we have now established that it may be governed by multiple `vm_area_structs` which in turn are protected by different locks. The answer to this is *files*. By representing even an anonymous shared mapping via a file internally (albeit one in the potentially only internally mounted *tmpfs*), the two different `vm_area_structs` and their corresponding page tables can be kept in sync.

Finally, it is worth pointing out here that a mapping's backing physical pages need not necessarily be contiguous: Observe that the lowest mapping in the right virtual address space is backed by two non-adjacent physical pages.

7.4 Implementation of Noodles

In this section I describe the implementation of the *Noodles* prototype in the form of a support library and a Linux kernel patch series, targeting version 5.15.116 of the mainline Linux kernel [14]. As my own patch series builds up on prior work in the form of an older patch series submitted to the Linux kernel in 2017 [92], I also cover and compare against this work where appropriate.

I explain how this older patch series is supposed to function and point out the fundamental flaws inherent in its design that prevent it from being sound. The bugs discovered range from mere race conditions over implementation problems to fundamental flaws in the basic design, namely dealing effectively with the Linux kernel’s inability to truly share a mapping.

Following this, I contrast this prior work with my own redesigned patch series: I explain how I implemented it, how it functions and why I chose this particular design, motivating my decisions with the chosen use case of the lightweight serverless platform. I also present the user space support library, called *libnoodles*, implemented by me for the purpose of benchmarking the *Noodles* prototype. Finally I conclude this section with some important takeaways from the development process.

Note that after discussing the abstract design principles inspiring *Noodles* and the targeted programming model in section 7.2 I focus on concrete implementation work and the challenges encountered during it in this section. I expressly aim to cover all the nitty gritty details be-deviling any real implementation work, as this is customary what sets apart a functioning prototype from a mere thought experiment captured on a whiteboard. Having to cater to hardware particulars and interface with legacy code, all the while being mindful of real world limitations when contrasting them with theoretical abstractions, is central to systems programming, after all.

7.4.1 Prior work

The original inspiration for *Noodles* came from the prior work done for a paper called *SpaceJMP: Programming with Multiple Virtual Address Spaces* [19] (see also subsection 8.1.1). This paper was published at *ASPLOS* in 2016 and presents the following idea: To switch into a new

virtual address space, it is in fact not necessary to do a full process context switch. Doing this is undesirable because it is expensive due to it requiring that the currently running process’s full context be saved and the new one’s loaded. Instead, switching the full process context and switching the virtual address space should be decoupled, making it possible to *switch only the virtual address space*.

Decoupling these two concepts is possible because it is sufficient (and could conceivably be much faster depending on the circumstances) to only switch out the *pointer to the page table root* to switch to a different virtual address space: On x86-64 the address of the page table root is stored in the CR3 register, also called the Page Directory Base Register (bdbr) [28]. On AArch64 the equivalent register is the TTBR0_EL0 register [7]. The SpaceJMP paper proposes that to switch out the entire paging context with a single store to either of these registers, though this doesn’t take into account any TLB invalidations possibly required³.

Unfortunately, though the central idea presented in the SpaceJMP paper seemed a very promising approach to implement a form of intra-process-isolation, the authors of the SpaceJMP paper had only built prototypes using the operating systems DragonFly BSD [17] and Barrelfish [21], [20] as a basis. Barrelfish is a research operating system now mainly used as a teaching tool to allow students to experiment with alternative operating system’s design. As for DragonFly BSD, even the much more common FreeBSD [76] only seems to hold a server market share in the tens of a percent [101]. (It is non trivial to find reliable numbers detailing the server market shares held by the different operating systems. This is due to it being both hard to measure server market shares exactly and the existing data often not being freely available. Nonetheless, the assumption that the use of DragonFly BSD [17] is not that widespread can be made fairly confidently.)

Given the use case I am targeting with *Noodles* (see section 7.1), I needed an implementation based on a more widespread operating system. In a pragmatic approach I chose the widely used Linux kernel, on which a whole class of operating systems are based. (Note that Microsoft’s server class versions of its operating system also hold a sig-

³If tagging TLB entries with address space identifiers is supported, a TLB invalidation might not be necessary: e.g. some Intel processors support tagging of TLB entries with Process Context Identifier (PCID)s [29] while the Arm equivalent is called Address Space Identifier (ASID) [5].

nificant market share but their source code is of course not generally available to academic researchers [101].)

7.4.2 A patch series

Fortunately, a search of the archival website <https://lore.kernel.org/> for “spacejmp” reveals that there has been a previous attempt to implement support for SpaceJMP like functionality in the Linux kernel: Till Smejkal submitted a patch series to the Linux kernel he called *Introduce first class virtual address spaces* in 2017 [92].

The patch series consists of 13 different parts (individual patches to be applied to a target Linux kernel codebase in the given order). It makes changes to 96 different files, with 5927 insertions and 545 deletions in total. 4 new files called *include/linux/vas.h*, *include/linux/-vas_types.h*, *include/uapi/linux/vas.h* and *mm/vas.c* are added, with *include/uapi/linux/vas.h* extending the user visible API of the Linux kernel. The patch series also defines a total of 9 new system calls for handling the new “first class” virtual address space abstraction and an additional 7 to handle a second new abstraction called *segments*. Note that this new *segment* abstraction is not to be confused with any other abstraction of the same name, e.g. the well known ELF segments. Instead this abstraction is inspired by the also called *segment* abstraction introduced by the SpaceJMP paper.

Perhaps unusually, the cover letter of the patch series admits that in the submitted version the series suffers from a problem the author of the patch series wasn’t able to resolve: Under certain conditions, after switching to one of the new first class virtual address spaces, a thread is no longer able to switch back to its original virtual address space and instead has to be killed.

The patch series received a fair number of comments from seasoned Linux kernel developers. Some of these comments were concerned with superficial issues, mentioning style conventions or pointing out that some user facing interfaces were missing the mandatory accompanying documentation. These issues seem generally easily fixable.

Other comments showed interest for the basic idea but also pointed out that the problem mentioned in the patch series’s cover letter – the inability to undo a virtual address space switch under certain conditions – seemed to be a fundamental one. There was some discussion as to how

high a degree the careful placement of mappings in the virtual address space of a process is the responsibility of the user (the programmer of the application) as opposed to it being that of the kernel. Should it be considered a bug if conflicting mappings in the virtual address space make a switch impossible? Or could this be likened to a call to `mmap(2)` failing due to requesting an unavailable virtual address with the flag `MMAP_FIXED_NOREPLACE`?

This topic of virtual address ranges or mappings conflicting in certain situations and how this gives rise to the problem of being unable to switch back to the original virtual address space is discussed in detail in subsection 7.4.4. To forewarn the reader, upon closer analysis of the patch series's code it turned out that the challenge of avoiding address range collisions and preventing incompatible updates to mappings is a much more fundamental problem than perhaps both the commenting developers and the original author of the patch series initially realized.

Another topic touched upon by some of the commenters was the topic of tagging TLB entries with virtual address space identifiers.

There was also a fair amount of opposition to the proposed new *segment* abstraction due to its functionality being perceived as too similar to what is already provided by the Linux kernel's implementation of the POSIX *shared memory* API [43]. With some justification the question was raised what the benefit of *segments* as proposed by the patch series are when compared to this already existing API.

The POSIX *shared memory* API provides a range of system calls to create shared memory objects. It allows to open them, map and unmap them from a process's virtual address space and close them after use. Note that the Linux kernel implements this API with largely the same code paths also used for shared anonymous memory mappings. This means that internally the creation of such a *shared memory* object is in truth simply the creation of yet another file in *tmpfs*, true to the spirit of Unix.

One crucial limitation of the *shared memory* API worth mentioning here is that when creating a new *shared memory* object/ file in *tmpfs* its size will always be zero initially. It is not possible to take an already existing mapping which was not originally set up as a *shared memory* object and turn it into one retroactively. In my opinion this would be desirable functionality to have since it would allow to save part of a

virtual address space not already backed by a file without any need for copying. However, to my knowledge, *segments* as proposed by the patch series also don't allow for this.

To return to our discussion of the patch series, as it is fairly sprawling and proposed changes to the memory management system of the kernel – considered by some to be an especially sensitive part – it may have had little chance to be accepted in the first place in any case. This, the problem mentioned in the cover letter and the perceived superfluity of the proposed *segments* because they seemed to be too similar to the already existing functionality provided by the *shared memory* API lead to the patch series never even receiving a second, improved version. As of 2024, work on it seems to have been abandoned.

7.4.3 Porting the patch series

Still, even if the work presented in the patch series is incomplete, upon embarking on the *Noodles* project the patch series seemed to provide a valid starting point. I chose to port the patch series to version 5.15.116 of the Linux kernel because it was close to the kernel version of my development machine and the last long term support kernel in the 5er series [89].

To keep development time down I focused my analysis on the parts of the patch series that introduce the basic functionality needed for switching virtual address spaces: That narrows it down to the first 10 parts. (Part 11 introduces the new *segment* abstraction inspired by the Space-JMP paper, part 12 implements a performance optimization allowing to complete part of the work lazily while part 13 adds support for accessing the new first class virtual address spaces via *procfs*.) I also chose to limit my port to the x86-64 architecture (though I later also added support for AArch64), ignoring any changes made by the patch series to support other architectures and I disabled transparent huge pages, simplifying greatly all code dealing with and manipulating page tables.

I successfully ported the chosen parts of the patch series to the Linux kernel version 5.15.116 (enough had changed since the kernel version the patch series had been developed against to prevent automatically applying the patch series). This meant adding additional fields to some of the core data structs of the memory management system, altering some of its functions to account for first class virtual address spaces and

splitting up functions allocating and setting up a `mm_struct`. This last part was necessary to decouple setting up a `mm_struct` and initializing its associated `task_struct`.

The patch series also introduces a whole set of new functions used to copy page tables. These functions, named `dup_<page table level>_range`, can be found in part 09. They iterate over the page table tree in a depth first manner, starting at the `pgd`, with a function in charge of each page table level. At the last level, they iterate over and directly copy the `ptes` found in the page tables.

Note that these functions are heavily inspired by the functions that are used by the Linux kernel itself to copy page tables when `clone(2)` is called in “forking mode”. However, the Linux kernel’s version are slightly different: Instead of making a direct, one-to-one copy of the page tables they instead modify certain `ptes` when iterating over them. To be exact, they modify the `ptes` corresponding to private mappings to enforce COW. (As an aside, I had to implement an additional function following the pattern of the rest since the Linux kernel now has a 5 level page table representation. The Linux kernel version 4.11-rc2 added the `p4d` level as a 5th level [32].)

Finally, after making sense of the code in charge of page table duplication, I analyzed the new files added by the patch series. Those are responsible for adding nine new system calls for the x86-64 architecture (among others). Please refer to code listing 7.12 for the signatures of the new system calls.

These nine new system calls implement the functionality required to create, manage, attach to, switch into and also destroy the new *first class* virtual address spaces, referred to as VAS for short. Each VAS is represented by the patch series as an ID coupled with a name in user space. Each task can form an attachment to a number of VASes, which is a requirement before being able to switch into a VAS. (More about how this works follows in the next subsection, subsection 7.4.4).

Unfortunately, as I analyzed how these nine new system calls operate and how their main data structures are managed, I became aware of several major (and minor) problems with the patch series: Concurrency isn’t properly managed in all cases, also the scheme to uniquely identify the different `vm_area_structs` is sorely lacking, trying to identify them by their start and end addresses only. But worst of all, the problem

```
1  /* Creating a first class virtual address space. */
2  int vas_create(const char* name, umode_t mode);
3
4  /* Deleting a first class virtual address space. */
5  int vas_delete(int vid);
6
7  /* Looking up a first class virtual address space by name. */
8  int vas_find(const char* name);
9
10 /* Attaching a task to a first class virtual address space. */
11 int vas_attach(pid_t pid, int vid, int type);
12
13 /* Attaching a task from a first class virtual address space. */
14 int vas_detach(pid_t pid, int vid);
15
16 /* Switching a task to a first class virtual address space
17    to which it is attached. */
18 int vas_switch(int, vid);
19
20 /* Querying the which virtual address space is currently active.
21    (First class or otherwise.) */
22 int active_vas(void);
23
24 /* Retrieving the permissions of a first class virtual
25    address space. */
26 int vas_getattr(int vid, struct vas_attr * uattr);
27
28 /* Setting the permissions of a first class virtual
29    address space. */
30 int vas_setattr(int vid, struct vas_attr * uattr);
31
```

Figure 7.12: The API for first class virtual address spaces as proposed by the patch series with explanatory comments by the author of this thesis [92]

referenced in the cover letter turned out to be far more severe than first thought.

In fact, it turned out that the Linux kernel's fundamental limitation when handling shared mappings (being only able to share the backing physical memory and not the page tables themselves) is not properly addressed by the patch series. This makes the entire patch series unsound in specific situations, causing e.g. supposedly shared mappings to diverge later on because of concurrent page faults. And as the underlying reason is a conceptual problem caused by how the Linux kernel's memory management system is structured – it does not have a first class abstraction representing a shared mapping, as mentioned in section 7.3 – no easy fix was possible in this case.

In contrast, the concurrency related bugs present in the patch series could conceivably be fixed without any major restructuring of the code. More severe is the patch series's lack of a proper naming scheme for mappings as is described in the next subsection, subsection 7.4.4, but here too it is relatively straightforward how to fix this as the need for unique identification of data structures is a common problem.

7.4.4 How the patch works and why it doesn't

In this subsection we analyze up close how the patch series proposes to realize first class virtual address spaces in the Linux kernel and why it fails to do so successfully (soundly). For this purpose I explain the lifecycle of a first class virtual address space or VAS, walking the reader through a series of possible scenarios. As we progress I point out why under certain conditions undesirable behavior occurs.

The main problem arises because, as discussed earlier in section 7.3, the Linux kernel has no real concept of a truly shared *mapping*. This is unlike other operating systems, e.g. both Dragonfly BSD [17] and Barrelfish [21] have support for this. In contrast, in the Linux kernel is possible to have a block of shared *physical memory* but internally this isn't governed by any shared `vm_area_struct` and instead simply represented by a file in *tmpfs*. No matter if the shared mapping is a POSIX *shared memory* object or has been created by a call to `mmap(2)`, there exists no shared `vm_area_struct` in kernel space. This is likely because of the impossibility of a `vm_area_struct` to be associated with two different `mm_structs` due to the difficulty of finding a working locking scheme in this case.

Instead, for shared memory regions the *tmpfs* workaround is used

if there is no backing file due to the memory region being anonymous. In addition, certain mutations of a `vm_area_struct` are disallowed if it is of type shared. As already mentioned in section 7.3, there exists no straightforward way to turn an originally anonymous and private mapping into a shared mapping at a later point because of the totally different code paths involved internally in their creation and handling.

This is a problem, because of the need for certain mappings containing important state to be present in any virtual address space a task wishes to switch into: While it is theoretically possible to just switch out the page table root with no further considerations (e.g. by simply writing to `CR3` on x86-64 architectures), this is not advisable. Indeed, if no care is taken, this will result in a crash: We have to ensure that for instance the code a task is currently running is also present in the new virtual address space. Otherwise the task will most likely segfault upon instruction fetch (or, even worse, we might end up with completely unpredictable behavior in the unlikely case of something else executable just happening to be stored at the exact same virtual address).

Other examples of execution context state that causes problems if it just disappears upon virtual address space switch include a task's stack, the heap or a library's state⁴.

Without the mappings holding this crucial data present also in the new virtual address space, the programmer would be required to manually place the relevant data at just the right virtual address. This however makes for an extremely delicate and cumbersome programming model since the slightest placement error would lead to hard to debug behavior.

Before delving further into the topic of how basically *hot swapping* virtual address spaces can go wrong, an explanation of how the patch series is supposed to function is necessary: Whenever a `task_struct` (representing a thread) wishes to switch into a fresh virtual address space two steps have to be performed. First, the first class virtual address space object needs to be created in kernel space by the application invoking a system call, `vas_create`. Then the patch series's approach is for the task to form an *attachment* to the new first class virtual address space "object"⁵.

⁴To give an example, I've had cause to discover that `glibc` does not handle it well if it suddenly loses access to part of its state.

⁵Though C is strictly speaking not an object-oriented programming language, the

The `vas` struct First class virtual address space objects as introduced by the patch series are represented in kernel space by a new datatype: the struct `vas`. This datatype can be seen as a kind of wrapper around a struct of a type we have already encountered: the `mm_struct`, representing virtual address spaces in the Linux kernel. The key difference is that the wrapped `mm_struct` is not referenced by any `task_struct` – instead it is specifically created when a new VAS is requested by calling one of the nine new system calls, `vas_create` (see figure 7.12 for its signature). In addition to a pointer to a `mm_struct`, the `vas_struct` also stores needed accounting data such as a reference counter controlling its lifecycle.

When abstracting away all other state apart from the mappings present in a virtual address space, a `mm_struct` can be viewed as an (ordered) collection of mappings or `vm_area_structs`. This is especially true for a VAS address space as created by the new system calls since it isn't referenced by any task and mainly makes use of the fields of the `mm_struct` that are concerned with `vm_area_structs`. In short, the patch series uses the already existing datatype of the `mm_struct` to express the newly introduced abstraction of a *persistent* virtual address space, which is *decoupled* from any specific execution context.

Once created, a VAS exists until it is explicitly destroyed. It is unaffected by the exit of the thread which originally created it or even the termination of the whole process of which that thread is a part of. In this regard a VAS behaves very similarly to a file. In addition, it has an ID just like processes and threads, a name by which it can be looked up (just like a file) and another reference count to count how many `task_structs` have formed an attachment to it. To look up a VAS's ID by its name, the new `vas_find` system call can be used.

A VAS is visible system wide, just as for instance files that can be opened by any process with the right access permissions. The permissions governing a VAS work similarly to the usual Linux file permissions: If a system call is issued trying to attach to a VAS, modify its attributes or delete a VAS, the effective user and group ID of the caller are checked against the user ID and group ID stored in the struct `vas`. Depending

Linux kernel still manages to structure its code in a fairly object-oriented manner that might surprise many a Java programmer. It uses structs instead of actual fully fledged classes and falls back to embedding inner structs such as structs serving as list or red-black-tree nodes when inheritance like functionality is needed.

on what permissions were originally passed on VAS creation, the system call will either proceed or fail with an error.

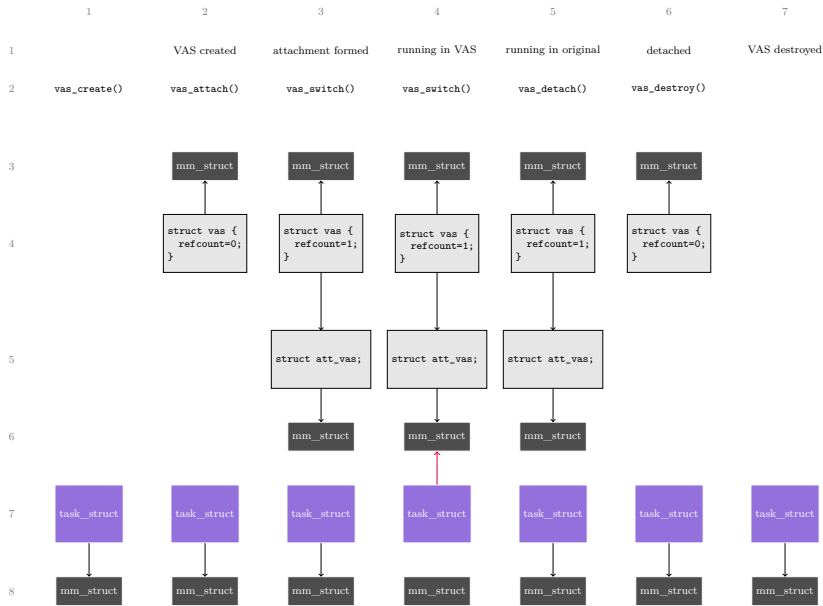


Figure 7.13: The lifecycle of the patch series's new first class virtual address spaces (VAS), time runs from left to right

Attaching to a vas struct After a VAS has been created, it is possible for any task with the right access permissions to form an *attachment* to it (which is required before any switch can take place).

See figure 7.13 for a visualization of the life cycle of a VAS: Time runs from left to right, from *column 1* to *column 7*, the numbers at the left and top most label the rows and columns of the figure. *Row 1* describes the current state of each column. *Row 2* shows the system call issued at a point i in time (*column i*) which then leads the system to transition to the state visualized in *column $i+1$* .

In *column 1* we have a `task_struct` running in its own virtual address space, represented by a `structmm` (*row 8, column 1*). In this

state the new system call `vas_create` is issued. After a VAS and the `mm_struct` it wraps have been created (*column 2, row 3-4*), a system call called `vas_attach` is issued by the `task_struct`. This causes a *third* `mm_struct` to be created (*row 6, column 3*).

This new `mm_struct` is then populated with the *union* of the mappings present in the `mm_struct` of the VAS to which the task is attaching and the mappings present in the task's original `mm_struct`. The newly created *third* `mm_struct` (*row 6*) can be seen as the product of *merging* both the `mm_struct` of the VAS and the `mm_struct` serving as the task's original virtual address space.

In figure 7.13 the newly created attachment is visualized from *column 3* onwards as a `struct att_vas`: Both the created `struct att_vas` encapsulating data about this attachment and the *third* `mm_struct` are shown (*row 5-6*).

Merging the two virtual address spaces ensures that the task does not crash immediately after updating its `mm_struct` pointer (this is how the virtual address space is switched in the Linux kernel) since all mappings that were present in its original virtual address space will also be present in the new one. The *third* `mm_struct` therefore contains the *union* of all mappings present in both the task's original virtual address space as well as all mappings present in the VAS's `mm_struct` which allows the task to continue running unimpeded.

Note that the `struct vas`'s *refcount* field is incremented after the task has formed an attachment to reflect that there is now one *attache*. This is also reflected in figure 7.13 (*column 3*).

Switching to a `vas` `struct` Finally, after having formed an attachment in *column 3*, the task is ready to execute the actual switch to the new virtual address space. From user space this is triggered by calling `vas_switch` with the VAS's identifier.

In kernel space switching causes the `task_struct`'s *mm* field to be re-set by calling a special function, aptly named `switch_mm`. This is the function which performs the heavy lifting of switching out the page table root. It is architecture dependent, meaning different architectures define their own implementations to make sure that the right caches (e.g. TLB) are invalidated at the right time etc.

In the middle of figure 7.13, *column 4*, you can see the state after the

switch has been completed and the task is now running in its new first class virtual address space. Note that the field `mm` of the `task_struct` has been updated and now points to the `mm_struct` constructed during the attachment step instead.

After running in the first class virtual address space for some time the application may decide that the task should switch back to its original virtual address space. To accomplish this it can call the `vas_switch` function a second time to switch the task back to its original virtual address space.

Clean up After having completed the switch back in column 5, calling `vas_detach` causes the attachment formed to be destroyed and leads to resources such as the third `mm_struct` being released back to the Linux kernel's internal caches. (Note that the Linux kernel usually declares dedicated caches for kernel space data structs which are used often enough.)

Once an application decides that the VAS itself is also no longer needed, it can issue the system call `vas_destroy`. If successful (no other attachments exist), this will eventually cause the `struct vas` itself to be cleaned up and then returned to the kernel's caches.

Merging `mm_structs` Since we are now familiar with the lifecycle of the VAS and how the patch series's API detailed in figure 7.12 is supposed to be used we can proceed to having a more in-depth look at the actual merging of virtual address spaces.

Figure 7.14 depicts this in a simplified way: To the left of the diagram is depicted the original virtual address space. It belongs to a single threaded process containing just thread `t1`. The original virtual address space contains three different mappings. For the sake of simplicity we assume at this point that these are private anonymous mappings. (Note however that their type has no real bearing on how merging virtual address spaces works.)

To the right of figure 7.14 the first class virtual address space `t1` wants to attach to is depicted. It contains two mappings that have been installed previously, when another thread was running in the VAS. This thread has since switched back and detached (on detaching the attachment `mm_struct` is unmerged and newly created mappings are

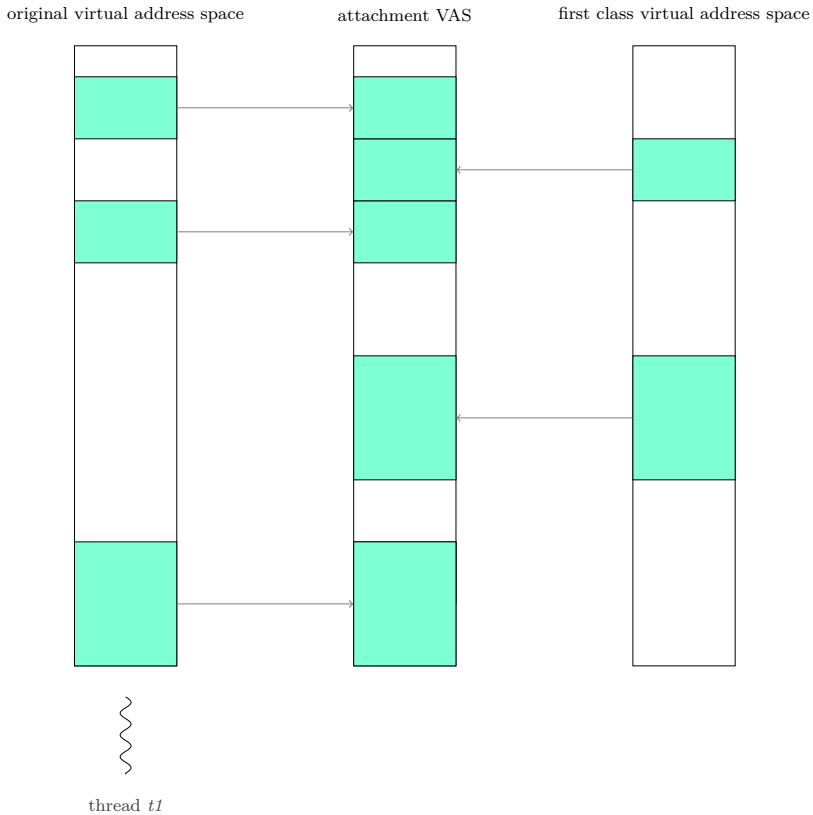


Figure 7.14: How the patch allows a task to attach to a first class virtual address space, a simplified overview

copied into the `mm_struct` representing the VAS).

The two mappings may map physical pages containing data thread $t1$ wishes to access which is a potential reason to attach to a VAS. Since first class virtual address spaces are persistent and decoupled from the life cycles' of processes, storing data in a VAS can potentially be used to persist mappings across the termination of a process. Alternatively, if only a fresh virtual address space is desired for thread $t1$, it is also

possible to attach and switch into an empty first class virtual address space.

The virtual address space shown in the middle of figure 7.14 corresponds to the `mm_struct` created once a thread forms an attachment to a VAS by calling `vas_attach`. As can be seen, the attachment virtual address space contains the *union* of the mappings contained in both the original virtual address space and the first class virtual address space.

The patch series achieves this by *copying* the `vm_area_struct` defining a mapping as well as the corresponding *page tables*. It is effectively making a *deep copy* since the original `vm_area_struct` and its copy don't share their page tables. Only the physical memory pages mapped in the page tables end up being shared. This is the purpose of the `dup_<page table level>_range` functions which we already encountered earlier, in subsection 7.4.3.

Mapping collisions Of course, looking at figure 7.14, it becomes immediately clear that simply unifying the mappings of both virtual address spaces (the original one and the VAS) is potentially problematic: What happens if a VAS happens to coincidentally have a mapping occupying a virtual address range that *intersects* with that of any mapping of the original virtual address space?

The answer to this question is that in this case, attaching to a VAS fails with a non-zero exit code. There is simply no way to reconcile two conflicting mappings. See figure 7.15 for a visualization of two mappings with conflicting virtual address ranges.

However, this is less of a drawback than it might seem at first glance. For one, having a system call potentially fail due to the presence of conflicting mappings in a virtual address space is nothing new: This behavior is very similar to what happens if `mmap(2)` is called with the flags `MMAP_FIXED_NOREPLACE` [39] and the caller asks the kernel for a virtual address which isn't available. (Note that the flag `MMAP_FIXED` might instead lead to part of an existing mapping being discarded if a conflicting mapping is present in the virtual address space!)

Moreover, a collision of the virtual address ranges of two mappings when merging the original virtual address space and the VAS is far less likely to happen than figure 7.15 might suggest. We have to remember that it is not drawn to scale for the obvious reason that today, on state of

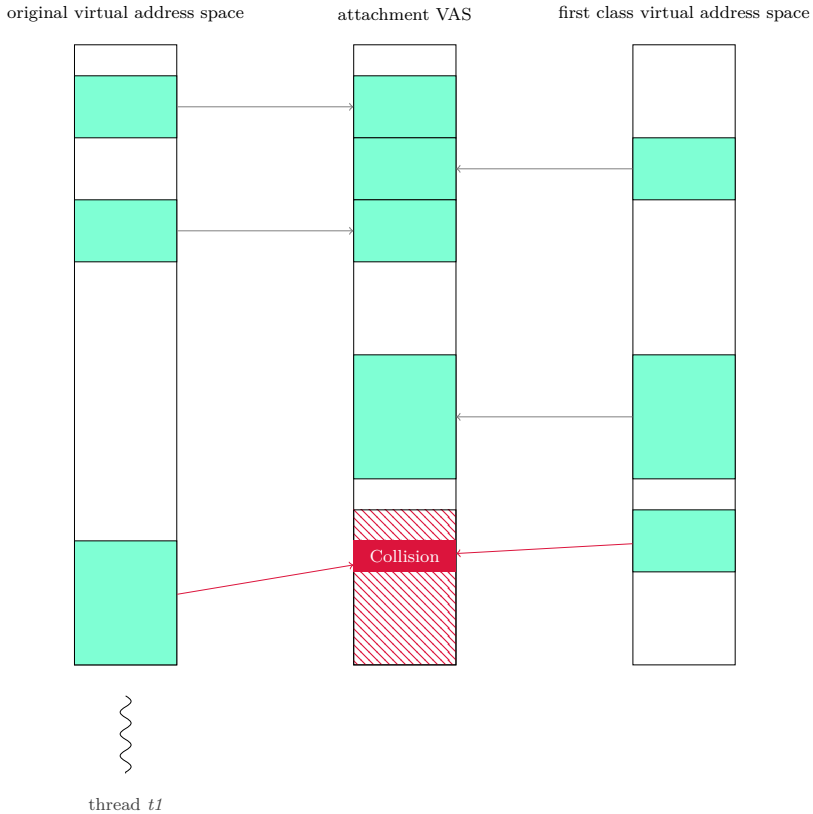


Figure 7.15: If in the first class virtual address space a mapping intersects with the virtual address range of a mapping in the original virtual address space, a collision occurs and attaching to the first class virtual address space fails.

the art hardware running recent Linux kernel versions, virtual address spaces are simply gigantic (and fit on no page): For example with 48 virtual address bits actually used, on x86-64, from the Linux kernel version 4.14 onwards, the virtual address space portion owned by the user is 128 TB big. With 56 virtual address bits used, this grows to a

staggering 64 PB [33].

So, keeping this in mind, is the collision of virtual address space ranges even a problem in practice? Unfortunately it is indeed, however, not primarily on forming an attachment to a VAS in the first place but later, right before switching to the attachment `mm_struct` and when trying to switch back. This is due to perform a *syncing* step right before actually making any switch. The syncing step is supposed to keep the copied `vm_area_structs` consistent with the `vm_area_struct` they are a copy of.

Syncing virtual address spaces At this point we have to remember that forming an attachment to any VAS is done by invoking the system call `vas_attach` but the actual switch into the new virtual address space will only take place later, once a thread calls `vas_switch`. This implies that its possible for other code to run *in between* these two system calls. That code might be executed by the calling thread itself but its also a possibility that some other thread executes in the same original virtual address space we start out with. In either case, this might lead to new mappings being placed in the original virtual address space in the meantime. Alternatively mappings might also be altered in a number different ways (e.g. a mapping might be extended, deleted or it might suffer a page fault).

And since the Linux kernel has no way of truly sharing the mapping *itself*, as we have seen earlier, in section 7.3, these changes will not be reflected automatically to the corresponding mappings contained in the attachment `mm_struct`. Because of this our copies might become outdated before we even get to calling `vas_switch`.

This is problematic since on forming an attachment, the attachment `mm_struct` is set up to contain the exact union of both the mappings contained in the original virtual address space as well as in the VAS. Remember, this is done to make sure that the thread switching can keep running uninterrupted, without crashing due to some of its data suddenly disappearing under it. This could happen, if for instance a new mapping was created in the original virtual address space between the two system calls: A mapping could be created by a library that needed more space to maintain its state and if we now just switched to the outdated `mm_struct` created on attachment, it would seem to that

library as part of its state had suddenly vanished into thin air.

The patch series tries to resolve this problem by so called syncing the attachment `mm_struct` with the original `mm_struct` right before actually switching a thread over. This is always done before every switch, not only on the switch to the new virtual address space but also on the switch *back* to the original virtual address space. And as a result of this, a switch might potentially *fail* (in any direction). This is because the sync the patch series performs does not succeed in all circumstances.

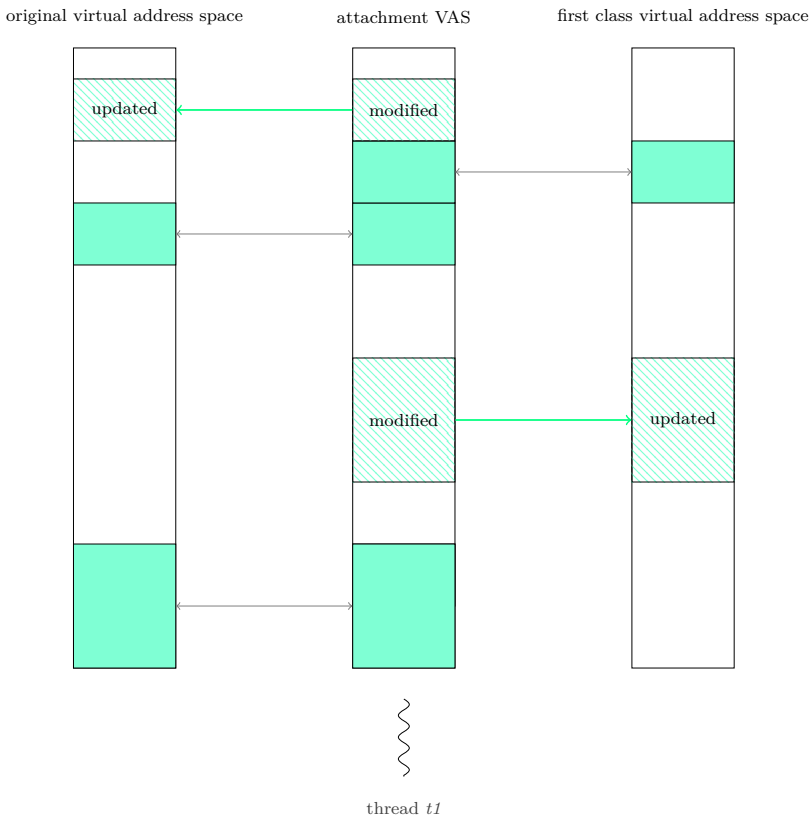


Figure 7.16: Preparing to switch back to the original virtual address space after running in the attachment VAS.

But before we will analyze in detail what can go wrong when trying to sync two virtual address spaces with each other, please see figure 7.16 for a visualization of a *successful* sync.

Figure 7.16 shows a possible situation after thread $t1$ from earlier (recall figures 7.14) has successfully switched to the created attachment virtual address space. It has been running in it for some time, executing code as planned and as a result of this, two of its mappings happen to have been *altered*.

One possibility for a mapping to be altered is for it to change its size. Another possibility is for its permissions to change or, perhaps, most likely, it suffers a page fault. As we have seen in section 7.3 a mapping might not be fully paged. Suffering a page fault after having been copied will cause its page tables to be updated (and the copy's page tables will not receive this update automatically).

All of these changes need to be reflected back to the corresponding copy of a mapping during a sync. In figure 7.16 both a mapping originating from the original virtual address space and one originating from the VAS have been altered in some way. Before the thread $t1$ can switch back, these changes need to be reflected back to the respective virtual address spaces to prevent data from suddenly disappearing (which is only one of the possible pathological behaviors we might get if the two virtual address spaces are out of sync on switching).

Conflicting updates Where the situation gets truly complex – as is so often the case – is in the multi-threaded case. Imagine that we have two threads $t1$ and $t2$ executing in the same process and after $t1$ has been switched to a new virtual address space, $t2$ continues to run in the original virtual address space. The result of this could be likened to two programmers editing the same file at once – there is always a possibility that this will lead to *conflicting updates* that cannot be resolved. This is also when we are most likely to end up with conflicting virtual address ranges.

Please see figure 7.17 for a visualization of an example of an update conflict: Imagine that our two threads, $t1$ and $t1$, running concurrently in their respective virtual address spaces, have each caused two mappings to be altered. Imagine further that $t1$ now wishes to switch back to its original virtual address space, mandating a re-syncing of the attachment

virtual address space and the original virtual address space (as well as a re-syncing of the attachment virtual address space and the `mm_struct` of the VAS).

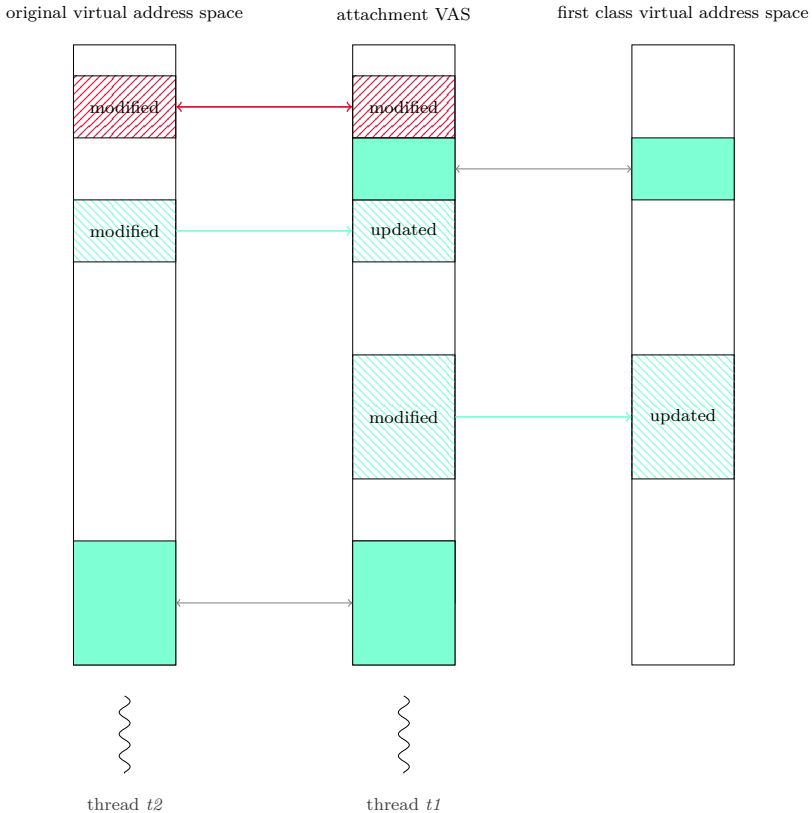


Figure 7.17: Preparing to switch back to the original virtual address space after running in the attachment VAS, conflict situation.

While some of these modifications can successfully be reflected across to the corresponding copy of a mapping (mainly if only one of the two copies has been altered) this will not work if both threads' execution has lead to the two copies of a mapping being altered in incompatible ways.

(Note that updates to a mapping need to be reflected both ways.)

To give an example of altering the two mapping copies in incompatible ways, its possible that both threads page fault on the second page of a mapping independently of each other. (That mapping might not have been fully paged in when it was copied during the attachment step.) Now, if the mapping in question is a shared mapping, this will not cause a problem because (all shared mappings being actually files) the page fault will end up being serviced with the same physical page. However, if we are dealing with a private mapping (be it file backed or anonymous), the two page faults will potentially result in the second part of the mapping being backed by *two different physical pages*. Basically the mapping *diverges*, just like a file with incompatible updates.

This is not only a problem because there is no way to reconcile these two copies with each other (simply discarding one of the physical pages in favor of the other will result in lost data). It is also a headache because it leads to a not very user friendly programming model: Depending on how much of a mapping was already paged in when switching to the VAS we might end up *partially* sharing a mapping!

Another example of an incompatible modification is if the top most mapping in the original virtual address space ends up being extended downwards, e.g. because it is coalesced with another, newly created mapping. Even if thread *t1* has made no modifications to its own copy of the top most mapping, we cannot re-sync it with the original virtual address space because in the attachment virtual address space there is no space to extend it.

No matter the reason, be it virtual address range conflicts or simply incompatible modifications made concurrently, there are many possible scenarios where an attempt to re-sync the attachment virtual address space fails. Depending on if this occurs after we have already switched to the attachment virtual address space or before we make the switch, the outcome might either be that we are unable to switch at all or that the thread gets stuck in the attachment virtual address space. Both scenarios are not desirable and potentially result in having no other choice than having to kill the thread in question.

Diverging page tables Even worse than a switch potentially failing due to irreconcilable modifications is that we might end up with par-

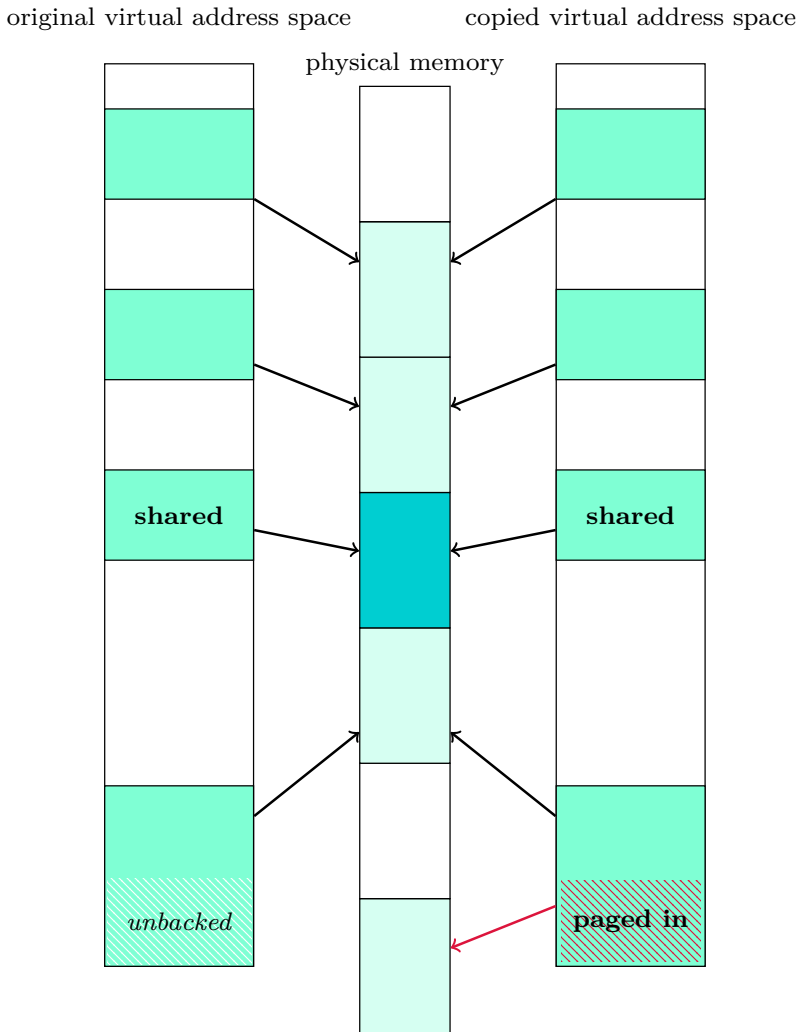


Figure 7.18: Mappings that are not fully paged in when they are copied can diverge if a page fault occurs later on.

tially shared mappings. This holds true even if the thread *t1* never even attempts to switch back to its original virtual address space, making it a bug that cannot even be mitigated by switching just once. While the line between feature and bug is sometimes fine when turning the virtual address space model practically on its head like the patch series does his kind of behavior simply does not allow for a sound programming model: How much of a mapping is already paged in should be transparent to the application (at least in terms of correctness even if not necessarily in terms of performance).

To flesh out this clearly undesirable scenario a bit more, imagine a newly created, private anonymous mapping that isn't fully paged in yet. This might be the case because part of its virtual address range has just never been accessed before. Such a mapping is shown at the bottom left of figure 7.18.

In the middle of figure 7.18 the physical memory pages are shown as an array. To its left and right we have two virtual address spaces: To the left is the original virtual address space, to the right the copied virtual address space.

Some time after all the `vm_area_structs` in the left virtual address space have been copied, an access to the second page of the mapping in the bottom right corner of figure 7.18 is made. A page fault occurs, the kernel gives us a fresh physical memory page and the page tables corresponding to the `vm_area_struct` are updated.

Since the `vm_area_struct` is a deep copy of its original, we will end up with two `vm_area_structs` that share all of their backing physical pages that were already mapped in the page tables when the copy was made but none of the physical pages that are paged in afterwards.

Identifying *twins* Another problem that the patch series doesn't sufficiently address is how to determine if two `vm_area_structs` are what I call *twins*, meaning one of the two was created by copying the other. This might seem straightforward at first glance – can't we just examine the virtual address spaces and if they occupy the same virtual address ranges declare them to be *twins*? – but this isn't sound for a number of reasons:

- What if one the two mappings is extended?

- What happens if one of the mappings is coalesced with an adjacent mapping?
- What if a mapping is deleted and then a new one is created in its place?

This last issue is reminiscent of what is sometimes referred to as the *A-B-A* problem which can occur in many different contexts. It is characterized by code failing to detect that a modification has taken place, say that a stack has been modified, because another thread's modifications coincidentally sum up in a way that looks identical to the previous state. In the case of a stack, this would be another thread popping *A*, pushing *B* and then pushing *A* again. If code only checks the top of the stack to make sure nothing has changed since it last checked, this will cause it to miss the fact that the other threads have modified the stack in the meantime.

In the case of the patch series a mapping being deleted might be missed by its syncing algorithm if another mapping is later created which just happens to occupy the same virtual address range (by chance or by malicious design). In fact, since the syncing algorithm tries to account for the fact that a mapping might grow, this bug can even be triggered if the new mapping doesn't occupy exactly the same virtual address range – just similar is already enough.

This problem could conceivably be addressed by having `vm_area_structs` keep track of either their original or their copies. However there is the slight wrinkle of how to handle coalescing mappings that needs to be addressed by any real solution.

7.4.5 Call graph of the patch series part 10

After having implemented a patch (series) to the Linux kernel there essentially two options if the goal is to use the new features/ fixes in a production setting: Either the patch (series) has to be continually maintained by its creators, porting it to new Linux kernel versions to make sure it doesn't at some point become obsolete, or it has to be upstreamed.

Unfortunately both these options are not very realistic when it comes to the patch series discussed in the last two subsections, subsection 7.4.3

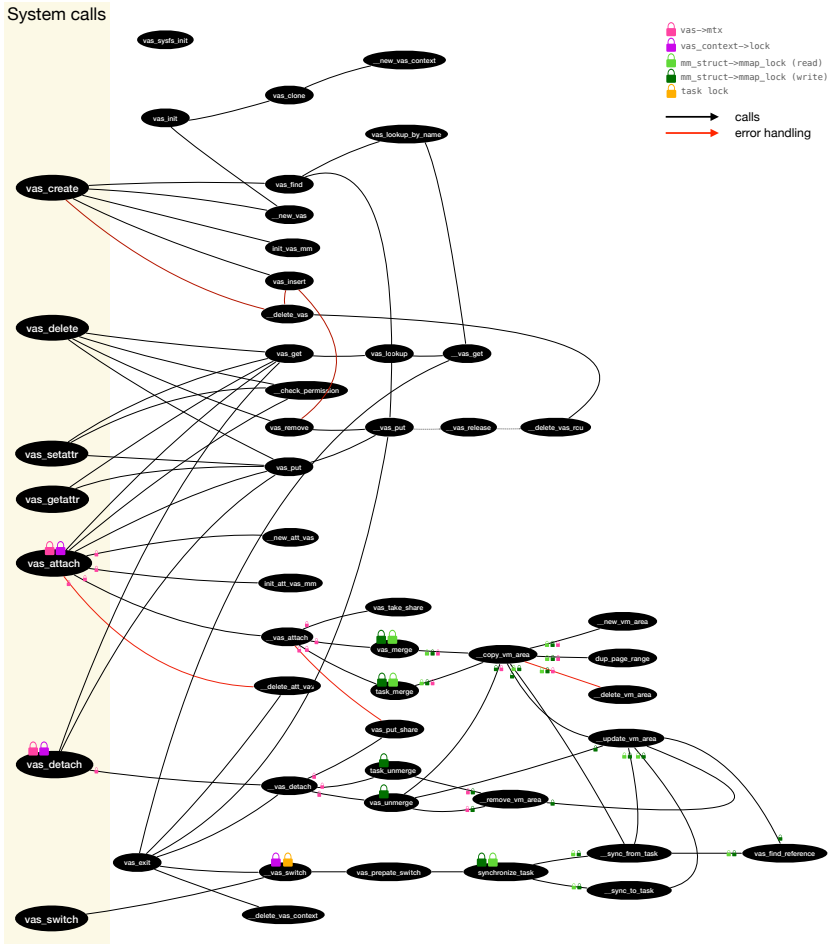


Figure 7.19: A call graph visualization of the code contained in file *vas.c* as well as in the accompanying header files, part of patch 10, patch series [92].

and subsection 7.4.4. Even leaving aside the already mentioned problems, the patch is simply too sprawling to be ever accepted by the Linux kernel community: See figure 7.19 for an call graph visualization of only part 10 of the patch series.

Just the locking scheme of the VAS is non-trivial, see the lock symbols in figure 7.19, as is evidenced that by the fact that the original patch series suffers from a possible race condition.

All of this complexity means that the other option, self-maintaining the patch series also isn't appealing. This is a pity because though there is definitely promising functionality contained in the patch series (and a huge amount of prior work), despite all its flaws. The basic idea of decoupling virtual address spaces from processes is still very powerful though in this thesis I will focus solely on the use case of intra-process-isolation and won't spend any time on the other potential use cases such as caching the contents of a virtual address space, sharing virtual address space content more easily etc.

For these reasons, I decided to re-design the patch series, with my design goals for the re-design motivated by the flaws and bugs detailed in this and the previous subsections, subsections 7.4.3, 7.4.4.

7.4.6 My patch

In this subsection, after having analyzed in depth the prior work in form of a patch series [92] in the previous subsections 7.4.3, 7.4.4, 7.4.5, I now continue by discussing my own implementation work. The re-designed implementation changes the focus from the design goals stated in the SpaceJMP paper [19] to providing intra-process-isolation instead.

In addition to that, there are number of issues which I discovered when analyzing the prior patch series that need to be addressed. The main issues are briefly summarized here for the readers convenience:

- **Race condition**

The locking and reference counting scheme of the patch series is unsound. There are certain interleavings of the new system calls that will lead to a corrupted state.

- **Mappings cannot be reliably identified as *twins***

The approach taken by the patch series to detect if a mapping was

copied from another – simply looking at the virtual address ranges while allowing for the fact that a mapping may grow down if a specific flag is set – is too simplistic.

- **An attachment may fail due to conflicting mappings**
Depending on placement of mappings in both the VAS and the original virtual address space prior to calling `vas_attach` an attachment may fail.
- **Switching into a VAS may fail due to conflicts**
Switching into a VAS may fail even after the attachment has already succeeded. This is due to the original virtual address space (or even the VAS if it is attached to multiple tasks at the same time) having changed in irreconcilable ways since forming the attachment.
- **Switching back may fail**
Switching back to the original virtual address space may also fail due to the two virtual address spaces having diverged beyond recovery. The most likely case for this to occur is if another thread that hasn't switched to the VAS continues running in the original virtual address space. However, since the method the patch series uses to detect if mappings are *twins* is imperfect, this problem may even occur in a single-threaded context.
- **Unsound**
A not fully paged in mapping may end up being shared partially by different threads, with some pages of the mapping being shared while others are not. Unless an application is aware of its exact paging state, this flaw leads to unpredictable behavior.
- **Complexity**
The patch series is too sprawling to be manageable, be it for easy maintaining or for upstreaming such a novel idea. The Linux kernel community relies on feedback supplied by volunteers (mostly), which is even more the case with controversial proposals than simple bugfixes. However, it is difficult to get meaningful and comprehensive feedback on a patch series this complex.

- **API to big**

Too much is added to the user API (implying that it may need to be maintained indefinitely once it has been added to the Linux kernel). This also lowers the chance of upstreaming the patch series further.

The two most important problems that need to be addressed are one, that two mapping copies may diverge, and two, being unable to switch due to the (re-)syncing of the two virtual address spaces failing. Unfortunately both these problems are not straightforward to address since at their root lays the Linux kernel's inability to truly share a mapping – they are thus the consequence of a conceptual problem.

To mitigate this, reconsidering on a conceptual level which part of the functionality originally proposed in the SpaceJMP paper [19] is strictly needed is key. In the case of *Noodles* this means focusing on those parts of the proposed functionality that are essential to enable intra-process-isolation.

Design principles for the *Noodles* implementation For my own patch series I identified the following design principles:

- **Tailor the implementation to the use case targeted**

The *Noodles* prototype is focused exclusively on a specific use case: This is the use case of the lightweight multi-tenant serverless platform and to narrow it down further, I focus only on memory access related intra-process-isolation. The benefit of this is that certain cases complicating the code before can now be disregarded altogether, simplifying the implementation greatly.

For instance, if only intra-process-isolation is desired, it is not necessary to support attaching to a VAS already containing mappings. And if we only support attaching to an empty VAS in the first place the problem that an attachment may fail due to conflicting preexisting mappings resolves itself.

Another example of the simplifications made possible by closely tailoring to the lightweight serverless use case is to completely disallow ever switching back. If we aim to sandbox user supplied code in a VAS, it is not necessary for a thread to be able to switch back

to its original address space. Instead it can simply terminate after having completed the user-provided function.

Finally, it is also possible to disregard any complications arising from mappings being made writable after having been copied into the new virtual address space. After the VAS has been created and the thread switched into it, due to our use case we only need to support *downgrading* permissions in the initial set-up step before running user supplied code. Following that, changing permissions (calling `mprotect(2)`) will be disallowed altogether. This can be enforced with either a `seccomp(2)` filter [42] or potentially by using the new system call `mseal(2)` [30] (though *Noodles* would have to be ported to a more recent Linux kernel version for this). In addition, I also make the assumption that the application hyper-visor will not increase its permissions on a mapping after having copied it into a VAS.

- **Make additional simplifying assumptions to manage the complexity of the implementation**

As this is a prototype intended as a proof of concept, I decided to make a number of additional simplifying assumptions: For instance I disabled the support for Transparent Huge Pages (THP) in the kernel configuration. This particular feature of the Linux kernel has hugely complicated all page fault handling code and would also mandate dedicated page table duplication functions.

Handling calls to `mremap(2)` is also considered out of scope though a simple solution would be to simply disallow problematic operations by setting specific flags for a mapping.

- **Identify and manage the problematic mapping types**

One key idea to finding a solution for the problem that the Linux kernel does not support truly shared mappings is to *differentiate between different cases* instead of aiming for a more general solution. For example, *private read-only* or *shared* mappings are fine (as we will see further below).

In fact, it's only the writable *private* mappings that need special handling. To prevent this type of mapping from diverging when suffering page faults concurrently later on, a possible solution is

to pre-fault the mapping in question before copying it. As this is expensive though, we only want to do this if absolutely necessary and not for all mappings in general.

- **Re-design the API with a focus on flexibility and simplicity**

Another main goal with a lot of synergy with the above goal of tightly tailoring the implementation to one use case is to make the API much smaller. I decided that a single new system call will be enough, merging several of the new system calls proposed by the patch series. This greatly simplifies testing and means the surface of attack is reduced.

- **Push some of the complexity to user space**

As it is difficult to get good enough performance with a completely automatic, general solution when deciding which mappings to pre-fault I choose to instead push some of the decision making to user space. The reasoning being that due to the targeted use case the user application envisioned for the prototype is a kind of hypervisor application: This means we are not dealing with a general purpose application here but one that can be trusted to make informed decisions about the mappings placed in its own virtual address space and those of its clients.

After our discussion of the design principles governing the *Noodles* prototype, let's have a closer look at mitigating the problem of the Linux kernel not properly supporting truly shared mappings:

A closer look at the contents of the virtual address space To solve the problem of the diverging mappings, it is important to have a look at what type of mappings an application typically has installed in its virtual address space: For this, please see figure 7.20. It depicts two virtual address spaces with the right one being a copy of the left one. This time, unlike the visualizations from before, more detail is given as to what type of mappings (and with which permissions) are in fact present in the virtual address spaces.

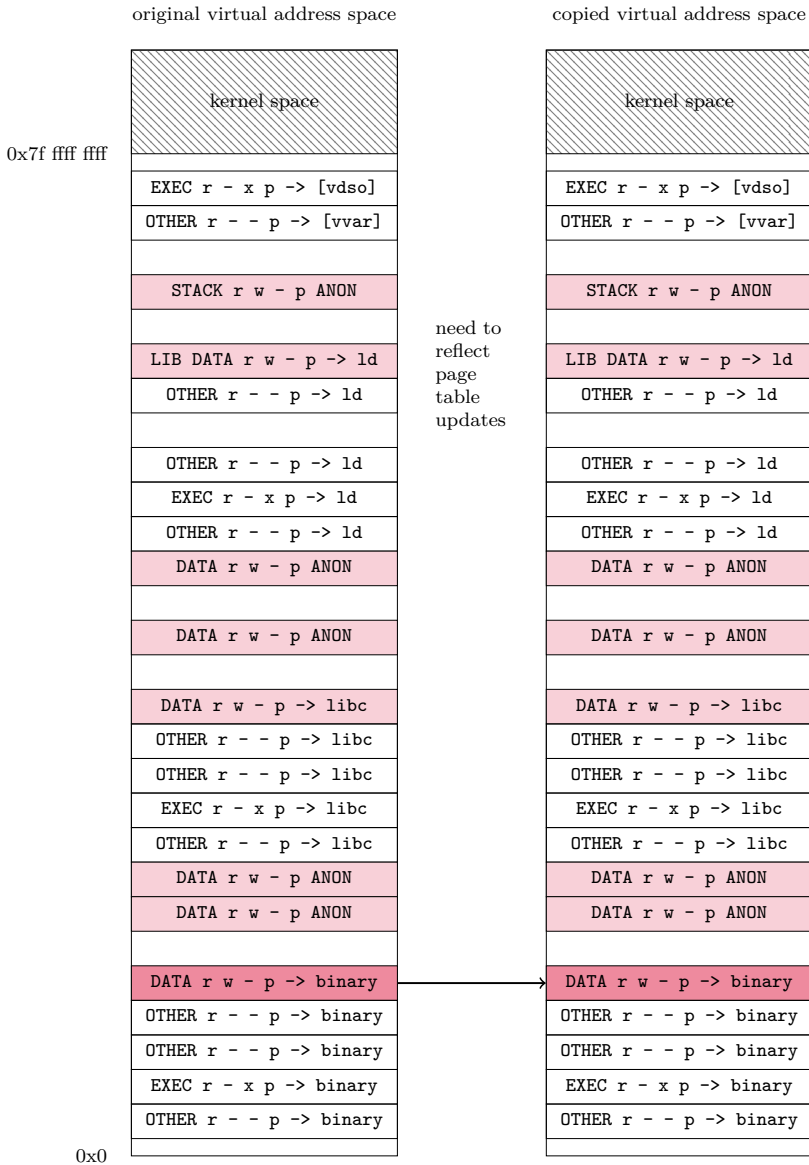


Figure 7.20: Visualization of the mappings present in a basic application’s virtual address space (Warning: Not drawn to scale.)

Note that due to space constraints the mappings shown in user space are not drawn to scale and neither is the kernel space (being in fact of equal size for x86-64 and AArch64 for Linux kernel versions from 4.14 onwards [33]). Also, the impression that most of the user's part of the virtual address space is filled is a false one – the mappings are much magnified to allow the reader to make them out and me to label them.

Keeping this in mind, it is still educational to see a real-life example of the mappings installed in a basic application (compiled from a simple C program, run on x86-64, assuming that 48 virtual address bits are used):

- `[vdso]` and `[vvar]`:
These two mappings are always present somewhere near the top of the user's part of the virtual address space. Virtual Dynamic Shared Object (vDSO) is a shared library automatically mapped into the virtual address space by the Linux kernel itself for the purpose of optimizing certain system calls [44]. `vvar` is its predecessor.
- The stack:
The stack is normally in its own mapping though be aware that it is possible for the Linux kernel to place an automatically allocated stack of an other thread in the same mapping. Most important for us here is that the stack is both read and writable and a private, anonymous mapping (notice the `p` next to the permissions in the figure).
- `/usr/lib/x86_64-linux-gnu/ld-linux-x86_64.so.2`:
The next five mappings map the linker and are all map parts of the same file (with different file offsets which are not shown in the figure). Notice that the different parts of the file are mapped with different permissions.
- `/usr/lib/x86_64-linux-gnu/libc.so`:
Five mappings labelled with `-> libc` belong to glibc. Again, they map different parts of the same file (different file offsets) and depending on their purpose they are mapped with different permissions.

- *binary*:
The mappings labelled `-> binary` map parts of the actual ELF file we are executing. There are several read-only parts, one read-writable mappings and finally the executable mapping containing our actual code.
- **ANON**:
Finally we have several mappings labelled **ANON**. These are all read-writable and are not backed by any file, they correspond to private anonymous mappings. One of these corresponds to the heap but of course both the application itself and libraries such as `glibc` may create additional such mappings.

The reader may have noticed at this point that all mappings shown in figure 7.20 are private. Private mappings are normally far more common than shared ones since those have to be set up especially for a specific purpose.

For my prototype I analyzed all of the possible mapping types, coming up with the key idea that the category of mappings that needed to be specially handled with a point solution – pre-faulting them – is the category of mappings that are both private and also writable.

The key idea After having gained a better understanding of the shape of the problem by analyzing a typical virtual address space layout, let's now discuss each of the different mappings types, making a case distinction for the different possible permissions set where appropriate. We determine for each case if special treatment is needed when copying. The conclusion of this is that there is one category of mappings which need to be taken care of by pre-faulting them: Mappings that are both private and *writable*.

The most straightforward case is that of the *shared* mapping: No matter if backed by a file or anonymous and irrespective of any permissions, *shared* mappings never need to be prefaulted. Even if page faults are suffered independently of each other for two copies of the same *shared* mappings, the page tables cannot diverge. This is due to the fact that the Linux kernel manages all *shared* mappings via the file abstraction: Because of this, the page tables for *shared* mappings can be reconstructed at any time, a fact that the Linux kernel itself takes advantage of when

optimizing `fork(2)` (while all other page tables are duplicated, those for *shared* mappings are reconstructed on demand for the child [48]).

Please see the table shown in figure 7.3 for an overview of the many different possible cases for *private* mappings. As can be seen, some cases are either by the nature of our use case or our simplifying assumptions made in the list of design principles 7.4.6 unproblematic, others are equivalent from the point of view of *Noodles*.

As can be seen, the cases we need to watch out for are the cases where a virtual address region is mapped as *writable*. To prevent these mappings from diverging we have to make sure that page faults are not suffered independently from each other, which can be solved by prefaulting the mapping in question. (Note that swapped out pages are no problem as in this case a *swap entry* will be present which can just be copied.)

Permissions	Note
r - -	This case does not need special handling: Private, read-only mappings are normally backed by a file since a private anonymous mapping that has never been written to by any process will just be all zero (in this hypothetical edge case the mapping diverging would be irrelevant). A page fault will thus just cause another page of the same backing file to be linked into the page tables. (Note that the manual page of <code>mmap(2)</code> specifies that if the backing file changes after the call to <code>mmap(2)</code> has already mapped the file, it is undefined if those updates are visible to a thread.) Of course, it's possible that a mapping is made read-only <i>after</i> having been written to previously. However, in this case the pages that have been written to are guaranteed to have been linked to by the ptes at some point. Even if these pages are later swapped out, a copying of the <code>vm_area_struct</code> and the relevant page table range will retain that information (via the <i>swap entry</i>), making sure that a later page fault will result in paging the right page back in.
r w -	This case requires prefaulting the page tables of the mapping (both for the anonymous and the file backed mapping). Otherwise, if the page tables are copied before the entire mapping is paged in and a page fault occurs later on, it is possible for this type of mapping to diverge.
r w x, - w -, - w x	For our purposes we can treat these cases as equivalent to r w -.
- - x, r - x	For our purposes we can treat these cases as equivalent to r - -.
- - -	This is obviously an edge case but with the assurance that permissions will ever only be downgraded (in the new virtual address space) or stay the same (in the hypervisor application), this is trivially safe.

Table 7.3: The table shows a case distinction of how to handle private mapping types when copying their corresponding `vm_area_struct`.

Noodles Keeping the design principles from list 7.4.6 in mind I chose to invent a new abstraction called a *noodle* (see also the previous section 7.2). A *noodle* is in its essence a *fat thread* – meaning a thread with its own page tables while still sharing everything else (the thread group ID, file descriptors etc.). My own patch series implements this by having a task attach to and then immediately switch into an empty VAS.

To prevent the two virtual address spaces from diverging between the two system calls `vas_attach` and `vas_switch` (and thus possibly preventing the switch), I merged them into a single system call, named `enter_noodle`. This system call also takes care of creating the VAS in the first place – since it is empty by design, there is no benefit to allowing users to create VASes as separate objects with their own lifecycle. All empty VAS are identical after all.

A thread can be turned into a *noodle* at any point in time but there is no undoing this transformation, as I decided against supporting the switch back into the original virtual address space. If more than one thread per VAS is desired, rather than allowing multiple threads to separately attach and then switch into the VAS, instead the single thread that has switched (and is now called a *noodle*) can simply create more threads directly.

To address the issue of the diverging mappings, I chose to pre-fault the mappings types which my analysis in figure 7.3 identified as critical before copying the `vm_area_struct`. To mitigate the performance penalty this naturally carries, I implemented an API that – despite consisting of only a single system call – gives the caller much more fine control: While the original patch series gives the caller no choice as to which mappings present in its original virtual address space will be copied into the VAS, my own patch series pushes this choice to user space.

The caller can decide which mappings will be copied into the freshly created *noodle*'s virtual address space by passing the right parameters to the new system call `enter_noodle`. A number of flags, as listed and discussed in the next subsection 7.4.7 allow the caller to choose first a general policy for deciding which mapping types should generally become part of the *noodles* virtual address space.

To allow for even more fine control, `enter_noodle` allows the caller to pass a list of mapping start addresses which will be interpreted as

either an exclusion or an inclusion list (depending on the flags passed). This allows the user to *override* the default policy for specially chosen mappings, making the API very flexible.

Exposing an API like this, to let *the caller* decide which mappings should be copied into a *noodle*'s virtual address space, allows to push some of the complexity to user space solely handled by the kernel before. The rationale behind this is that the application calling the system call (in our use case this will be the hypervisor application) knows much more about what data is stored in its virtual address space, what is stored where and, most importantly, which data should remain available to the *noodle* and which data should be hidden from it.

Letting the application make the decision also helps to mitigate the cost of pre-faulting private, writable mappings: The user can exclude any unneeded private, writable mappings and instead only include necessary mappings such as that containing the stack. It is also probable that explicitly included private and writable mappings are “hot”, meaning in this context that they have either been accessed recently or will be very soon. In the former case, the physical pages backing the mapping will likely already be present in the page tables while in the latter case we would suffer a page fault anyway when the thread turned *noodle* accesses that data.

Apart from cost savings, another advantage gained by giving the caller the ability to exclude certain mappings is that it allows to carefully choose the data a *noodle* has access to, finally giving us the intra-process-isolation we desire. In fact, this is even stronger than what can be done with Intel's MPK or Arm's PIE/POE: Using memory keys/ protection domains allows to mark certain regions of the virtual address space as not accessible but the backing physical pages remain mapped into the virtual address space via the page tables. If an attacker manages to flip a bit in a pte, e.g. with a Rowhammer [34] like attack, they could thus regain access.

In contrast, isolating user provided code in a *noodle* allows to not have a mapping installed in the virtual address space at all by explicitly excluding it when the *noodle* is created. This means that there are no page tables to target with Rowhammer and no pte bits that can be flipped.

Another point worth emphasizing is that choosing only a few map-

pings to be copied into the virtual address space of the *noodle* reduces the number of page tables the kernel has to copy, trimming down the page table tree and the associated cost. That is especially important when comparing to the traditional `fork(2)` which is the only other way to get a fresh virtual address space for an execution context.

See figure 7.21 for a visualization of the effect of calling `enter_noodle` on the page tables of a process.

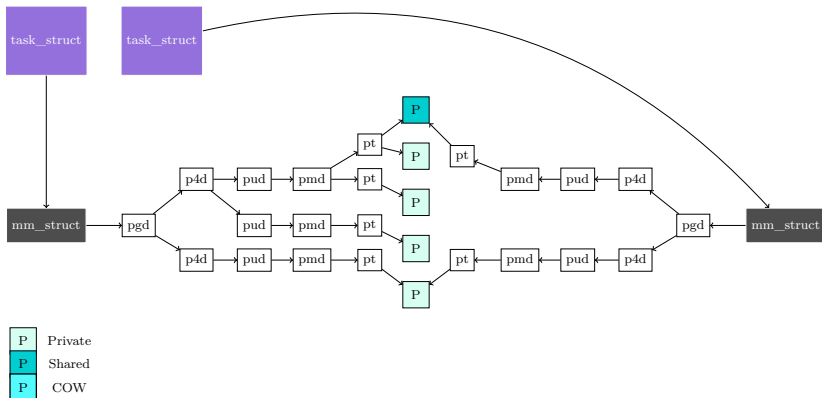


Figure 7.21: State of page tables after calling `enter_noodle()`: Only a subset of the page tables is copied.

All of the design decisions mentioned above greatly simplify the implementation of *Noodles*. As a result of this, the call graph corresponding to the central functions of my implementation is much smaller and easier to understand than that of the original patch series.

My patch series's call graph Figure 7.22 shows the call graph of the central part of my own implementation. Notice that in contrast to the call graph shown in figure 7.19 here we only have a single system call, `enter_noodle`, which is the sole entry point into the kernel part of the *Noodles* prototype. (Even though some of the functions bear the same name as a function in the call graph of the original patch series, I have changed or re-implemented many of these functions.)

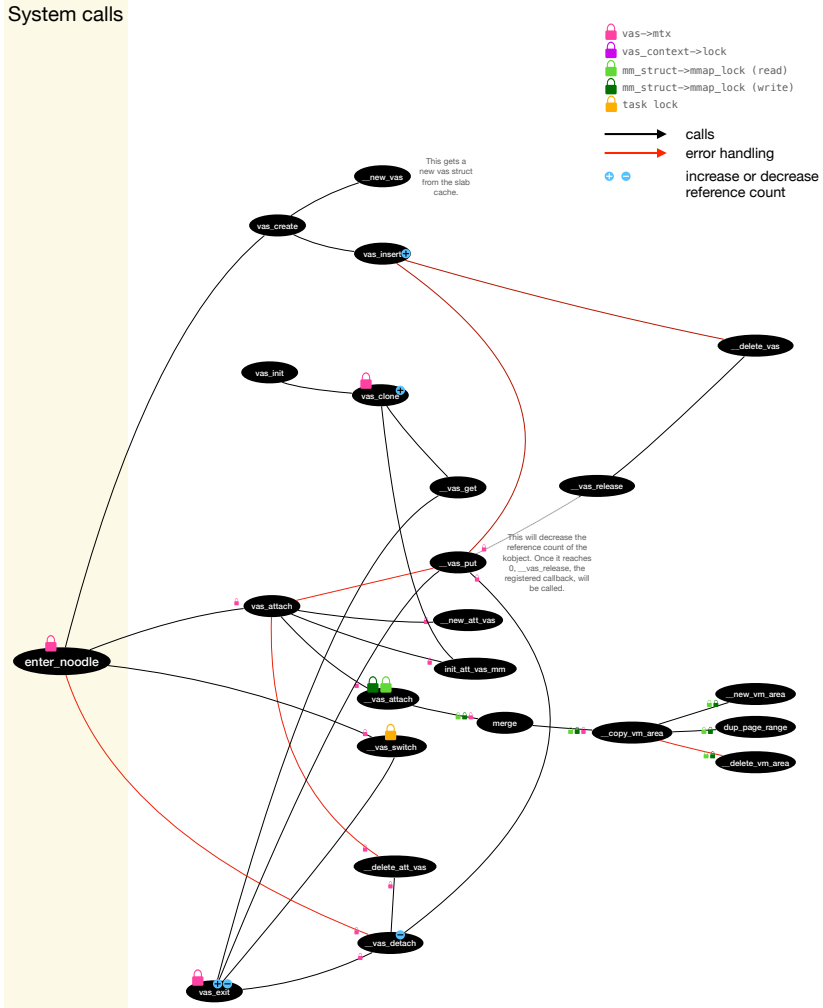


Figure 7.22: A call graph visualization of the code contained in file *vas.c* as well as in the accompanying header files, my own patch series.

7.4.7 enter_noodle

This subsection describes the single system call serving as the entry point and API for my implementation (a re-design of the original patch series [92] examined in previous subsections). See figure 7.23 for the signature of the new system call and the flags that can be passed as its first parameter. For an explanation of the different flags that can be asserted, see the list 7.4.7 below.

The second parameter of `enter_noodle` takes a list of virtual addresses that will be interpreted as the start addresses of a list of mappings. The addresses are passed as `void` pointers because `mmap(2)` returns such a pointer on success and this keeps the formats consistent but also because this is the least amount of data that can be passed to identify a mapping (short of identifying them by indices which is clearly undesirable). For performance reasons it is important to keep the amount of data that needs to cross the user space/ kernel space boundary as small as possible.

The third parameter is the number of virtual start addresses contained in the `addrs` list.

```

/* Flags for enter_noodle. */
#define VAS_SWITCH_NONE      0x00000000
#define VAS_SWITCH_ALL      0x00000001
#define VAS_SWITCH_NPWR    0x00000002
#define VAS_SWITCH_SO      0x00000004
#define VAS_SWITCH_LIN     0x00000008 /* include list */
#define VAS_SWITCH_LEX     0x00000010 /* exclude list */
#define VAS_SWITCH_MASK    0x0000001F

enter_noodle(int flags, void **addrs, int size);

```

Figure 7.23: Signature of `enter_noodle()`

Let us now continue with an overview of all the flags that can be passed to the `enter_noodle` system call via the first parameter. Note that due to the flags parameter actually being a bitmap, any combination of flags is possible – though of course not necessarily legal.

Flags of `enter_noodle`:

- **VAS_SWITCH_NONE**: Asserting this flag causes the default policy to be not to copy any mapping into the new virtual address space. This flag exists mainly for completeness sake and to combine it with others as it is not practical to exclude all mappings present in the original virtual address space on entering a *noodle* – though in combination with an *include list* using it is feasible.
- **VAS_SWITCH_ALL**: This flag corresponds to the original patch series's behavior. It causes all mappings present in the original virtual address space to also be included in the newly created virtual address space of the *noodle*. Note however that this is expensive as to prevent the problem of the diverging, partially paged in mapping, all mappings that are private and writable need to be pre-faulted.
- **VAS_SWITCH_NPWR**: This flag stands for *no private writable mappings* and – as its name suggests – it excludes all mappings which are difficult to handle when creating the *noodle*'s new virtual address space. This might not seem very useful since there are many private writable mappings that are crucial for an execution context (be it a thread or a *noodle*) – for instance the stack is one of these mappings. However, if used together with the **VAS_SWITCH_LIN** flag to pass an inclusion list it allows for great flexibility.
- **VAS_SWITCH_SO**: Asserting this flag will cause the system call to adopt a policy where only shared mapping will be copied into the virtual address space of the *noodle*.
- **VAS_SWITCH_LIN**: This flag must be asserted for the system call to interpret the passed list of virtual addresses as the start addresses of mappings that should be included *even if the chosen policy would otherwise exclude them*. This is very useful as it allows to make exception for specially chosen mappings such as e.g. the stack of a thread.
- **VAS_SWITCH_LEX**: If this flag is asserted, the passed list of virtual addresses is interpreted as an exclusion list instead: All mappings with start addresses in this list will be excluded from the *noodle*'s

virtual address space *regardless of* if the chosen policy would include them or not. This is again intended mainly as a mechanism to make exceptions.

- **VAS_SWITCH_MASK**: This flag allows to check the input, making sure no unknown flags are passed.

As mentioned above, it is intended for the caller to combine one of the policy flags (**VAS_SWITCH_ALL**, **VAS_SWITCH_NPWR**, **VAS_SWITCH_SO** and **VAS_SWITCH_NONE**) with either the **VAS_SWITCH_LIN** or the **VAS_SWITCH_LEX** flag. The idea is to select a general policy and then to make some adjustments by passing either an inclusion or an exclusion list to make exceptions to that policy.

How to call `enter_noodle` For completeness I give a very basic example how `enter_noodle` can be called here, though the more interesting examples will follow in the next section, section 7.5.

In principle `enter_noodle` can be called like any other system call, though the application should be aware that since the transformation of a thread into a *noodle* is irreversible, it can be beneficial to create a second thread first. This way, a normal thread can keep running in the original virtual address space if so desired.

Note also that *libnoodles*, the support library which I implemented and which is described in the next subsection, subsection 7.4.8, provides two wrappers for the system call. These take care of passing the right system call number to the `syscall(2)` function provided by `glibc`. The example in figure 7.24 uses one of these two wrappers (the other wrapper is a convenience function to make calling `enter_noodle` a bit less cumbersome though it offers the same functionality).

Figure 7.24 shows the most basic of usages: The policy for deciding which mappings to copy into the *noodle*'s virtual address space is **VAS_SWITCH_ALL** which will cause *Noodles* to copy all mappings present in the original virtual address space. Instead of a list of virtual start addresses `NULL` is passed which is allowed in the case that the third value passed is zero as here. This simply means not to make any exceptions to the policy chosen.

As is customary, `enter_noodle` returns 0 on success and a non-zero value otherwise.

```

int res = enter_noodle(VAS_SWITCH_ALL, NULL, 0);
if (res != 0) {
    fprintf(stderr, "enter_noodle: Failure.\n");
    return EXIT_FAILURE;
}

```

Figure 7.24: Example of calling `enter_noodle()`

7.4.8 libnoodles

In this subsection I describe the second part of my implementation, the user space support library called *libnoodles*. It provides two convenience wrappers for `enter_noodle` as well as functionality to manage mappings via a handy user space representation. The idea is that *libnoodles* is used in conjunction with a modified Linux kernel which has been patched to support *Noodles*.

In addition to the functionality mentioned above, *libnoodles* also provides testing and benchmarking functionality. The latter has been used to gather most of the data plotted and analyzed in the next section, section 7.5.

Wrapper for the `enter_noodle` system call As mentioned, *libnoodles* provides two different wrappers for the `enter_noodle` system call: The first of those two is very bare bones and mainly used for benchmarking, to make sure that cycle measurements are distorted as little as possible. Its signature is given in the code snippet 7.23 reproduced above and it accepts, next to a flags and a size parameter, a “raw” list of virtual start addresses.

Because dealing with addresses directly can be error prone, I have implemented a second wrapper for `enter_noodle` that provides the same functionality but that takes as input instead a list of mapping structs. See its signature in code snippet 7.25. It uses a conversion function also provided by *libnoodles* to convert from the list of mapping data structures to the required bare virtual addresses array. The other two parameters are unchanged.

```
int enter_noodle(int flags, struct mapping *list, int size);
```

Figure 7.25: Signature of the second convenience wrapper for `enter_noodle()`

```
enum MAPPING_TYPE {
    PRIVATE,
    SHARED
};

struct mapping {
    struct mapping *next;
    struct mapping *prev;
    char *pathname;
    unsigned long start_addr;
    unsigned long end_addr;
    unsigned long inode;
    unsigned long offset;
    enum MAPPING_TYPE type;
    char permissions[4];
    unsigned char device[2];
};
```

Figure 7.26: Definition of the `struct mapping` data type

Management of mappings *libnoodles* manages mappings with the help of the user space mapping abstraction given in code snippet 7.26. This data struct of course doesn't encapsulate all of a mapping's important state such as its kernel space "mirror image" `vm_area_struct` does. Instead, it is designed to encapsulate the information concerning mappings that is exposed by the kernel via the *procfs* filesystem (usually mounted as */proc*). *procfs* is a pseudo filesystem, just like *sysfs*, and exposes a host of information to a process (about itself) under */proc/self*. The file */proc/self/maps* lists all mappings present in the virtual address space of a process. A `struct mapping` of *libnoodles* is designed to encapsulate the information contained in one line of this file. See

figure 7.27 for an excerpt from this file.

```

user@machine:/proc/self$ cat maps
aee68af50000-aee68b099000 r-xp [...] 2885936 /usr/bin/bash
aee68b0a8000-aee68b0ad000 r--p [...] 2885936 /usr/bin/bash
aee68b0ad000-aee68b0b6000 rw-p [...] 2885936 /usr/bin/bash
aee68b0b6000-aee68b0c1000 rw-p [...] 0
aee6a2d82000-aee6a2f2a000 rw-p [...] 0 [heap]

[...]

ffffeef62000-ffffeef83000 rw-p [...] 0 [stack]

```

Figure 7.27: Excerpt of `/proc/self/maps`

As can be deduced from the output captured in figure 7.27, the process whose virtual address space is summed up by this file is a *bash* shell. The first three lines clearly show the executable being mapped into the virtual address space. The first two values of each line are the virtual start and end address of the mapping in question.

libnoodles has utility functions for parsing the `/proc/self/maps` file and storing the information read in form of a linked list of `struct mapping` data structs. This makes managing the virtual address space with `enter_noodle` much easier which is why the second wrapper, see code snippet 7.25 takes a list of such mapping structs as its input.

The benchmarking functions also take advantage of the mapping management functionality *libnoodles* offers as we will see in section 7.5. They call the `create_mappings` function to automatically populate or *pre-load* the virtual address space with n private anonymous mappings of a specific size to simulate differently loaded virtual address spaces.

7.4.9 Lessons learnt

To conclude this section, I want to point out some lessons learnt during the implementation process:

- Even conceptually simple, straightforward abstractions can be tricky to implement in a real-life system, especially if said real-life system

has a big code base with millions of lines of pre-existing code. Any such system will have its own abstractions and concepts baked in. Those are not only hard to impossible to change retroactively but may also be incompatible with new abstractions and concepts we wish to introduce. Because of too little awareness of how costly it can be to change inherited design decisions, the difficulty of implementing new abstractions and ideas in existing, non-trivial systems is often underestimated.

- The Linux kernel is built to a certain extent around the file abstraction. As a result *private anonymous* mappings are treated almost like 2. class citizens – or at least stepchildren. Though this is likely a carry-over from Unix which is famous for the “everything is a file” design concept [99], in situations where something cannot naturally be represented as a file this causes problems.
- Virtual address space management is tough because there are so many different cases to consider: Different types of mappings, private and shared, anonymous and file backed, with different permissions set. In addition, each possible case of adjoining, intersecting and disjoint virtual address ranges needs to be considered.
- To be able to handle the complexity of virtual address space management when implementing a new idea or abstraction it is crucial to focus on an actual use case, such as the lightweight serverless platform here. This allows to make simplifying assumptions, handling only the cases actually relevant for the chosen use case. If a prototype shows the benefits of a new idea/ abstraction it is always possible to then extend support gradually at a later state.
- Scientifically speaking many of the issues that arise when working on non-trivial real-life systems such as incompatible Linux kernel versions or the difficulty of deriving a configuration file that allows to build a functioning kernel in a reasonable amount of time are considered trivial. In practice however, these things do matter and can eat up an astonishing amount of research time.

7.5 Evaluation

This section is devoted to the evaluation of the *Noodles* prototype presented in depth in the previous section, section 7.4. The evaluation is both an empirical one with micro benchmarks counting how many cycles are spent as well as one by example, demonstrating how *Noodles* can be used to achieve intra-process-isolation in a server application.

I show that when pairing `enter_noodle` with `clone(2)` to create a new execution context – a *noodle* – which only has access to a subset of the mappings present in the original virtual address space, it incurs only a fraction of the cost (10%-11%) compared to combining `fork(2)/clone(2)` with `madvise(2)` to achieve the same result. I also show that `fork(2)/clone(2)` is extremely expensive and this just becomes worse as the number of mappings in the virtual address space grows (and with the number of mappings the amount of physical memory mapped into the virtual address space).

In this section I compare and evaluate different ways of creating a new execution context on Linux: On one side of the spectrum we have the *thread* which is lightweight but does not provide any meaningful form of isolation. On the other side we have the *process* which is heavyweight but provides strong isolation (in comparison with a thread, of course there are even stronger forms of isolation but these are also even more heavyweight – by an order of magnitude – e.g. the virtual machine). In between we have `enter_noodle` used in combination with `clone(2)` (to first create a new thread) which gives us a form of intra-process-isolation which is meaningful without making the operation of creating the new execution context to heavyweight. Note that lightweight and heavyweight are used in this section as terms to classify how much overhead an isolation method incurs (be it intra- or inter-process-isolation) compared to running the same workload without any isolation.

In the course of this section I benchmark different system calls used to create a new execution context: I benchmark the traditional `fork(2)` by calling glibc's wrapper for `fork(2)`. I also call `clone(2)` directly and pass different flags to instruct it to behave either just like `fork(2)` or to create a thread instead. In addition, I micro benchmark `madvise(2)` which can be used in conjunction with `fork(2)/clone(2)` to create new execution contexts which only have access to a subset of the virtual address space their parent has access to. This allows to compare directly

against the combination of `clone(2)` and `enter_noodle`: `enter_noodle` itself I evaluate empirically with several micro benchmarks which call it repeatedly in different scenarios and with different flags.

This section is structured as follows: First I present a series of micro benchmarks evaluating the cost of the central system calls `fork(2)`, `clone(2)` and `madvise(2)` to establish a baseline. I then show results for a series of micro benchmarks comparing the same system calls against `enter_noodle` in different usage scenarios. Finally I present an actual example application based on OpenSSI's echo server demo [50], [51]. I augmented the echo server demo with *Noodles* and demonstrate how an attack inspired by the well known *Heartbleed* vulnerability [23] can be foiled. I also benchmark a simple example HTTPS server based on the same demo with Apache benchmark [2] to show it's feasible to use *Noodles* in a real application.

But before we delve into the benchmarks themselves, I want to highlight several challenges inherent to benchmarking system calls in the next subsection, subsection 7.5.1. These challenges motivate the micro benchmarking setup.

7.5.1 Challenges inherent to benchmarking system calls

In this subsection I want to highlight why benchmarking system calls is not quite the same as benchmarking normal functions which run exclusively in user space: For a start, system calls by their very nature cross over into kernel space⁶. This is problematic for two reasons: Firstly, while in user space each process has its own virtual address space with its own (mostly) private data. (A mapping that is shared is normally the exception and especially set up to be shared.) Even in the presence of multiple threads most data will in practice only ever be accessed by a single execution context at a time: Shared data like a global variable which will be accessed concurrently is even in a multithreaded scenario usually limited to a few key pieces of data.

This is essentially reversed in kernel space. Here by contrast all non-local (not on the stack) data structures are shared between all tasks *per*

⁶The only exceptions to this are system calls like `gettimeofday(2)` which has been optimized to avoid crossing into kernel space by instead making use of vDSO [44].

default. There is only one kernel context so to speak and virtually all data structures have to be designed to be *thread-safe* (or rather task-safe). As a consequence, code running in kernel space might be subject to *jitter* caused by locks and atomic operations to a greater degree than is the case for most applications. Also, in contrast to benchmarking a multithreaded application where there may also be variation due to lock contention etc. in kernel space the other tasks contending for the same lock may not be under the control of the programmer doing the benchmarking.

The second reason why a function being benchmarked which partially executes in kernel space is problematic has to do with scheduling: On the return code path to user space of any system call there is located what is called a *scheduling point*. This is the point where the Linux kernel's scheduling logic runs and potentially decides to *suspend the current task* (depending on a number of factors such as the scheduling policy, the time the task has already run, its niceness value etc.) [33]. Of course this is a worst case scenario for benchmarking: If at all possible we do not want to be suspended while running code that is to be timed since it distorts our measurement.

As a last point, it is important not to forget that part of the cost of `fork(2)/clone(2)` is hidden (or deferred): While `fork(2)clone(2)` copies (almost) all of the page tables eagerly on forking as we saw in section 7.3 all memory mapped as private is of course only copied when it is modified later on because of COW. The hope is that if neither the child nor the parent later writes to a certain page, we can forgo copying it altogether. But of course, depending on the actual access patterns, that hope may be false and a page may have still have to be copied later on. A cost to which is added the cost of a page fault. All of this means that it is not straightforward to estimate the actual cost of `fork(2)/clone(2)` since it essentially depends on the workload and may not be paid all at once. This is of course less than ideal for benchmarking.

7.5.2 Benchmarking setup: Machine, benchmarks

After having discussed the challenges inherent in benchmarking system calls in the previous subsection, subsection 7.5.1, I now describe the benchmarking setup in this subsection.

The benchmarking machines To minimize jitter to the best of my ability, the benchmarks were run on bare metal, not in a virtual machine hosted by any cloud provider. The modified 5.15.116 Linux kernel was installed directly on a server class AArch64 machine. It has a 48-core Marvell Cavium ThunderX-1 ARMv8 CPU clocked at 2 GHz, running a stock Ubuntu 22.04 LTS distribution. The CPU has 128 GiB RAM.

The benchmarks were also run on a second machine (the kernel was also installed directly, the distribution used was a stock Ubuntu 22.04 LTS) to be able to compare to x86-64: The machine used is an x86-64 server with 2 AMD Ryzen 9 7950X 16-core processors (32 cores in total) and 64 GiB RAM.

Porting to AArch64 I originally implemented `enter_noodle` as an Intel x86-64 system call but later also ported it to AArch64 for practical reasons. This was straightforward as the Linux kernel is structured in a way that allows even page table managing code to be written in an architecture independent way: This is accomplished by having the different supported architectures each implement their own version of a function/-macro to e.g. write a pte. Because of this, porting `enter_noodle` to AArch64 didn't involve much more than also registering the new system call for the AArch64 architecture.

Measuring cycles I chose to measure elapsed time by counting CPU cycles. I measured these on the AArch64 benchmarking machine with the following instead of Intel's usual `rdtsc` instruction:

```
__asm__ __volatile__(
    "MRS %0, CNTVCT_ELO; ISB"
    : "=r" (value)
);
```

This is inspired by what the Linux kernel itself uses instead of the usual `rdtsc` instruction when running on AArch64. Note that the code given above is for starting the clock, the placement of the barrier varies depending on if we are starting or stopping the clock. For stopping it the barrier needs to be placed *before* reading `CNTVCT_ELO` to make sure nothing is reordered past us “stopping the clock”.

Description of the *libnoodles* micro benchmarks The support library *libnoodles* implements several different micro benchmarks. These are designed to measure how long it takes to create a new execution context and to restrict it to a subset of the original virtual address space. Some measure the well-known `clone(2)` in different configurations to establish baselines, others measure the system call `enter_noodle`.

To measure how the amount of memory mapped into the virtual address space influences the performance of the different system calls, the benchmarking process's virtual address space is preloaded with n mappings during the set-up step. Their number and size is then varied over different runs of the micro benchmarks. See below an overview of all the different micro benchmarks:

- *fork*:
This micro benchmark calls glibc's wrapper for `fork(2)` in a loop. The clock is started immediately before forking and stopped first thing in the child. This defines the cost of forking as the number of cycles until the child can execute instructions provided by the programmer.
The parent waits for each child to finish before starting the next iteration of the benchmark. This is done so that the number of processes running simultaneously on the system doesn't explode as the benchmark progresses. Otherwise, this either causes enormous amounts of jitter or – even worse – may lead to `fork(2)` suddenly failing due to momentarily exhausted resources.
- *clone (fork mode)*:
This micro benchmark calls glibc's wrapper for `clone(2)` instead and measures how many cycles `clone(2)` takes to fork a child process. To have it behave like `fork(2)`, the flag `SIGCHLD` is passed. As with *fork* the clock is stopped as soon as the child is able to execute the first programmer provided instruction.
- *clone (thread mode)*
This micro benchmark measures how many cycles `clone(2)` takes to create a new thread instead of a child process. The flags passed are:

```
#define CLONE_THREAD_FLAGS (CLONE_VM | \
```

```

CLONE_FS | \
CLONE_FILES | \
CLONE_SIGHAND | CLONE_THREAD | \
CLONE_SYSVSEM | \
CLONE_PARENT | \
CLONE_VFORK | \
SIGCHLD)

```

Note that even though glibc calls `clone(2)` internally to create a *pthread*, calling `clone(2)` directly will result in a thread of which glibc isn't aware.

- *fork subset* This micro benchmark is equivalent to the *fork* micro benchmark except that before running it, during the set-up step, the mappings preloaded into the virtual address space are also marked as `MADVISE_DONTFORK`. This lets `fork(2)` know not to copy these mappings into the child's virtual address space.
- *advise*:
This micro benchmark is the counterpart to the *fork subset* benchmark: The *advise* benchmark measures how long it takes to call `advise(2)` *n* times (to mark *n* different mappings either as `MADVISE_DONTFORK` or `MADVISE_DOFORK` – the micro benchmark toggles between these two). This is to measure the otherwise hidden cost incurred during the set-up step of the previous micro benchmark, *fork subset*.
- *advise + fork subset*
This corresponds to an aggregate of the two previous micro benchmarks *fork subset* and *advise*, summed up for the readers convenience. (They were added data point by data point.)
- *noodle prefault*:
This micro benchmark turns a thread (created by calling `clone(2)` before starting the clock) into a *noodle* by calling `enter_noodle`. The flags passed are `VAS_SWITCH_ALL` and `VAS_SWITCH_LEX`. Instead of an exclusion (or inclusion list) simply `null` is passed. This plus the chosen policy will cause *Noodles* to prefault all *private writable* mappings to prevent them from diverging later on.

As expected, defaulting every *private writable* mapping is very expensive even though we warm up this benchmark (the same way as all the other ones), meaning that the cost is simply caused by *checking* that all pages are linked in the page tables.

- *noodle subset*:

This micro benchmark was designed to be as equivalent as possible to the *fork subset* benchmark. `enter_noodle` is again called with the `VAS_SWITCH_ALL` and `VAS_SWITCH_LEX` flags but this time all n mappings installed in the set-up step are passed as the exclusion list. The result is a *noodle* (instead of a child process as with *fork subset*) that has a virtual address space with a subset of the mappings installed that were present in the original virtual address space.

Note that all *private writable* mappings that are installed before the set-up step (e.g. the mapping containing the stack of the thread, the heap, etc.) still need to be defaulted.

- *noodle npwr*:

Lastly, this micro benchmark plays to the strengths of *Noodles*: This time `enter_noodle` is called with the `VAS_SWITCH_NPWR` and `VAS_SWITCH_LIN` flags. This causes the policy to be not to include any *private writable* mappings in the virtual address space of the *noodle* per default. This makes creating a *noodle* a very fast operation but excluding all *private writable* mappings without any exceptions would result in an unrunnable *noodle*. To still create a viable execution context we therefore make an exception for the mapping containing the stack of the thread that is to be turned into a *noodle*. This single mapping then is defaulted by *Noodles* to prevent it from diverging.

Note that just making an exception for the mapping containing the stack is enough to run our benchmarking code but of course depending on the code that needs to be executed more exceptions might have to be made. As long as the number of exceptions is small, this is still going to be efficient.

Preloading the address space Before actually starting a specific micro benchmark *libnoodles* makes sure to pre-load the virtual address

space of the benchmarking process with n *private anonymous* mappings. They are both read- and writable. This is to simulate different workloads resulting in differing amount of memory being mapped into the virtual address space.

Preloading is done by using *libnoodles*'s `create_mappings` function which internally calls `mmap(2)`. `create_mappings` takes n as a parameter and then creates n non-consecutive mappings of a given size (defined in a header file of *libnoodles*). These newly created mappings are exactly 4KB (one page) apart (with nothing mapped in between them) to prevent them from being coalesced into a single mapping by the Linux kernel. Per default `create_mappings` creates mappings which are 1 page (4KB) big but I have also re-run the benchmarks with bigger sizes (8KB, 32KB and 64KB).

Touching all mappings After creating the n mappings, another function of *libnoodles*, called `touch_mappings`, is used to “touch”, meaning write at least one byte to, all pages of the just created mappings. This is done to make sure they are actually backed with physical pages and the corresponding p4ds, puds, pmds and ptes are all allocated. After this we can be sure that the page tables which need to be copied are fully populated which is an important factor when benchmarking `fork(2)clone(2)`. (Note that we do not care if something is swapped out as this will not shrink the page tables themselves.)

Number of runs and warm-ups Note that for each data point in the graphs following in the next subsections the benchmark in question – after preloading the virtual address space with n mappings – was run 1000 times after first warming it up with 500 extra warm-up runs. What is ultimately plotted is the median of the cycles counted for each of the 1000 runs. In addition, the 95th percentile error bar is also plotted. This is done to get reliable numbers with as little jitter as possible though it still reappears in the graphs showing the numbers for the bigger mapping sizes as the variance becomes so great.

7.5.3 Baselines

In this subsection I present the graph plotting only the baseline benchmarks, see figure 7.28. It shows the performance of a few important system calls, namely `clone(2)` (running in *fork* mode), `clone(2)` (running in thread mode), glibc's `fork(2)` and `madvise(2)`.

The benchmarks in question were chosen to first establish a frame of reference, characterizing the system's baseline performance when creating new execution contexts. To the baselines shown in figure 7.28 we later, in the next subsection, compare the different benchmarks calling `enter_noodle`.

There are 6 different micro benchmarks plotted in figure 7.28, with n , the number of mappings installed during the set-up step, varied along the x-axis. The size chosen for the preloading mappings is 4KB.

As can be seen in figure 7.28, unsurprisingly, `clone(2)`, when it isn't working in forking mode but only creating a thread, performs much better than when it actually has to create a whole new process (a child process). This difference in cost is only accentuated as n increases from left to right on the x-axis and the number of mappings present in the virtual address space prior to calling `clone(2)` grows. While the line corresponding to *clone (thread mode)* is constant with regards to the number of additional 4KB mappings installed in the virtual address space, *clone (fork mode)*'s cost increases linearly.

This is as expected since creating a new thread is, as we saw in section 7.3, a much more lightweight operation: For one it doesn't involve creating a new `mm_struct` but instead simply creates a new `task_struct` and has its `mm` field point to the already existing `mm_struct` belonging to the calling task. Another reason for the difference in performance is that forking causes `clone(2)` to duplicate the entire page tables of a process.

That cost grows from the left to the right because more mappings present in the virtual address space also mean (as long as they have been paged in at least once) more memory is needed to store the whole page table tree. In consequence it becomes more costly to copy the entire page tables as the number of mappings present in the virtual address space grows. Strikingly, even though *clone (fork mode)* is so expensive, this benchmark does not account for any of the cost incurred when page faults suffered later on trigger the duplication of written to pages (COW).

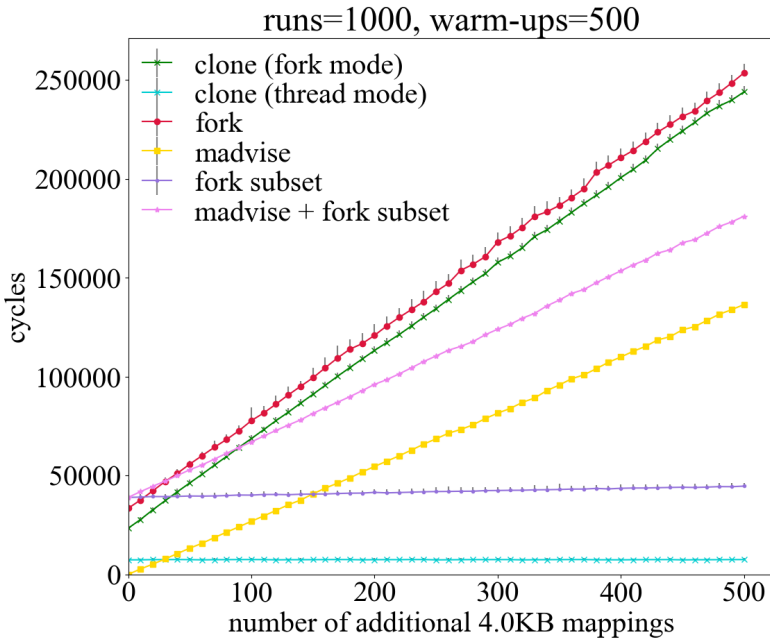


Figure 7.28: **The cost of forking:** This graphs plots baseline micro benchmarks run on the AArch64 machine. Each data point is the median of 1000 runs with 95th percentile error bars. The number of **4KB** mappings used to preload the virtual address space is varied on the x-axis.

Let’s now compare *fork* to *clone (fork mode)*. Those two benchmarks performed very similarly, the slight but constant overhead visible is likely caused by glibc: For the *fork* benchmark we call its usual wrapper function while with the *clone (fork mode)* benchmark we call glibc’s `clone(2)` wrapper instead.

Next, we compare *fork* and *fork subset*. Remember that for the latter all the additional n 4KB mappings installed in the virtual address space during the preloading phase have been marked as `MADVISE_DONTFORK` be-

forehand. Otherwise the setup for those two benchmarks is identical. As we can see, marking the the additional n mappings as `MADVISE_DONTFORK` has a huge effect: Due to excluding the additional mappings the cost of calling `fork(2)` essentially remains constant, keeping the line corresponding to the *fork subset* benchmark flat. This makes sense if one considers that due to excluding all additional mappings the amount of memory that needs to be copied when duplicating the page tables also remains constant.

Note that if you look very closely at the line for *fork subset* you can see that from the left to the right here *is* a very slight increase. However, it amounts to just a few cycles more per additional mapping which can be explained by the fact that the kernel has to check each mapping on forking to determine if a mapping needs to be forked or not.

Now, while marking mappings which won't be needed by a child process anyway might seem like a good strategy to reduce the cost of `fork(2)`, unfortunately there is a drawback: Calling `madvise(2)` n times to mark n mappings as *don't fork* – the cost of which can also be seen in the graph, it is the line labeled *madvise* – is also quite expensive and of course the cost increases the bigger n gets. Note that it is not possible to mark n non-contiguous mappings with a single `madvise(2)` call: Attempting to call it on a range with gaps in it will cause it to fail.

Lastly, compare the line adding the cost of the *madvise* and the *fork subset* benchmarks to the *fork* benchmark. As can be seen, though the cost of *madvise + fork subset* increases as the number of mappings preloaded into the virtual address space grows, simply calling `fork(2)` without any marking of mappings is even more expensive.

To conclude the analysis of figure 7.28, it is clear that `clone(2)/fork(2)` is a very expensive system call. The main factor driving up the cost is – perhaps surprisingly – not having to copy the physical pages mapped into the virtual address space holding a process's data but having to copy the *page tables* actually mapping that physical memory.

Further, it can also be seen that limiting the virtual address space of a child process to a subset of the parent's original one is an effective measure to reduce the cost of `fork(2)`. However, this only helps partially since having to mark all the mappings to be excluded with `madvise(2)` is still quite expensive.

7.5.4 Analysis of results

In this subsection I analyze the data gathered by running the *Noodles* related micro benchmarks on AArch64. Some selected baselines are also re-plotted in the graphs for comparison. (Note that all baseline benchmarks were run in sequence with the *Noodles* benchmarks presented here to keep the comparison as fair as possible.)

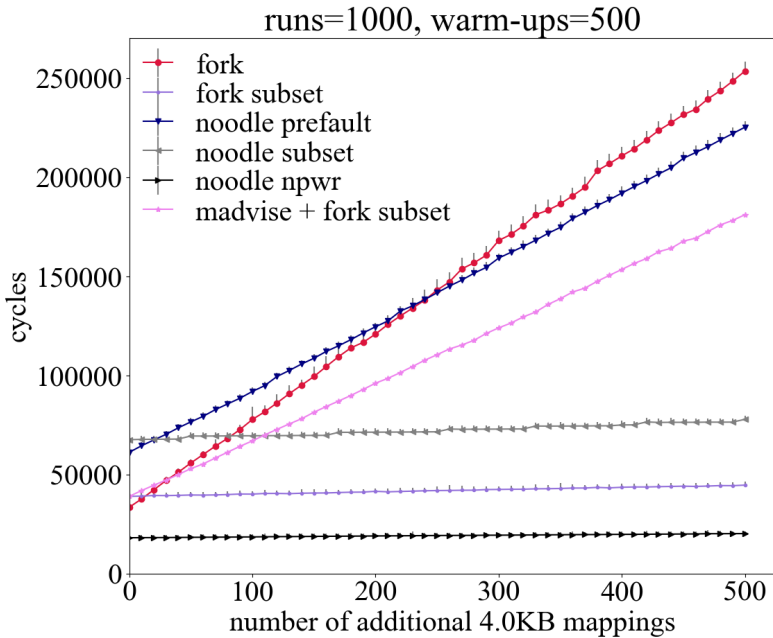


Figure 7.29: **Noodles vs. fork:** This graphs plots the *Noodles* micro benchmarks as well as some of the baseline benchmarks from figure 7.28. Each data point is the median of 1000 runs on the AArch64 benchmarking machine with 95th percentile error bars. The number of **4KB** mappings used to preload the virtual address space is varied on the x-axis.

Micro benchmarks shown The graph in figure 7.29 shows the results for 6 different micro benchmarks: *fork*, *fork subset* and *madvise + fork subset* which we've already saw in figure 7.28 as well as three new ones: *noodle prefault*, *noodle subset* and *noodle npwr*. Those measure the cost of the new `enter_noodle` system call in three different scenarios.

4KB mappings Let's now analyze the graph shown in figure 7.29: As we can see (and as expected) *noodle prefault* is extremely costly – even more costly than *fork* in fact, at least up to a point. Interestingly enough, as n is increased along the x-axis, *fork* regains its place as the most costly micro benchmark. This happens at about 250 additional 4KB mappings preinstalled in the virtual address space. I speculate that this is due to both micro benchmarks (*fork* and *noodle prefault*) having to make a full copy of the page table tree but *fork* (in addition to having to copy state unrelated to the virtual address space, a cost which should be constant for our purposes) also having to change the permissions on the ptes corresponding to private writable mappings because of COW. Alternatively *noodle prefault* might have better cache locality due to copying the page table tree after having just walked it to prefault it.

Next, I compare *fork subset* to *noodle subset*. Here we can see that both these micro benchmarks have (near) constant performance as the number of additional mappings present in the virtual address space grows. *noodle subset* is more expensive than *fork subset*. This constant difference can be explained by the fact that *noodle subset* still has to prefault all private writable mappings that were already present in the virtual address space before the preloading step. That includes the mapping containing the stack but also the heap, mappings holding data belonging to glibc etc. See figure 7.20 in the last section, section 7.4 for more details on which mappings are present in a virtual address space.

However, when comparing *fork subset* to *noodles subset* it is important to keep in mind that the cost for marking the mappings before forking also has to be counted. This cost, caused by having to call `madvise(2)` n times is included in *madvise + fork subset*. At 110 additional 4KB mappings, the lines for *noodles subset* and *madvise + fork subset* cross. From that point onwards, the latter is more expensive.

This makes sense as the cost for *madvise + fork subset* is (partly) dependent on n with having to call `madvise(2)` n times while *noodles*

subset has to prefault a hand full of *private writable* mappings which is while costly also a constant cost factor. (Having to pass more mapping start addresses in the exclusion list is by comparison negligible though also slightly increases in cost as n grows.)

Lastly, let's examine the performance of the *noodle npwr* micro benchmark. It's cost also remains constant as we increase n as we can see in figure 7.29. Again, this makes sense since the number of mappings that need to be prefaulted is constant (just one, the mapping containing the stack whose start address is passed as the inclusion list). Also, with the policy excluding *private writable* mappings per default we don't have to pass all additional n mappings' start addresses as part of the exclusion list. As a result *noodle npwr* hits a performance sweet spot so to speak: It combines a low base cost with (nearly) constant performance as the amount of memory mapped into the virtual address space grows from the left to the right.

Conclusion To summarize, the cost of calling `enter_noodle` very much depends on the concrete makeup of the virtual address space: To be precise, it matters most how many *private, writable* mappings from the original virtual address space need to be copied into the *noodle*'s new virtual address space and therefore need to be prefaulted.

`fork(2)` has clearly been shown to be a very costly system call, a cost that can only partly be alleviated by marking unneeded mappings as `MADVISE_DONTFORK` beforehand since the cost to call `madvise(2)` repeatedly is also non negligible. To quantify the difference in performance, *noodle npwr* incurs on average only 11% of the cost incurred by *madvise + fork subset* for 500 additional 4KB mappings.

Bigger mappings After gathering the data presented in the last two graphs, figure 7.28 and figure 7.29, I re-ran all micro benchmarks on the AArch64 machine with bigger mappings used to preload the virtual address space: The sizes used were 8KB, 16KB, 32KB and 64KB. The results of this can be seen in the figures figure 7.30a, figure 7.30b, figure 7.33a and figure 7.33b.

If we look at the case of 8KB mappings, where the virtual address space is preloaded with n mappings of double the size used before, we can clearly see that the resulting graph is very similar to the ones pro-

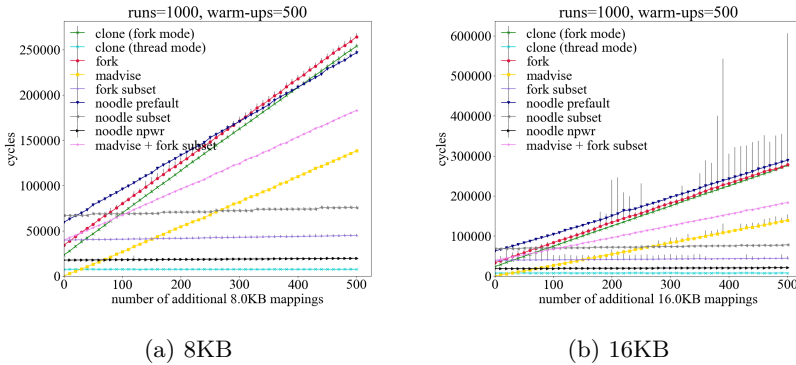


Figure 7.30: **Similar results for 8KB and 16KB:** This graphs plots the *Noodles* micro benchmarks that were on AArch64 with mapping sizes **8KB** and **16KB**. Each data point is the median of 1000 runs with 95th percentile error bars. The number of mappings used to preload the virtual address space is varied on the x-axis and increases from the left to the right.

duced for the 4KB case (compare with figure 7.28 and figure 7.29). The only real difference is that for *noodle default* the cost per additional preloading mapping is a bit higher: The line corresponding to *noodle default* crosses the line for *fork* only at the 300 additional mappings mark instead of already at about 250 additional mappings as was the case for figure 7.29.

We now move on to the graph visualizing the data collected for the 16KB case, see figure 7.30b. One thing that is immediately striking is that *fork* now shows a lot of jitter – and it gets worse as n is increased from the left to the right. To make sure that this wasn't caused by interference on the benchmarking machine, I actually ran the entire benchmarking suite for the 16KB case twice. However, this problem persisted and it always affected the *fork* benchmark series. The cause for this is unknown as it didn't affect the *clone (fork mode)* benchmark series. (This can be seen more clearly in the figure 7.31b, further below, where the results for the *clone (fork mode)* micro benchmarks for different mapping sizes are compared.)

Also interesting is that *noodle prefault* is now always more expensive than *fork* though the difference becomes smaller as n is increased. This backs up what could already be seen in the previous graph in figure 7.30a: As the size of the preloading mapping increases, the cost per additional mapping goes up too. That is to be expected since a bigger mapping will span more *ptes* and thus it will be more expensive both to copy the pages tables and to prefault them.

What is a bit hard to make out due to different y-axis scales is how the different micro benchmarks compare against themselves, e.g. how *fork* from the 4KB data set compares against *fork* from the 16KB data set. Because of this, I created separate comparison plots that show only a single micro benchmark each but plot it for 4KB, 16KB, 32KB and 64KB. See figure 7.31a for a graph that shows how the performance of `fork(2)` changes as the size of the preloading mapping is increased. As can be easily made out is that `fork(2)` incurs a higher cost per additional preloading mapping as the size of it is increased, just like *noodle prefault*. The effect just isn't quite as pronounced.

It makes sense that *fork* also incurs a higher cost per additional mapping as the size of the mapping increases. Even though it doesn't have to prefault it, it still has to copy the page tables. This means that the same argument applies as with *noodle prefault*, the bigger the mapping and thus the more *ptes* are spanned, the higher the cost per additional mapping. See also figure 7.31b and figure 7.31c showing the results for *clone (fork mode)* and *noodle prefault* to observe this effect.

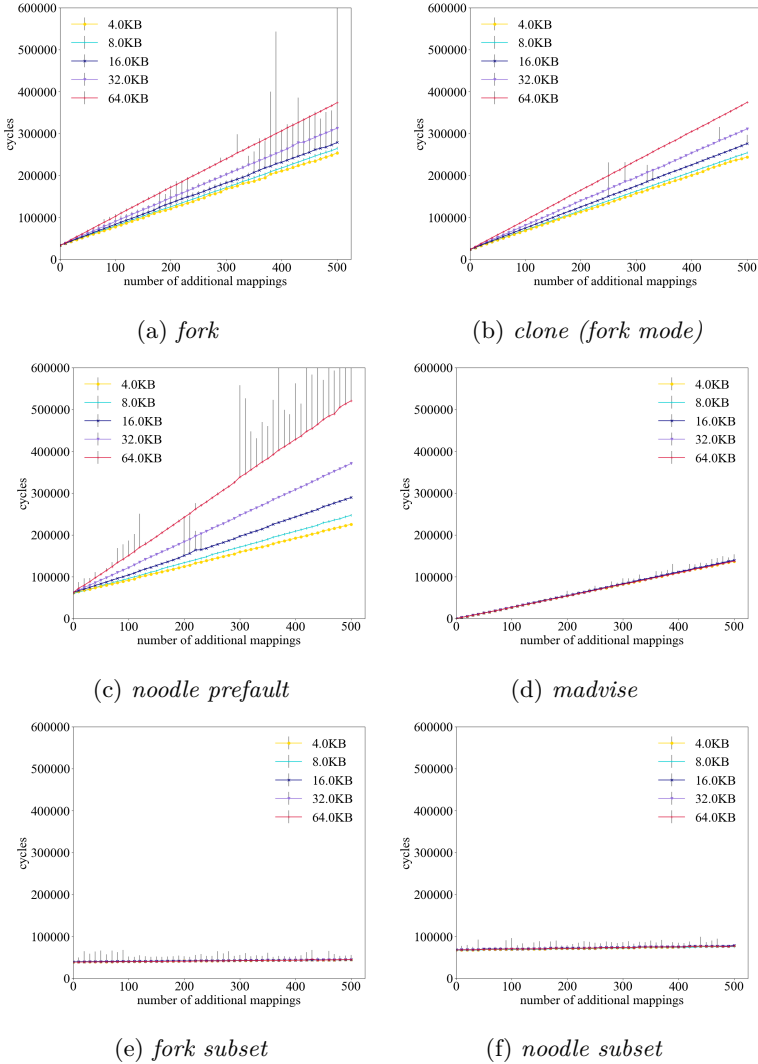


Figure 7.31: **Compare mapping sizes:** These graphs plot the micro benchmark results gathered on AArch64 for different mapping sizes, grouped by micro benchmark. Each data point is the median of 1000 runs with 95th percentile error bars. The number of preload mappings is varied on the x-axis.

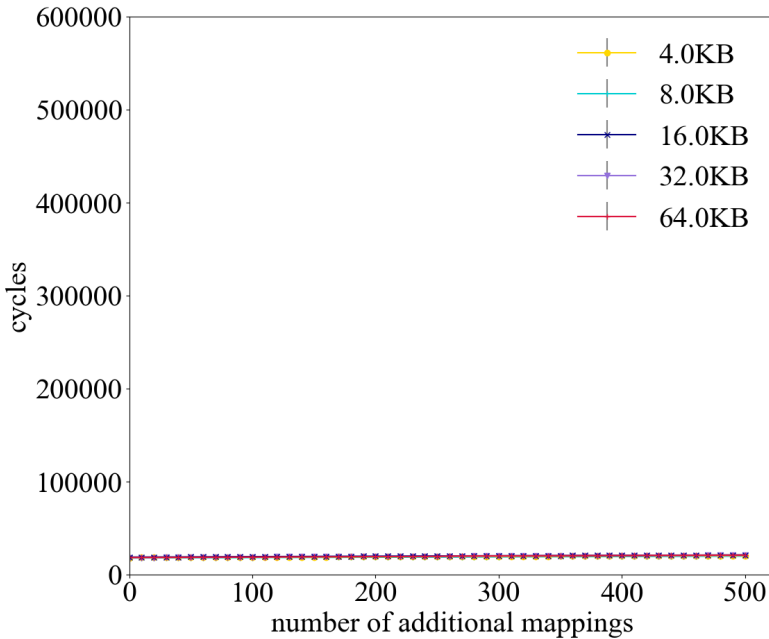


Figure 7.32: This graphs compares all the *noodle npwr* benchmark series. Each data point is the median of 1000 runs with 95th percentile error bars. The number n of mappings used to preload the virtual address space is varied on the x-axis and increases from the left to the right.

On the other hand, *madvise*, *fork subset*, *noodle subset* and *noodle npwr* are all unaffected by an increased preloading mapping size. This can clearly be seen in the figures figure 7.31d, figure 7.31e, figure 7.31c and figure 7.32 respectively. Note that the upper limit of the y-axis was set to 600000 cycles for all these graphs for ease of comparison.

For completeness's sake, figure 7.33a and figure 7.33b show the results of using 32KB respectively 64KB as the size for the preloading mappings with the micro benchmark plots again grouped by mapping size. Especially figure 7.33b emphasizes what we have already observed: As the size of the preloading mapping grows, the cost of having to copy

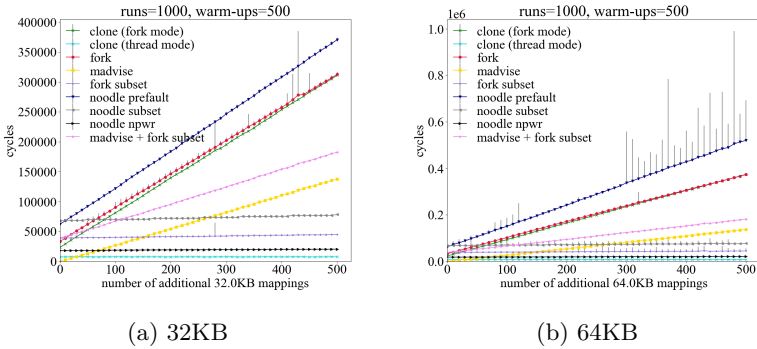


Figure 7.33: **Results for 32KB and 64KB:** These graphs plot the *Noodles* micro benchmarks run on the AArch64 machine for mapping sizes 32KB and 64KB. Each data point is the median of 1000 runs with 95th percentile error bars. The number of mappings used to preload the virtual address space is varied on the x-axis and increases from the left to the right.

the corresponding page tables of a mapping and then also to prefault them (walking the page table tree to make sure everything is paged in) starts to dominate more and more.

x86-64 For comparison's sake the whole micro benchmark suite implemented as part of *libnoodles* was also run on a x86-64 server class machine. Figure 7.34 shows the results of this with the size used for the preloading mappings being set to 4KB (figure 7.34a), 8KB (figure 7.34b), 16KB (figure 7.34c), 32KB (figure 7.34d) and 128KB (figure 7.34f).

The most notable difference that can be observed is that relatively speaking *fork* and *clone* (*fork mode*) are much more expensive on this machine than *noodle prefault*. To give a concrete example, on average, for 500 4KB mappings preinstalled in the benchmarking process's virtual address space, *noodle prefault* incurs only about 30% of the cost of *fork*. This suggests that walking the page tables to check that all physical pages are linked into the virtual address space is much more expensive on AArch64 than on x86-64. This is given credence by the fact that on x86-64, *fork subset* is more expensive than *noodle subset* (the result was

the opposite for AArch64).

Finally, *noodle npwr* incurs about 10% of the cost of *fork subset + madvise* which is very similar to the 11% observed on AArch64.

It's also worth pointing out that the reason that the plots are noisier and show more variance is that our AArch64 machine's processors do much less reordering of instructions than the processors of our x86-64 machine. (This makes the AArch64 machine especially good for benchmarking.)

Note It is important to point out here that even though I've conducted this investigation into how `clone(2)/fork(2)`, `madvise(2)` and the new `enter_noodle` are affected by the amount of memory mapped into the virtual address space being increased up to $64 \text{ KB} \times 500 = 32000 \text{ KB} = 31.25 \text{ MB}$, most virtual address spaces tend to have only a handful of mappings installed.

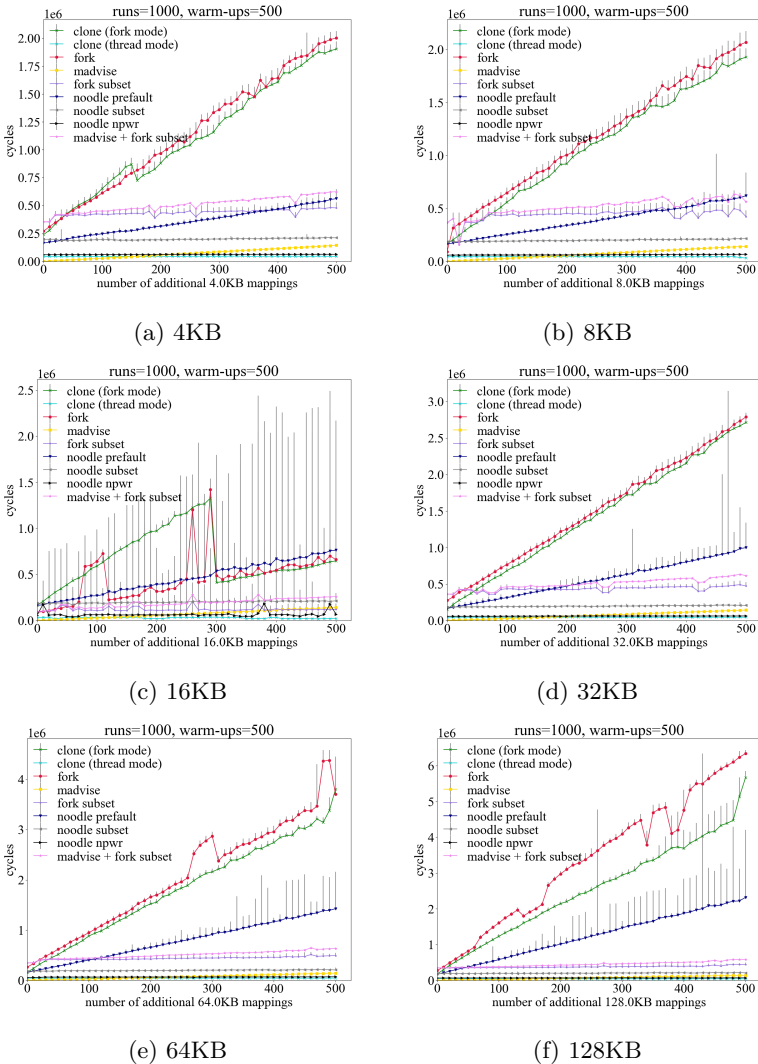


Figure 7.34: **Micro benchmark results for x86-64:** These graphs show the results for running the same micro benchmarks as before but on an x86-64 machine this time (my colleague Roman Meier assisted with this). Each data point is the median of 1000 runs with 95th percentile error bars. The number of preloading mappings used is varied along the x-axis.

7.5.5 Echo server use case

In this subsection I evaluate *Noodles* with the help of an example application after the evaluation with micro benchmarks of the previous subsection, subsection 7.5.4.

To prove that *Noodles* can indeed be used to provide intra-process-isolation, I used `enter_noodles` to protect an example application: I first modified OpenSSL’s echo server demo [50], [51] to introduce an artificial vulnerability, inspired by the well known Heartbleed vulnerability [23]. This vulnerability I was then able to successfully exploit in the unprotected application. I got the server to send my client the contents of a supposedly “secret” file stored on the server. Finally, I modified the application again to take advantage of *Noodles*, successfully preventing the disclosure of sensitive data.

I’ve also implemented a simple HTTPS server application (also based on the echo server demo) which sequentially serves GET requests. This application I then benchmarked with Apache benchmark [2], presenting the results of this at the end of this subsection.

The setup I conducted my first experiment on a virtual Ubuntu 22.04.4 LTS machine, running on an AArch64 processor. The kernel installed was the Linux kernel version 5.15.116, modified to support *Noodles*.

As the target application for my experiment I selected OpenSSL’s echo server demo. I chose it since it is readily available and a server connected to multiple mutually distrusting clients makes sense as a use case for intra-process-isolation. At the same time the echo server being a demo application this allowed me to quickly retrofit it with *Noodles* for the second part of the experiment.

In its unmodified form the echo server simply listens on a socket until a client connects to it. It then tries to establish an SSL connection with that client. If successful, the client (which is also included in the echo server demo) then sends the server a simple string message which the servers echos back to it. (Hence the name echo server.)

The “bug” To test if *Noodles* can be used to prevent sensitive data disclosure in the event that an application exhibits a vulnerability similar to the well known Heartbleed bug [23], I introduced an artificial “bug” in the demo application. Note that due to time constraints the vulnerability

introduced is not an exact replication of the Heartbleed bug but instead inspired by it in spirit: A successful exploit also causes the server to read past the end of an internal buffer and thus inadvertently send the client potentially sensitive data.

I implemented my artificial bug by having the server accept not only a message to later echo back but also (though somewhat contrived) a size parameter, telling the server how many bytes it should send back to the client. I then modified the server's `per_connection` function to “mistakenly” use this size parameter sent by the client to determine how many bytes to read from the internal receive buffer to echo back (instead of the (correct) receive buffer length variable, called `rxlen`).

In a next step I used `mmap(2)` to map my “secret” target file right after the server's receive buffer in the virtual address space. My goal was to find parameters to send to the server so that my client would receive the content of the target file (the string “This is a secret message.”). This succeeded, given a big enough size parameter the server would read past the end of its receive buffer, sending along the secret message after what it had actually received from the client.

How *Noodles* foils the attack To prove that *Noodles* can prevent such an attack from succeeding, I modified the target application again: This time I called `enter_noodle` in its `per_connection` function, turning the thread handling the connection to my malicious client into a *noodle*. The flags passed were `VAS_SWITCH_ALL` and `VAS_SWITCH_LEX`. This allowed me to pass the start address of the mapping of the secret file as part of the exclusion list.

I repeated the experiment from above, the client sending the server the same input that had before lead it to disclose sensitive data. This time the server still tried to read past the end of the receive buffer as its code still contained the bug but suffered a segmentation fault on the attempt. The attack was thus foiled, the client received no sensitive data and the server crashed rather than silently leaking internal data. (Of course it would be possible to handle this case more gracefully should one wish to.)

The code See figure 7.35 for an excerpt of the server's `per_connection` function. Note that the code has been condensed for clarity, e.g. most

error handling has been omitted.

On line 4 `enter_noodle` is called to turn the thread handling the connection into a *noodle*. Note that the `per_connection` function gets called for each client connection the server accepts. Also, I modified the demo application so that there will be one `pthread` created for each new client connection. That thread then executes the `per_connection` function.

On line 5 the exclusion list (containing just a single element here) is passed to `enter_noodle` plus the size of the list, 1. The exclusion list is here just a pointer to a void pointer (`void **`) (an array of void pointers), so we can just take the address of `secret`, a variable containing the void pointer pointing to the start of the mapping of our secret file.

On line 14 the server accepts the SSL connection from the client. Note that some of the setup before this can happen is omitted as it is not relevant for us here. Also the error handling on line 15 is omitted.

On line 19 the server goes into a `while` loop to continually echo back messages to the client (it will only break out of this loop once the client sends “kill” which again is omitted here).

On line 21 the server receives the client’s message, say “BadMessage 8192”. Obviously the number the client sends in this example is much bigger than the number of bytes the server has actually received, which it stores in the variable called `rxlen`. The server then parses the number sent by the client (supposedly a size parameter) and stores it in the variable `num_chars_req`.

On line 34 we have our artificial bug: The server tries to send the client `num_chars_req` bytes back instead of the `rxlen` it actually received. Remember, as the secret file is mapped immediately after the receive buffer in the server’s original virtual address space and both these mappings are 1 page (4KB) each, this would cause the server to send the client the contents of its entire secret buffer. However, since we’re not execution in a normal thread but in a *noodle* with a reduced virtual address space that *doesn’t contain the secret mapping*, an access attempt to the secret buffer will cause a segmentation fault – thus preventing the disclosure of sensitive data.

```

1 void *per_connection(void * arg) {
2
3 #ifdef PROTECTED
4     int res = enter_noodle(VAS_SWITCH_ALL | VAS_SWITCH_LEX,
5         &secret, 1);
6     if (res != 0) {
7         [...]
8     }
9     [...]
10 #endif
11     [...]
12
13     /* Wait for SSL connection from the client */
14     if (SSL_accept(ssl) <= 0) {
15         [...]
16     } else {
17         printf("Client SSL connection accepted\n\n");
18         /* Echo loop */
19         while (true) {
20             /* Get message from client; ... */
21             if ((rxlen = SSL_read(ssl, rxbuf, MAPPING_SIZE)) <= 0) {
22                 if (rxlen == 0) {
23                     printf("Client closed connection\n");
24                 } else {
25                     printf("SSL_read returned %d\n", rxlen);
26                 }
27                 [...]
28             }
29             [...]
30             /* Echo it back but make the "mistake" of allowing
31                both the rxbuf and the mapping following it
32                immediately in the virtual address space to be read
33                by reading num_chars_req instead of rxlen. */
34             if (SSL_write(ssl, rxbuf, num_chars_req) <= 0) {
35                 [...]
36             }
37         }
38     }
39     [...]
40 }

```

Figure 7.35: The modified `per_connection` function of OpenSSL’s echo server demo, edited for clarity and brevity [50], [51]

Apache benchmark [2] results After my experiment to prove that `enter_noodle` can prevent the exploitation of a vulnerability, I then also evaluated a simple HTTPS server with Apache benchmark [2]. The server application is again based on the echo server demo of OpenSSL [50] and sequentially serves GET requests. It either serves them in a newly forked process, a *pthread* (for the baseline) or a *pthread* turned *noodle*. The response times recorded by Apache benchmark are plotted in figure 7.36 as a CDF graph.

Note that the machines (an AArch64 machine and a x86-64 machine) used for benchmarking the simple HTTPS server are the same which were used above, their description can be found in subsection 7.5.2 above.

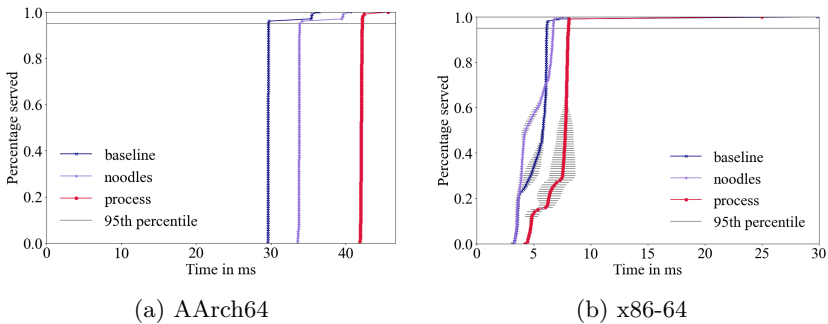


Figure 7.36: **HTTPS server application benchmarks:** Results from benchmarking a *Noodles* example application (HTTPS server) based on an OpenSSL demo [50] with Apache benchmark [2], plotted as a CDF.

The three different scenarios compared are the following:

- A new *pthread* per request: For this set-up each new request is handled in a freshly created *pthread*. This set-up serves as the baseline as it provides no isolation.
- Forking a new process per request: For this set-up creating the *pthread* is replaced with forking and handling each new request in a new process.
- Creating a *pthread* followed by calling `enter_noodle`: For this set-up, after creating a new *pthread* `enter_noodle` is called with the

flag `VAS_SWITCH_ALL` to cause it to predefault all *private writable* mappings. This is done to establish an upper bound in terms of cost, by excluding some of the mappings cost could be further reduced.

Figure 7.36a shows the results for the AArch64 machine while figure 7.36b shows the results for the x86-64 machine. For x86-64 performance is similar for the baseline and the set-up where we also transform the *pthread* into a *noodle*. Forking a new child process each time is more expensive by comparison.

On AArch64 creating a *noodle* adds a bit of overhead compared to the baseline but it is significantly less expensive the forking a new process each time. (Note that due to our AArch64 machine being designed specifically for stable networking performance and its cores almost never reordering instructions, the measurements for this particular machine are extremely stable, showing almost no variance.)

Conclusion The example attack demonstrated above might seem contrived at first glance but buffer overruns fall into one of the most common categories of security sensitive bugs commonly found in C/C++ code: Memory safety bugs. For example, the Chromium Projects reports that roughly 70% of all serious security bugs in the Chrome codebase fell into this exact category [77].

The example application shows that intra-process-isolation in general and *Noodles* in particular can prevent such bugs from leaking sensitive data and that it is thus worthwhile to partition a virtual address space into different domains of trust.

Finally, the evaluation of the simple HTTPS server with Apache benchmark [2] shows that it is indeed feasible to use *Noodles* in an application without incurring too much overhead.

7.6 Conclusion and future work

Motivated by a lack of intra-process-isolation mechanisms available, I presented the design of *Noodles* and a possible programming model for it in this chapter. Notably *Noodles* does only depend on commonly available hardware support and isn't limited to a fixed number of protection domains. The design and intended semantics I described in detail in section 7.2, after providing the motivation for *Noodles* in section 7.1.

I've successfully implemented the idea of having threads with their own virtual address spaces, so called *noodles*, in a patch of my own that I've used to modify the Linux kernel version 5.15.116. This work was presented in section 7.4. I then evaluated the resulting kernel in section 7.5 both empirically with micro benchmarks (showing that *Noodles* incurs only 10%-11% of `fork(2)`'s overhead in specific scenarios) as well as by using *Noodles* to protect a server demo application from inadvertent data disclosure.

In addition to this, I shed light on how exactly the Linux kernel manages the virtual address space in section 7.3, showing that the Linux kernel has no real abstraction for a truly shared mapping. I also discussed relevant prior work, a previous patch series submitted to the Linux kernel, in detail, pointing out why it doesn't work as intended.

Though *Noodles* is still at the prototyping stage, I believe it demonstrates that even a code base as complex as the Linux kernel can be retrofitted with the idea to decouple virtual address spaces and processes.

7.6.1 What's next

To develop *Noodles* into a production grade patch series that can be submitted to the Linux kernel, there are several avenues worth exploring:

- **Mirrored VMAs:**

Currently the Linux kernel doesn't have a native abstraction to represent a truly shared mapping which is part of multiple virtual address spaces. I therefore propose the introduction of a new type of mapping, the *mirrored* type. This would be a special type that *private* mappings (both *anonymous* and file backed ones) are transformed into once they have been copied into the *noodle*'s

new virtual address space or vice versa a copy has been made of them. *Mirrored* mappings (respectively `vm_area_structs`) would keep track of either the mapping they had originally been copied from, their *original* so to speak, or their copies.

The semantics of such *mirrored* mappings would be as follows: Whenever a page fault is suffered and the faulting address falls within the virtual address region covered by a *mirrored* mapping, the page fault handler itself would take care to modify either the original or the copies in the same way (depending on if the page fault was suffered by the copies or the original). That would mean also updating the page table of the original or the copies.

If a *mirrored* mapping is deleted, its twin's type would change (back) to being a *private* mapping. Permission changes would also not be reflected to the twin.

The great benefit of having *mirrored* mappings is that it makes any costly prefaulting unnecessary, thus greatly improving performance of *Noodles*. However, the main challenge (and the reason why I have not yet implemented this) is that a locking scheme will have to be found that cannot result in a deadlock: When a page fault is handled, the lock located on the active `mm_struct`, `mm_lock`, will eventually have to be taken (first in read, then in write mode). Since *mirrored* mappings are not covered by the same `mm_struct`, this means that the lock of the other `mm_struct` (the one either the original or copy belongs to) would also have to be taken in the page fault handler. If no care is taken when implementing this and for instance page faults are suffered simultaneously by the original and the copy, the attempt to also take the other lock could easily result in a deadlock.

Normally such situations are resolved by establishing a global order of all locks and then making sure that any locks are always taken in the exact same order. Unfortunately, with the `mm_lock` being already held when control is passed to the main part of the page fault handler this might involve first dropping the lock again before attempting to take both `mm_locks` in the correct order.

There are also additional locks protecting the upper level of the page table trees that have to be taken care of.

In conclusion, while it seems doable to implement *mirrored* mappings, extreme care needs to be taken to correctly manage all concerned locks and handle concurrency. This is especially true because parts of the page fault handling code path are quite *brittle*, differentiating between different cases often implicitly by merely checking if certain fields of certain structs are set (or are *null*).

- **Switching back:**

Another worthwhile goal is to make it possible, after all, for a thread to undo its transformation into a *noodle* by switching back to its original virtual address space. This would even allow for an alternative programming model, as described in subsection 7.2.4, and in general give the application programmer more flexibility.

Having the additional type of the *mirrored* mapping would simplify the implementation of switching back (see the description of *mirrored* mappings and their semantics just above). However, even with *mirrored* mappings it would still be possible for two virtual address spaces, that of the *noodle* and the original one, to diverge to the point of being irreconcilable: If, for instance, in both virtual address spaces a new mapping is installed independently of each other and those two's virtual address ranges happen to intersect, switching back would either be impossible or the semantics of a switch back would have to be changed to allow for the possibility of certain mappings to disappear upon transforming back into a thread.

- **Integration into `clone(2)` itself**

Also a possible future work direction to explore is to attempt to merge the new `enter_noodle` system call directly into `clone(2)` itself: Instead of having to first create a thread to avoid losing access to the original virtual address space by transforming the main and only thread into a *noodle*, `clone(2)` would then automatically create a thread for us. After thread creation, it would then switch it into the new virtual address space without ever returning control to user space in between.

On the downside, while it might be more efficient to call only a single system call, it would also take away some of the flexibility of `enter_noodle`. Also, `clone(2)`, especially its most recent version

`clone3(2)`, is already a bit overloaded with options and in terms of the functionality it covers. (It has developed into a veritable Swiss army knife of a system call.)

- **Can we do it in a module after all?**

An alternative to developing a patch (series) to change the Linux kernel itself is to implement functionality that needs to be added to the kernel in a module instead. This has a number of advantages: First, it greatly simplifies development because it does away with the necessity of having to re-compile the kernel every time a change to the code is made. Also, a module can be inserted and removed without having to reboot (there are ways to speed up re-booting, developing in a virtual machine can be helpful, also there is `kexec(8)` [41] which allows to potentially cut down on reboot time).

Second, implementing kernel functionality that cannot be upstreamed in a module simplifies the necessary self-maintenance.

However, as attractive as modules are, they have a number of limitations: For example, a module cannot define a new system call since adding a new system call requires editing a number of (architecture dependent) files and adding the new system call to the lists kept there.

In fact, this problem isn't just limited to system calls: Whenever already existing kernel functionality needs to be *changed* in any way and that functionality is implemented in the core kernel source itself (not in a module be it an in-tree module or one maintained by a third-party) a module isn't a good option. While it might be sometimes a possibility to work around this by *duplicating* code instead of *modifying* it that is obviously not very elegant, generally undesirable and also won't work in all circumstances (for instance when trying to add a field to an already existing data structure).

Noodles adds a new system call. To implement it in a module instead, that functionality could in theory be implemented by defining custom behavior for `sysioctl` (in fact there are a number of ways to call custom code that needs to execute in kernel space from user space). Much more of a problem would be that *Noodles* also modifies several preexisting kernel functions (mostly by adding addi-

tional parameters but not always) and that it needs these modified versions to be called by other kernel code, eliminating the option of just creating a modified duplicate: An example for this is resource clean-up on thread termination. To handle this, *Noodles* adds its own clean-up work to the Linux kernel's functions that take care of freeing resources after a task terminates. This cannot be done from a module short of dynamically rewriting the kernel's code, e.g. by replacing the functions in question with modified ones.

To conclude this short discussion, trying to press *Noodles* into a module, while not completely out of the question would likely severely complicate its implementation.

This exploration of possible future work concludes our discussion of *Noodles* and this chapter.

Chapter 8

Related Work

In this chapter I discuss prior work that is either related to the proposed novel intra-process-isolation mechanism *Noodles* (discussed in detail in chapter 7) or to the lightweight C-runtime *Mistletoe* (discussed in chapter 6) or to intra-process-isolation in general.

8.1 General intra-process-isolation

This section surveys several related papers that propose some kind of general intra-process-isolation mechanism. Where appropriate it compares the proposed mechanisms to *Noodles*. (Related work that focuses on intra-process-isolation with hardware support in the form of Intel’s MPK [27] can be found in the next section, section 8.2.)

8.1.1 SpaceJMP [19]

The SpaceJMP paper presented at ASPLOS 2016 is one of the main inspirations for the *Noodles* project. Furthermore, as already mentioned in chapter 7, a patch [92] developed for the Linux kernel several years back, aiming to implement SpaceJMP for Linux, served as a basic starting point for my own prototype.

- **Motivation:** The SpaceJMP paper argues that the virtual address space as generally provided on 64 bits machines isn’t big enough for

certain extremely data intensive applications. The paper correctly points out that not all of the 64 bits a virtual address consists of on a 64 bits machine are actually in use on today's machines during virtual to physical address translation. Instead, the unused higher bits of the address are simply set to 0 for user space addresses and to 1 for kernel space addresses. This results in a unusable hole in the virtual address space located between the user's part of it and the part belonging to the kernel [33].

The paper further argues that it is expensive for process manufacturers to use more bits in the virtual to physical address translation process and thus not feasible (at least in the short term) to increase the actually usable part of the virtual address space. Instead, the paper suggests further that this can be mitigated by decoupling processes and virtual address spaces, making it possible for a process to attach to multiple virtual address spaces and then switch between them at will. This could also allow to efficiently save the content of a virtual address space which is indeed a good argument (in fact, this could be useful to mitigate cold starts on serverless platforms).

The SpaceJMP paper also criticizes the interface offered by `mmap(2)`, pointing out that it is entirely possible to just overwrite an address mapping if the caller of `mmap(2)` isn't careful. (Though this is indeed true, this can be mitigated by passing the right flags to `mmap(2)`.)

- **Key idea:** The idea at the center of the SpaceJMP paper is to decouple virtual address spaces and processes. Instead of just viewing virtual address spaces as a side product of process creation, the paper proposes to promote virtual address spaces to so called "first class citizens": They could be created separately from processes, could survive a process lifetime to be reused at a later point by another process (for easy sharing of the virtual address space content) and they could be destroyed at any point if they are no longer needed (presuming all previously attached processes have switched away and detached).

Though *Noodles* is as already stated inspired by this paper, there are nevertheless a number of key differences: First, while the SpaceJMP

paper only mentions processes and doesn't extend its idea to threads, *Noodles* explicitly targets threads (as viewed from user space) or tasks (as viewed from kernel space). Another key difference is that while SpaceJMP proposes multiple virtual address spaces as a means to increase the storage space available to an application, *Noodles* by contrast aims to isolate a thread from other threads running in the same virtual address space. *Noodles* also doesn't aim to share pointer graphs between different processes or potentially persist them beyond a process lifetime – on the contrary, *Noodles*'s goal is to limit the original virtual address space to a user-defined subset to sandbox a thread. Finally, SpaceJMP only presents prototypes for DragonFly BSD [17] and Barrelfish [21], it doesn't have a solution for the Linux kernel which the *Noodles* prototype targets.

8.1.2 Light-Weight contexts: An OS abstraction for safety and performance [45]

The lwC paper was published at OSDI 2016 and directly cites the SpaceJMP paper. Furthermore, it also attempts to deconstruct the process abstraction by decoupling certain elements – execution state and the execution agent (e.g. a thread) in this case.

- **Key idea:** lwC stands for Light-Weight Context and encapsulates the execution state of a process (including its file descriptors and credentials) at a certain point in time. A lwC can be thought of (and used as) a snapshot. Note that a snapshot is effectively taken by fork when creating a child process before making the new process runnable. The key difference here is that this “child” – this snapshot – is never actually started but essentially remains paused indefinitely until a thread switches to it with a special, newly created system call (`lwSwitch()`). The calling thread will then start executing at the exact code line (and in the state) the snapshot was taken (a snapshot can be taken by another new system call that creates lwC and then returns a file descriptor to it, called `lwCreate()`). This essentially amounts to a `goto` statement that allows you to go back in time – even undoing earlier privilege drops.

Another way to think about lwCs is to compare them to coroutines:

The paper states that their new abstraction can be viewed as a coroutine that is also isolated and privilege separated.

- **Use cases:** As described in the lwC design section and demonstrated in the evaluation section of the lwC paper, lwCs can be used for different types of use cases: Algorithms are presented for creating a snapshot and then rolling the execution state back to it, an event-driven server with session isolation, sensitive data isolation and the implementation of a reference monitor.

For intra-process-isolation, the paper describes the necessary steps as follows: First we need to create a new lwC. In a second step the current lwC then drops some of its privileges (making e.g. the virtual memory section where cryptographic keys are stored inaccessible). After that workloads that shouldn't have access to the isolated virtual memory section are executed. Whenever we do need to access it we switch into the lwC created before dropping our privileges before returning to the default lwC. (See the paper [45] for more details.)

The described different usage patterns which are given as algorithms by the paper are used with a whole range of applications: Especially interesting for us is that they use the isolation (and monitor) use case with Apache [3] and nginx [47].

Note that their implementation targets FreeBSD [76].

It's interesting to compare the *light-weight contexts* paper to *Noodles* because they seem both inspired by the SpaceJMP paper [19] – except in different ways: The *light-weight contexts* paper separates the state of an execution context from the unit or agent actually performing said execution, the thread, while *Noodles* goes from a virtual address space shared by all threads to one per thread (with the rest of the state of the process remaining shared).

To summarize, both approaches can be said to destruct the process abstraction but in different ways, making them mostly orthogonal to each other. In addition, because they target different operating systems and FreeBSD and Linux have differently structured memory management systems – strongly influencing how expensive it is to snapshot a process's state – it's hard to compare their performance directly. Though

it can be said *Noodles* that if a similar procedure to forking is used to create the snapshot then *Noodles* has an advantage since it performs less work (only duplicating chosen parts of the mappings forming the virtual address space).

8.1.3 Enforcing Least Privilege Memory Views for Multithreaded Applications [12]

This paper was published at CCS 2016 and presents Secure Memory View (SMV)s.

- **Motivation:** Similar to other papers discussed in this subsection, the SMV paper makes the case that selective memory isolation is needed in big, monolithic applications to prevent one component from corrupting another. Specifically, the authors argue that what they call a “third generation” privilege separation technique is needed since earlier generations are either unable to support multithreaded applications or only by drastically reduction performance.
- **Key idea:** The SMV model introduces three new abstractions: The *memory protection domain*, the *secure memory view* itself and the SMV thread. The *memory protection domain* refers to a contiguous range of virtual addresses (a virtual address can only be part of a single *memory protection domain*) and seems roughly analogous to the familiar `vm_area_struct` (see section 7.3 for details). A *memory protection domain* can be associated with a *secure memory view* which is a thread container and will define how (read-only, read-write etc.) the *memory protection domain* can be accessed by its threads.

Note that a SMV thread is a thread associated with a *secure memory view* and that it can only be associated with a single one. A *memory protection domain* on the other hand can be associated with more than one *secure memory view* at a time. This allows for different components – thread groups – to have potentially different permissions (access policies) for a set of virtual address regions.

While *Noodles* programming model is markedly different from that presented in the SMV paper, there are commonalities when it comes to

the implementations of the respective prototypes. The SMV paper also targets the Linux kernel (version 4.4.5) and by virtue of having to work with the same memory management system design, some challenges were solved in a similar way: For instance, the SMV prototype also duplicates page tables to give a group of threads a unique view of the virtual address space (and have the MMU enforce different access permissions for the same virtual memory regions for different thread groups). However, while *Noodles* chooses to give a *noodle* turned thread a fully featured virtual address space, the SMV paper instead associates multiple sets of page tables with the same `mm_struct`. Moreover, SMV's prototype only supports differing permissions for a specific virtual address region and doesn't allow to etc. unmap the virtual address altogether.

8.1.4 Endoprocess: Programmable and Extensible Subprocess Isolation [105]

The *endoprocess* paper published in 2023 at NSPW has goals related to those of *Noodles* – namely providing a novel abstraction for intra-process-isolation – but proposes a completely different solution:

- **Motivation:** In a vein similar to this thesis, the *endoprocess* paper argues that intra-process-isolation is a requirement because mutually distrusting components become part of the same process in almost every system of non-trivial complexity (nginx [47] is given as an example). The paper claims further that a new abstraction is needed, called the *subprocess*. The reason given for this is that previous intra-process-isolation solutions are allegedly too application specific, change the system interface too much or tightly couple applications in need of intra-process-isolation to the operating system.
- **Key idea:** The paper proposes to install a monitor inside each process, a so called *endokernel*, turning the process into an *endoprocess*. This monitor is to be isolated from the rest of the application which is in turn split into subprocess which are both isolated from each other and the *endokernel*. The *endokernel* then manages these subprocesses, essentially virtualizing the operating system's interface for their benefit to minimize porting overhead.

The paper does not specify a specific intra-process-isolation mechanism to separate the monitor and the different subprocesses from each other for the general design of the *endoprocess*. It just mentions that different mechanisms exist to implement this. (Though their prototype *Little MAC* apparently uses Intel’s MPK [27].)

It is suggested that the separation of an application can be done with the help of special source code annotations though no specific example is provided. It seems that the idea is to automatically separate an application into different subprocesses depending on what and how data is used. The idea is to provide a framework to let a developer implement different data sharing policies while also providing some default implementations of these policies for convenience.

All in all it seems that their proposal is to achieve intra-process-isolation by effectively wrapping a process into something akin to microkernel VM so that the different protection domains can then be split up between different subprocess. Notable is here that the kernel of this “VM” is actually isolated from the rest of the *endoprocess* with an intra-process-isolation mechanism.

When comparing to *Noodles* it is apparent that though *Noodles* is less general because it specifically targets the Linux kernel, *Noodles* is also much more flexible since it only proposes a new primitive that a developer can use in any way as they like. The *endoprocess* paper proposes a whole framework by contrast, a framework that might in fact be able to make use of *Noodles* as its intra-process-isolation mechanism.

8.2 Isolation using memory protection keys

This section focuses on prior work proposing some form of intra-process-isolation scheme employing memory protection keys (primarily Intel’s MPK [27] though much is also applicable to the Arm equivalent PIE/POE [6]).

8.2.1 *libmpk*: Software Abstraction for Intel Memory Protection Keys (Intel MPK) [72]

This paper was published as part of the proceedings of Usenix ATC 2019.

- **Motivation:** As already touched upon in the section 5.2, the new (at the time) hardware protection keys, MPK, available on recent Intel processors, suffer from three problems: For one, there are only 16 different keys available (including one corresponding to the default domain, key 0). Second, they suffer from the protection-key-use-after-free problem: Calling the system call `pkey_free(2)` offered by the Linux kernel does not in fact invalidate any ptes already associated with the key being freed or remove the key from them. Finally, they can be seen as being orthogonal to `mprotect(2)`’s process granularity protection model.
- **Key idea:** The key idea of the *libmpk* paper is to implement a library called *libmpk* that virtualizes the 16 protection keys. The paper proposes to use `mprotect(2)` to mark pages belonging to a virtual key as inaccessible when the virtual key in question needs to be spilled (in the case of not having enough physical keys to map the virtual ones to). For spilling not recently used virtual keys are selected.

The paper also presents real-world application evaluations concerning OpenSSL [50] and a Javascript JIT compiler (among others).

The paper’s prototype consists of a user space library and Linux kernel source code the authors released on Github [71].

8.2.2 Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries [24]

This paper, to which I am referring to as the Hodor paper in the following description, was published at ATC 2019.

- **Motivation:** In contrast to the other papers discussed in this chapter, Hodor is concerned with isolating specially privileged libraries (e.g. libraries implementing a kernel-bypass by acting as a user space driver for a specific device – usually for reasons of performance). These specially privileged libraries are included by applications wishing e.g. to interact with a specific device.

Unfortunately, while bypassing the kernel can come with impressive performance benefits, it also comes with serious security concerns as a library normally runs in the same virtual address space as the rest of the application including it. Thus the library has no straightforward way to prevent the application from e.g. corrupting its data structures. This concern implies this is another use case for intra-process-isolation.

- **Key idea:** Hodor proposes and evaluates 3 different ways to isolate such a security critical library (that should really be considered to be part of the operating system as well) from the rest of an application: The first idea is to have a separate set of page tables for the library context, similar to what the SMV paper [12] proposes. When now a library function is called, the execution control flow is directed to go over a so called trampoline where the pointer stored in CR3 (pointing to the page global directory) is swapped out (the lower levels of the page tables are shared).

The second idea works similarly but uses Intel’s EPT mechanism. Finally, the third idea uses Intel’s MPK [27] to isolate the library instead.

The paper evaluates the three presented options with a range of different benchmarks. The one using MPK performs best.

By contrast, *Noodles*’s key idea is a more general and can be applied in more than one use cases while Hodor is solely concerned with isolating data-plane libraries.

8.2.3 ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK) [94]

This paper to which I'm referring as the ERIM paper in the following, was published at Usenix Security 2019.

- **Motivation:** The authors of the ERIM paper argue that intra-process-isolation is needed to protect cryptographic keys from bugs such as Heartbleed [23] or language runtimes of memory safe languages from potentially unsafe, native libraries. The paper argues further that while it is possible to use conventional page-table based isolation (enforced by the MMU) by simply updating the permissions in the ptes, this isn't always an option: If the protection domain switches occur frequently, this incurs too much overhead.
- **Key idea:** The paper presents ERIM which uses Intel's MPK [27] to partition memory into two different protection domains, a trusted and an untrusted domain. This is then paired with binary inspection (potentially even rewriting the binary) and call gates to prevent rogue updates to the PKRU register storing the permissions for the 16 keys (the register being writable from user space).

An example use case for ERIM would be a program with a runtime (running in the trusted protection domain) that loads native libraries (running in the untrusted protection domain). In general ERIM targets application with only two protection domains that switch between them frequently.

Again, when comparing to *Noodles* it is evident that *Noodles* proposes something much more general, a novel operating system level intra-process-isolation primitive, while ERIM proposes a scheme specific to certain use cases. Because of this, they can be said to be orthogonal to each other.

8.2.4 VDom: Fast and Unlimited Virtual Domains on Multiple Architecture [106]

The VDom paper was published at ASPLOS 2023.

- **Motivation:** The VDom paper also points out that many applications have different components running inside the same virtual address space and that it's desirable to isolate those in different protection domains. It further states that while hardware primitives such as Intel's MPK [27] perform very well when using them for intra-process-isolation since this makes switching the protection domain very cheap, there are unfortunately only 16 different domains available (both on Arm and Intel hardware). As a result of this, according to the paper, existing solutions, e.g. libMPK [72] would incur too much overhead: When an application needs more than 16 different protection domains, *libmpk* needs to spill a protection domain and falls back on `mprotect(2)` for this. This is problematic when an application uses more than 16 different protection domains and those also happen to be active at the very same time (e.g. in a server with more than 16 concurrent worker threads). In this case a thread has to wait until a key becomes available, causing massive slowdowns.
- **Key idea:** The key idea of the VDom paper is to group threads into different virtual address spaces depending on how many and which protection domains they need to be able to access and thus be able to efficiently switch between. This is reminiscent of what was already discussed in the context of *Noodles*: if a thread is associated with a new virtual address space it gets its own contingent of memory protection keys.

The VDom paper virtualizes memory protection keys (which it calls *vdoms*) by mapping them to the hardware provided memory protection keys (*pdoms*) available in different virtual address spaces. To implement the different virtual address spaces, the VDom paper associates more than one set of page tables with a single `mm_struct`. To prevent the TLB from thrashing because of the potentially many updates to register storing the `pgd`, the paper proposes to use ASIDs.

To implement all of this, the authors of the paper modified a Linux kernel 15.17 (the authors report changing 4290 lines of code across 58 source files which is a much bigger change than needed for the *Noodles* prototype). Both Intel x86-64 and Arm platforms are

supported.

The prototype described by the VDom paper targets the Linux kernel and has commonalities with both those of the SMV paper and that of *Noodles* though more so with the former: As chosen by the SMV paper for its implementation the VDom prototype also uses only a single `mm_struct`. This it then associates as needed with multiple sets of page tables (tagged with an ASID to avoid unnecessary TLB flushing) to support potentially unlimited memory protection keys.

8.2.5 Comparison to *Noodles*

To conclude this section, let's compare *Noodles* to Intel's MPK [27] and Arm's PIE/POE [6]: First, in terms of the number of protection domains supported, the new architectural extensions only support up to 16 different domains, while *Noodles* supports as many different protection domains as the system has the memory available to store the necessary data structures (their size is largely determined by how big the subset of page tables copied is).

On the flip side, *Noodles* does not allow a thread to switch protection domain after it has already been turned into a *noodle* but instead requires a rendezvous-style programming model.

Permission changes and switching protection domain is equivalent when working with memory protection keys. For *Noodles*, it requires updating the permission bits in the ptes.

Another difference is that while turning a thread into a *noodle* is a privileged operation, switching protection domains when using memory protection keys is largely unprivileged (an exception to this is Arm's PIE). As a consequence of this, memory protection keys normally have to be paired with another technique to achieve intra-process-isolation, as can e.g. be seen with ERIM [94] (see subsection 8.2.3) and Hodor [24] (see subsection 8.2.2).

I propose that instead of viewing *Noodles* as standing in competition to memory protection keys (and the systems built on top of them), it should be viewed as another tool in the arsenal of the programmer as memory protection keys and *Noodles* can potentially be used in conjunction. The principle of defense-in-depth argues for having more than one layer of defense, the necessity of which has become more accepted

recently, in part due to the discovery of vulnerabilities such as Heartbleed [23]. One example of the growing acceptance that having only one mechanism (even if sufficient in the absence of bugs) isn't good enough in practice is the recent inclusion of the new system call `mseal(2)` in the Linux kernel [80] (`mseal(2)` allows to disallow certain changes to memory mappings, one possible use case of this being to prevent the mapping in question from becoming executable).

8.3 Work related to *Mistletoe*

While *Mistletoe*'s approach seems to be unusual in that it combines system programming (or at least low level access) with GraalVM's Native Image [66], there is prior work modifying Java code and modifying Java application behavior.

8.3.1 Controlled, systematic, and efficient code replacement for running java programs [49]

The PROSE paper was published in 2008 at Eurosys.

- Motivation: The paper points out that when an application is developed it is often not known which parts will need to be instrumented for performance analysis and bottleneck determination later on, leading to extensive changes often distributed all over the application code at a later point in time. Debugging and logging code might also end up sprinkled all over an application when hunting down an elusive bug, making it hard to cleanly reverse all these changes later on (especially if unrelated changes were made at the same time). Another usage case is that of applications needing to be patched that cannot be restarted without causing some kind of service disruption.

All these challenges, plus being able to “hot swap” a method implementation for A/B testing, the paper proposes to address by dynamically replacing code of the running Java application.

- Idea: The key idea of this paper is to replace method bodies at runtime (though no interface changes of any kind are supported), without ever recompiling the running Java application. Change sets are called *aspects* that in turn consist of potentially multiple *crosscuttings*. A *crosscutting* contains both an *advice* – the actual code change itself – and an optional *cutpoint*. The latter is a pattern that selects specific methods from a more general list of potentially targeted methods.

When comparing *Mistletoe* to PROSE it needs to be pointed out that they target different use cases: While *Mistletoe* attempts to make changes (or additions) to the behavior of the VM itself, PROSE targets

the Java application. Then, there is a difference in when the changes are made: PROSE does its work at runtime while *Mistletoe* performs changes before an application is even started (this can be after an application has already been compiled due to loading *libmistletoe* dynamically). PROSE also doesn't target native images but Java application running on top of a conventional JVM.

8.3.2 Debugging and tracing tools

Aside from systems like presented in the PROSE paper [49], there also exists an abundance of tools for debugging and tracing Java applications. I name two examples below but those are only tangentially related to *Mistletoe* as the targeted use case is different (being primarily to debug and trace the Java application itself while *Mistletoe* targets the VM of GraalVM's Native Image.)

The JVM TI The Java Virtual Machine Tool Interface (JVM TI) is a low level debugging interface that allows debugging and profiling tools to attach to the JVM. It is event based, so that tools (which are running in the same process as the JVM) can be notified when e.g. a breakpoint is hit or an exception is raised (among many other possibilities). Note that JVM TI is currently not supported by GraalVM Native Image [67].

The main difference to *Mistletoe* is that JVM TI is event based and geared toward allowing the implementation of debugging tools. It's purpose is not to change how the JVM behaves.

JDK Flight Recorder JDK Flight Recorder is aimed at profiling Java applications with as little overhead as possible. It generates trace files in a binary format and can be understood as the Java equivalent of Arm's CoreSight tracing framework [35].

Chapter 9

Future Work

In this chapter I discuss possible future work that combines both the first part of this thesis, see chapter 6, and the second part, see chapter 7.

It seems straightforward to attempt to combine GraalVM [55]’s *isolates* [102] with the *Noodles* prototype. There is however a limitation currently presented in the *Noodles* prototype that would ideally be resolved first before attempting to do this: To execute each *isolate* in a separate *noodle*, the ability to switch back to the original virtual address space seems essential. As already outlined in subsection 7.6.1, implementing a custom page fault handler and thus creating a new type of mapping, termed *mirrored mappings*, should make this feasible.

Alternatively, only using *isolate* once could also be an option as *Noodles* is currently optimized for the use case where a *noodle*’s state is destroyed after it has run to completion.

Having a separate heap for each *isolate* is an ideal fit for *Noodles* but another obstacle that would have to be resolved first is how stacks are handled: In the version of GraalVM I use in this thesis (commit: `de8da6cee7c0d6efe1f44970518cf02ad7f933ad`), threads and also stacks are orthogonal to *isolates*. When switching to another *isolate* this is accomplished with regard to the stack by adding a new stack frame thus making it not only possible but likely that a stack will contain stack frames from different *isolates*. While it would theoretically be possible to just share the mapping containing the stack between the original virtual address space and that of a newly created *noodle*, it makes more

sense from a security perspective to establish a clear separation between the stack frames belonging to different *isolates*.

With these obstacles addressed, it would become possible to build a lightweight serverless platform running user-provided functions in separate *isolates*. In fact, the *isolates* demo [63] provided by Oracle on Github even suggests something along the lines: It consists of a server (built on top of the Netty application framework [90]) which accepts mathematical functions as user-provided input. Those it then plots in a separate *isolate*.

Chapter 10

Conclusion

In this thesis I have made the case that a lightweight intra-process-isolation mechanism is needed, though traditionally hardware and operating systems support existed only for inter-process-isolation. Even though this is now starting to change, the emerging hardware support for intra-process-isolation comes with severe limitations, as we have seen in section 5.2, and is not yet readily available.

Of the many scenarios that demand intra-process-isolation, this thesis focuses primarily on the requirements of lightweight multi-tenant serverless platforms, similar to e.g. GraalOS [53] and Cloudflare Workers [9]. As discussed in section 5.3, for serverless platforms to be economically feasible, the overhead incurred to isolate each user-provided workload should be as small as possible. If each function – even though running only for a very short amount of time – has to be isolated in its own container or even virtual machine, a lot of resources are essentially wasted to fulfil the promise of automatic scaling and pay-as-you-go. This in turn results in high prices as especially memory usage is directly correlated to infrastructure cost for the platform operator.

Instead, a more lightweight intra-process-isolation is needed – not just for lightweight serverless platforms but also for web servers, to isolate the core part of applications from libraries and reduce the blast radius of memory disclosure vulnerabilities such as Heartbleed [23]. For many of these use cases processes are either a too coarse grained abstraction that comes with a high performance penalty or it is simply not

possible to retroactively partition an application into different processes within a reasonable amount of time.

My approach to the non-trivial difficulty of adapting a real-life system to make use of a new intra-process-isolation primitive (be it a hardware supported one such as MPK or an operating system based one such as *Noodles*) is *Mistletoe*: *Mistletoe* is a lean C-runtime imposing negligible overhead in most scenarios. It enables a systems programmer to place hooks in GraalVM [55]’s execution flow and execute custom code. I developed *Mistletoe* while exploring GraalVM’s *isolates* [102] and their applicability to intra-process-isolation. *Mistletoe* consists of both custom hooks as well as a dynamically loaded library making it possible to adapt behavior without any need to recompile either GraalVM itself or an application built with it’s Native Image builder.

In chapter 6, after I presented the design of *Mistletoe* and discussed the requirements and goals that inspired it, I discussed several different possible use cases for *Mistletoe*. Furthermore, I presented and evaluated a prototype application using it for per-*isolate* cycle accurate billing. I ran both micro benchmarks and benchmarks part of the standard Java benchmarking suite DaCapo [8], proving that it is possible to leverage *Mistletoe* with reasonable overhead in a real-world. It represents an easy and straightforward way for system programmers to implement new functionality that can then be called directly from within GraalVM itself.

In the second part of this thesis I presented a novel intra-process-isolation mechanism called *Noodles*. The key idea of *Noodles* is to allow a thread to switch to its own virtual address space at any point in time (while still remaining part of the same process context) by calling a newly implemented system call, `enter_noodle`. The new system calls API allows the caller to easily specify which data stored in the original virtual address space should be shared with the newly created fat thread while being able to remove access to all other data. Note that *Noodles* provides stronger security guarantees than most other mechanisms since a mapping of physical pages is actually removed from the page tables and not only marked as inaccessible – providing a stronger defense against attacks such as Rowhammer [34].

I presented a possible programming model to make use of *Noodles* in the context of a lightweight serverless platform, this being the targeted use case for *Noodles*. After discussing *Noodles*’s design and semantics

in section 7.2 I then presented the implementation in section 7.4, taking care to concretize how the design of the Linux kernel’s memory management system impacted my prototype. Among other things, I discussed the Linux kernel’s inability to truly share a mapping (as opposed to the mapped physical data pages) between multiple virtual address spaces.

The resulting Linux kernel patch targets the recent 5.15 Linux kernel with the new system call `enter_noodle` serving as its API. The results obtained by running my custom benchmark suite comparing *Noodles* against other ways to create new execution contexts show that *Noodles* is only 10%-11% as expensive as `fork(2)` in relevant scenarios (I ran benchmarks on both an x86-64 and an AArch64 machine).

Finally, I demonstrated *Noodles* applicability to real-world applications by presenting an experimental HTTPS server application using *Noodles* to protect its secret data. *Noodles*’s fine grained memory isolation capabilities allowed the server to prevent the exploitation of a Heartbleed [23] inspired bug I had deliberately introduced. I also evaluate the example server application with Apache’s server benchmarking tool [2], further demonstrating *Noodles* feasibility.

Glossary

AVR AVR is a RISC architecture used by microcontrollers originally developed by Atmel which has since been acquired by Microchip Technology.. 82

glibc The GNU C Library. 66, 115, 135, 158, 159, 168, 173, 177, 178, 181, 182, 185

inter-process-isolation Isolation between execution contexts running in different process contexts. 89

intra-process-isolation Isolation between different execution contexts, e.g. threads, running in the same process context. 89, 128

pkey *pkey* is what the Linux kernel calls Intel's memory protection keys.. 28

Acronyms

AOT Ahead of Time. 34, 42, 43, 44, 47, 58, 66

ASIC Application Specific Integrated Circuit. 64

ASID Address Space Identifier. 128, 215, 216

bdb Page Directory Base Register. 128

CHERI Capability Hardware Enhanced RISC Instructions. 61, 62

COW Copy-On-Write. 46, 47, 76, 95, 97, 105, 116, 117, 119, 123, 132, 175, 181, 185

ELF Executable and Linkable Format. 40, 66, 129, 159

EPT Extended Page Tables. 213

ETM Embedded Trace Macrocell. 63

FPGA Field Programmable Gate Array. 62, 63, 64

IPC Intra Process Communication. 20, 22

JDBC Java Database Connectivity. 82

JDK Java Development Kit. 18, 23, 31, 34, 41, 219

JIT Just In Time. 34, 35, 36, 38, 40, 42, 44, 46, 212

- JNI** Java Native Interface. 32, 53, 64, 66
- JVM** Java Virtual Machine. 14, 15, 18, 21, 23, 31, 32, 34, 35, 36, 37, 38, 40, 41, 43, 44, 45, 46, 52, 53, 54, 55, 56, 57, 86, 87, 219
- JVM TI** Java Virtual Machine Tool Interface. 219
- JVMCI** JVM Compiler Interface. 37
- LTS** Long Term Support. 14, 16
- lwC** Light-Weight Context. 93, 207, 208
- MMU** Memory Management Unit. 22, 25, 96, 97, 113, 210, 214
- MPK** Memory Protection Keys [27]. 13, 14, 15, 18, 22, 27, 28, 29, 87, 100, 101, 163, 205, 211, 212, 213, 214, 215, 216, 224
- MPM** Multi-Processing Module. 27
- NPU** Neural Processing Unit. 64
- NUMA** Non-Uniform Memory Access. 62
- p4d** p4d. 118, 132, 180
- PCID** Process Context Identifier. 128
- pgd** Page Global Directory. 111, 118, 132, 215
- pid** Process ID. 100, 109
- PIE** Permission Indirection Extension. 13, 15, 18, 22, 28, 100, 101, 163, 212, 216
- pmd** Page Middle Directory. 118, 180
- POE** Permission Overlay Extension. 13, 15, 18, 22, 28, 100, 101, 163, 212, 216
- pt** Page Table. 118, 119

- pte** Page Table Entry. 22, 23, 28, 118, 132, 161, 163, 176, 180, 185, 188, 212, 214, 216
- pud** Page Upper Directory. 118, 180
- QoS** Quality of Service. 97
- SMV** Secure Memory View. 209, 210, 213, 216
- tgid** Thread Group ID. 109
- THP** Transparent Huge Pages. 155
- tid** Thread ID. 109
- TLB** Translation Lookaside Buffer. 130, 138, 215, 216
- TTD** Translation Table Descriptor. 28
- TTE** Translation Table Entry. 28
- UID** User ID. 26
- VAS** (First class) Virtual Address Space. 132, 134, 136, 137, 138, 139, 140, 141, 143, 145, 146, 147, 152, 153, 154, 155, 162
- vDSO** Virtual Dynamic Shared Object. 158
- VMA** Virtual Memory Area. 113
- XSLT** Extensible Stylesheet Language Transformation. 83

Chapter 12

References

- [1] AGACHE, A., BROOKER, M., FLORESCU, A., IORDACHE, A., LIGUORI, A., NEUGEBAUER, R., PIWONKA, P., AND POPA, D.-M. Firecracker: lightweight virtualization for serverless applications. In *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation* (USA, 2020), NSDI'20, USENIX Association, p. 419–434.
- [2] ab - Apache HTTP server benchmarking tool. <https://httpd.apache.org/docs/current/programs/ab.html>. Accessed on 09-04-2024.
- [3] Apache HTTP Server Project. <https://httpd.apache.org/>. Accessed on 05-22-2024.
- [4] Apache Lucene. <https://lucene.apache.org/>. Accessed on 10-07-2024.
- [5] ARM LIMITED. *Address Space Identifiers - Tagging translations with the owning process*. <https://developer.arm.com/documentation/101811/0104/Address-spaces/Address-Space-Identifiers---Tagging-translations-with-the-owning-process>.
- [6] ARM LIMITED. *Learn the architecture - AArch64 memory attributes and properties*, November 2022. Permission Indirection

- and Permission Overlay Extensions, <https://developer.arm.com/documentation/102376/0200/Permission-indirection-and-permission-overlay-extensions>.
- [7] ARM LIMITED. *TTBR0 EL1, Translation Table Base Register 0 (EL1)*, September 2024. <https://developer.arm.com/documentation/ddi0601/2024-09/AArch64-Registers/TTBR0-EL1--Translation-Table-Base-Register-0--EL1->.
 - [8] BLACKBURN, S. M., GARNER, R., HOFFMANN, C., KHANG, A. M., MCKINLEY, K. S., BENTZUR, R., DIWAN, A., FEINBERG, D., FRAMPTON, D., GUYER, S. Z., HIRZEL, M., HOSKING, A., JUMP, M., LEE, H., MOSS, J. E. B., PHANSALKAR, A., STEFANOVIĆ, D., VANDRUNEN, T., VON DINCKLAGE, D., AND WIEDERMANN, B. The dacapo benchmarks: Java benchmarking development and analysis. *SIGPLAN Not.* 41, 10 (oct 2006), 169–190.
 - [9] BLOOM, Z. Cloud Computing without Containers. <https://blog.cloudflare.com/cloud-computing-without-containers>, September 2018. Accessed on 01-31-2024.
 - [10] CASS, S. IEEE Spectrum The Top Programming Languages 2023. <https://spectrum.ieee.org/the-top-programming-languages-2023>, August 2023. Accessed on 07-30-2024.
 - [11] CHIH-EN LIN ET. AL. [RFC PATCH 0/6] Introduce Copy-On-Write to Page Table. <https://lore.kernel.org/all/20220519183127.3909598-1-shiyn.lin@gmail.com/>, March 2022. Accessed on 05-21-2024.
 - [12] CHING-HSIANG, H. T., KEVIN, H., EUGSTER, P., AND PAYER, M. Enforcing least privilege memory views for multithreaded applications. CCS '16, Association for Computing Machinery, p. 393–405.
 - [13] COCK, D., RAMDAS, A., SCHWYN, D., GIARDINO, M., TUROWSKI, A., HE, Z., HOSSLE, N., KOROLIJA, D., LICCIARDELLO, M., MARTSENKO, K., ACHERMANN, R., ALONSO, G., AND ROSCOE, T. Enzian: An open, general, cpu/fpga platform for

- systems software research. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2022), ASPLOS '22, Association for Computing Machinery, p. 434–451.
- [14] COMMUNITY, T. L. K. D. Git repository of Linux 5.15.116. <https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/?h=v5.15.116>, January 2024. Accessed on 01-31-2024.
- [15] DAVE MCCRACKEN. Sharing Page Tables in the Linux Kernel. <https://kernel.org/doc/ols/2003/ols2003-pages-315-320.pdf>, 2003. Accessed on 06-18-2024.
- [16] DAVE MCCRACKEN. Implement shared page tables. <https://lwn.net/Articles/149804/>, August 2005. Accessed on 11-05-2024.
- [17] DILLON, M. DragonFly BSD Sources. <https://www.dragonflybsd.org/>. Accessed on 06-14-2024.
- [18] DUBOSCQ, G., WÜRTHINGER, T., STADLER, L., WIMMER, C., SIMON, D., AND MÖSSENBÖCK, H. An intermediate representation for speculative optimizations in a dynamic compiler. In *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages* (New York, NY, USA, 2013), VMIL '13, Association for Computing Machinery, p. 1–10.
- [19] EL HAJJ, I., MERRITT, A., ZELLWEGER, G., MILOJICIC, D., ACHERMANN, R., FARABOSCHI, P., HWU, W.-M., ROSCOE, T., AND SCHWAN, K. SpaceJMP: Programming with Multiple Virtual Address Spaces. ASPLOS '16, Association for Computing Machinery, p. 353–368.
- [20] ETH ZURICH'S SYSTEMS GROUP. BarrelfishOS/barrelfish-spacejmp. Accessed on 07-03-2024.
- [21] ETH ZURICH'S SYSTEMS GROUP. The Barrelfish operating system. Accessed on 07-03-2024.

- [22] (FENGJING), W. X. Explaining Memory Issues in Java Cloud-Native Practices. https://www.alibabacloud.com/blog/explaining-memory-issues-in-java-cloud-native-practices_599957, May 2023. Accessed on 08-09-2024.
- [23] CVE-2014-0160. <https://nvd.nist.gov/vuln/detail/CVE-2014-0160>, July 2014. Accessed on 05-08-2024.
- [24] HEDAYATI, M., GRAVANI, S., JOHNSON, E., CRISWELL, J., SCOTT, M. L., SHEN, K., AND MARTY, M. Hodor: Intra-Process isolation for High-Throughput data plane libraries.
- [25] HEO, T. Control Groups v2. <https://docs.kernel.org/admin-guide/cgroup-v2.html>. Accessed on 01-31-2024.
- [26] HOSSLE, N., NEUGSCHWANDTNER, M., BLAIR, W., MEIER, R., KUCHLER, T., AND ROSCOE, T. Noodles: Intra-Process-Isolation With Fat Threads. Not yet published.
- [27] INTEL CORPORATION. *Intel(R) 64 and IA-32 Architectures Software Developer Manuals*. Available for download here: <https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html>.
- [28] INTEL CORPORATION. *Intel(R) 64 and IA-32 Architectures Software Developer Manuals*. Page: Vol.3A 7-5, available for download here: <https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html>.
- [29] INTEL CORPORATION. *Intel(R) 64 and IA-32 Architectures Software Developer Manuals*. Page: Vol.3A 4-38, available for download here: <https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html>.
- [30] JEFF XU. Introduction of mseal. <https://docs.kernel.org/userspace-api/mseal.html>. Accessed on 11-07-2024.
- [31] JONATHAN CORBET. Improving shared memory performance. <https://lwn.net/Articles/149888/>, August 2005. Accessed on 11-05-2024.

- [32] JONATHAN CORBET. Five-level page tables. <https://lwn.net/Articles/717293/>, March 2017. Accessed on 02-07-2024.
- [33] KAIWAN N. BILLIMORIA. *Linux Kernel Programming*. Packt Publishing, 2021.
- [34] KIM, Y., DALY, R., KIM, J., FALLIN, C., LEE, J. H., LEE, D., WILKERSON, C., LAI, K., AND MUTLU, O. Flipping bits in memory without accessing them: an experimental study of dram disturbance errors. *SIGARCH Comput. Archit. News* 42, 3 (jun 2014), 361–372.
- [35] LIMITED, A. Arm CoreSight. <https://developer.arm.com/Architectures/CoreSightArchitecture>. Accessed on 09-14-2024.
- [36] THE LINUX MAN-PAGES PROJECT. *capabilities(7) - Linux manual page*, 12 2023. <https://man7.org/linux/man-pages/man7/capabilities.7.html>.
- [37] THE LINUX MAN-PAGES PROJECT. *clone(2) - Linux manual page*, 12 2023. <https://man7.org/linux/man-pages/man2/clone.2.html>.
- [38] THE LINUX MAN-PAGES PROJECT. *fork(2) - Linux manual page*, 12 2023. <https://man7.org/linux/man-pages/man2/fork.2.html>.
- [39] THE LINUX MAN-PAGES PROJECT. *mmap(2) - Linux manual page*, 12 2023. <https://man7.org/linux/man-pages/man2/mmap.2.html>.
- [40] THE LINUX MAN-PAGES PROJECT. *vfork(2) - Linux manual page*, 12 2023. <https://man7.org/linux/man-pages/man2/vfork.2.html>.
- [41] THE LINUX MAN-PAGES PROJECT. *kexec(8) - Linux manual page*, 06 2024. <https://man7.org/linux/man-pages/man8/kexec.8.html>.
- [42] THE LINUX MAN-PAGES PROJECT. *seccomp(2) - Linux manual page*, 06 2024. <https://man7.org/linux/man-pages/man2/seccomp.2.html>.

- [43] THE LINUX MAN-PAGES PROJECT. *shm_overview(7) - Linux manual page*, 06 2024. https://man7.org/linux/man-pages/man7/shm_overview.7.html.
- [44] THE LINUX MAN-PAGES PROJECT. *vDSO(7) - Linux manual page*, 06 2024. <https://man7.org/linux/man-pages/man7/vdso.7.html>.
- [45] LITTON, J., VAHLDIK-OBERWAGNER, A., ELNIKETY, E., GARG, D., BHATTACHARJEE, B., AND DRUSCHEL, P. Light-Weight Contexts: An OS Abstraction for Safety and Performance. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (Savannah, GA, Nov. 2016), USENIX Association, pp. 49–64.
- [46] MAURICE VINCENT WILKES, DAVID JOHN WHEELER, STANLEY GILL. *The Preparation of Programs for an Electronic Digital Computer: With Special Reference to the EDSAC and the Use of a Library of Subroutines*. Addison-Wesley Press, 1951.
- [47] nginx. <https://nginx.org/en/>. Accessed on 05-22-2024.
- [48] NICK PIGGIN. Re: Process creation time increases linearly with shmem. <https://lwn.net/Articles/149893/>, August 2005. Accessed on 11-08-2024.
- [49] NICOARA, A., ALONSO, G., AND ROSCOE, T. Controlled, systematic, and efficient code replacement for running java programs. *SIGOPS Oper. Syst. Rev.* 42, 4 (Apr. 2008), 233–246.
- [50] OpenSSL Cryptography and SSL/TLS Toolkit. <https://www.openssl.org/>. Accessed on 06-03-2024.
- [51] OpenSSL Simple Echo Client/Server. <https://github.com/openssl/openssl/tree/3472732cd23f97f97f50bf0bdc0a7d762fbba/demos/sslecho>, May 2024. Accessed on 06-03-2024.
- [52] ORACLE. Graal Compiler. <https://www.graalvm.org/latest/reference-manual/java/compiler/>. Accessed on 07-27-2024.

- [53] ORACLE. GraalOS - High-performance serverless application deployment platform. <https://graal.cloud/graalos/>. Accessed on 01-31-2024.
- [54] ORACLE. GraalVM. <https://github.com/oracle/graal>. Accessed on 08-15-2024.
- [55] ORACLE. GraalVM - An advanced JDK with ahead-of-time Native Image compilation. <https://www.graalvm.org/>. Accessed on 07-25-2024.
- [56] ORACLE. GraalVM Adoption. <https://www.graalvm.org/use-cases/>. Accessed on 07-30-2024.
- [57] ORACLE. GraalVM Architecture. <https://www.graalvm.org/22.0/docs/introduction/>. Accessed on 07-25-2024.
- [58] ORACLE. GraalVM JavaScript and Node.js Runtime. <https://www.graalvm.org/latest/reference-manual/js/>. Accessed on 07-25-2024.
- [59] ORACLE. GraalVM LLVM Runtime. <https://www.graalvm.org/latest/reference-manual/llvm/>. Accessed on 07-26-2024.
- [60] ORACLE. [graalvm/mx](https://github.com/graalvm/mx). <https://github.com/graalvm/mx/>. Accessed on 07-26-2024.
- [61] ORACLE. Interface IsolateThread. <https://www.graalvm.org/sdk/javadoc/org/graalvm/nativeimage/IsolateThread.html>. Accessed on 08-09-2024.
- [62] ORACLE. Interface ObjectHandles. <https://www.graalvm.org/sdk/javadoc/org/graalvm/nativeimage/ObjectHandles.html>. Accessed on 08-10-2024.
- [63] ORACLE. Isolates for GraalVM Native Images. <https://github.com/graalvm/graalvm-demos/tree/master/native-netty-plot>. Accessed on 08-09-2024.
- [64] ORACLE. Java and JVM. <https://www.graalvm.org/latest/reference-manual/java/>. Accessed on 07-25-2024.

- [65] ORACLE. Java Native Interface (JNI) in Native Image. <https://www.graalvm.org/latest/reference-manual/native-image/dynamic-features/JNI/>. Accessed on 09-19-2024.
- [66] ORACLE. Native Image Basics. <https://www.graalvm.org/latest/reference-manual/native-image/basics/>. Accessed on 09-06-2024.
- [67] ORACLE. Native Image Compatibility Guide. <https://www.graalvm.org/latest/reference-manual/native-image/metadata/Compatibility/>. Accessed on 10-29-2024.
- [68] ORACLE. Polyglot Programming. <https://www.graalvm.org/latest/reference-manual/polyglot-programming/>. Accessed on 07-25-2024.
- [69] ORACLE. The Java HotSpot Performance Engine Architecture . <https://www.oracle.com/java/technologies/whitepaper.html>. Accessed on 07-25-2024.
- [70] ORACLE. Truffle Language Implementation Framework. <https://www.graalvm.org/latest/graalvm-as-a-platform/language-implementation-framework/>. Accessed on 07-25-2024.
- [71] PARK, S., LEE, S., XU, W., MOON, H., AND KIM, T. libmpk (a software abstraction for MPK. <https://github.com/sslabs-gatech/libmpk/>. Accessed on 10-11-2024.
- [72] PARK, S., LEE, S., XU, W., MOON, H., AND KIM, T. libmpk: Software abstraction for intel memory protection keys (intel MPK).
- [73] PAUL MENAGE, PAUL JACKSON, CHRISTOPH LAMETER. Control Groups. <https://docs.kernel.org/admin-guide/cgroup-v1/cgroups.html>. Accessed on 01-31-2024.
- [74] PROJECT, D. DaCapo Benchmarks. <https://www.dacapobench.org/>. Accessed on 10-09-2024.

- [75] PROJECT, D. The benchmarks. <https://dacapobench.sourceforge.net/benchmarks.html>. Accessed on 10-07-2024.
- [76] PROJECT, T. F. The FreeBSD Project. <https://www.freebsd.org/>. Accessed on 06-15-2024.
- [77] PROJECTS, T. C. Memory Safety. <https://www.chromium.org/Home/chromium-security/memory-safety/>. Accessed on 06-20-2024.
- [78] SIMON, D., WIMMER, C., URBAN, B., DUBOSCQ, G., STADLER, L., AND WÜRTHINGER, T. Snippets: Taking the high road to a low level. *ACM Trans. Archit. Code Optim.* 12, 2 (jun 2015).
- [79] THE APACHE SOFTWARE FOUNDATION. Apache MPM worker. <https://httpd.apache.org/docs/2.4/mod/worker.html>. Accessed on 10-14-2024.
- [80] THE LINUX KERNEL DEVELOPMENT COMMUNITY. `do_mseal`. <https://elixir.bootlin.com/linux/v6.12-rc1/source/mm/mseal.c#L212>. Accessed on 10-28-2024.
- [81] THE LINUX KERNEL DEVELOPMENT COMMUNITY. kernel clone. <https://elixir.bootlin.com/linux/v5.15.116/source/kernel/fork.c#L2560>. Accessed on 05-13-2024.
- [82] THE LINUX KERNEL DEVELOPMENT COMMUNITY. Memory Protection Keys. <https://www.kernel.org/doc/html/v5.15/core-api/protection-keys.html>. Accessed on 01-31-2024.
- [83] THE LINUX KERNEL DEVELOPMENT COMMUNITY. `mm_struct`. https://elixir.bootlin.com/linux/v5.15.116/source/include/linux/mm_types.h#L402. Accessed on 05-13-2024.
- [84] THE LINUX KERNEL DEVELOPMENT COMMUNITY. `task_struct`. <https://elixir.bootlin.com/linux/v5.15.116/source/include/linux/sched.h#L721>. Accessed on 05-13-2024.
- [85] THE LINUX KERNEL DEVELOPMENT COMMUNITY. `task_struct` old. <https://elixir.bootlin.com/linux/0.01/source/include/linux/sched.h#L77>. Accessed on 05-23-2024.

- [86] THE LINUX KERNEL DEVELOPMENT COMMUNITY. Tmpfs. <https://docs.kernel.org/filesystems/tmpfs.html>. Accessed on 05-25-2024.
- [87] THE LINUX KERNEL DEVELOPMENT COMMUNITY. vm_area_struct. https://elixir.bootlin.com/linux/v5.15.116/source/include/linux/mm_types.h#L319. Accessed on 05-13-2024.
- [88] THE LINUX KERNEL ORGANIZATION, INC. The Linux Kernel Archives. <https://www.kernel.org>. Accessed on 01-31-2024.
- [89] THE LINUX KERNEL ORGANIZATION, INC. The Linux Kernel Archives - Active kernel releases. <https://www.kernel.org/category/releases.html>. Accessed on 05-08-2024.
- [90] THE NETTY PROJECT. Netty. <https://netty.io/>. Accessed on 10-14-2024.
- [91] THE V8 PROJECT. What is V8? <https://v8.dev/>. Accessed on 10-15-2024.
- [92] TILL SMEJKAL ET. AL. RFC PATCH 00/13 Introduce first class virtual address spaces. <https://lore.kernel.org/linux-arch/20170314020709.vxeglus54k76i7rn@arch-dev/T/>, March 2017. Accessed on 01-31-2024.
- [93] TRÖNDLE, S. Tba. Master's thesis, ETHZ, 2024.
- [94] VAHLDIEK-OBERWAGNER, A., ELNIKETY, E., DUARTE, N. O., SAMMLER, M., DRUSCHEL, P., AND GARG, D. ERIM: Secure, efficient in-process isolation with protection keys (MPK). USENIX Association.
- [95] WATSON, R. N. M., WOODRUFF, J., NEUMANN, P. G., MOORE, S. W., ANDERSON, J., CHISNALL, D., DAVE, N., DAVIS, B., GUDKA, K., LAURIE, B., MURDOCH, S. J., NORTON, R., ROE, M., SON, S., AND VADERA, M. Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (USA, 2015)*, SP '15, IEEE Computer Society, p. 20–37.

- [96] WEIDMANN, M. Arm A-Profile Architecture Developments 2022. <https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/arm-a-profile-architecture-2022>, September 2022.
- [97] WEINGARTEN, M. E. Hardware accelerated trace analysis for compiler optimizations. Master's thesis, ETHZ, 2023.
- [98] WEINGARTEN, M. E., HOSSLE, N., AND ROSCOE, T. High Throughput Hardware Accelerated CoreSight Trace Decoding. Published at DATE, 2024.
- [99] Everything is a file. https://en.wikipedia.org/wiki/Everything_is_a_file, June 2024. Accessed on 07-16-2024.
- [100] Skylake (microarchitecture). [https://en.wikipedia.org/wiki/Skylake_\(microarchitecture\)](https://en.wikipedia.org/wiki/Skylake_(microarchitecture)), October 2024. Accessed on 10-25-2024.
- [101] Usage share of operating systems - Market share by category. https://en.wikipedia.org/wiki/Usage_share_of_operating_systems#Market_share_by_category, June 2024. Accessed on 07-08-2024.
- [102] WIMMER, C. Isolates and Compressed References: More Flexible and Efficient Memory Management via GraalVM. <https://medium.com/graalvm/isolates-and-compressed-references-more-flexible-and-efficient-memory-management-for-graalvm-a044cc50b67e>, January 2019. Accessed on 01-31-2024.
- [103] WIMMER, C., STANCU, C., HOFER, P., JOVANOVIC, V., WÖGERER, P., KESSLER, P. B., PLISS, O., AND WÜRTHINGER, T. Initialize once, start fast: application initialization at build time. *Proc. ACM Program. Lang.* 3, OOPSLA (oct 2019).
- [104] WÜRTHINGER, T., WIMMER, C., WÖSS, A., STADLER, L., DUBOSCQ, G., HUMER, C., RICHARDS, G., SIMON, D., AND WOLCZKO, M. One vm to rule them all. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms*,

- and Reflections on Programming & Software* (New York, NY, USA, 2013), Onward! 2013, Association for Computing Machinery, p. 187–204.
- [105] YANG, F., HUANG, W., KAOUDIS, K., VAHLDIEK-OBERWAGNER, A., AND DAUTENHAHN, N. Endoprocess: Programmable and extensible subprocess isolation. In *Proceedings of the 2023 New Security Paradigms Workshop* (New York, NY, USA, 2023), NSPW '23, Association for Computing Machinery, p. 92–101.
- [106] YUAN, Z., HONG, S., CHANG, R., ZHOU, Y., SHEN, W., AND REN, K. Vdom: Fast and unlimited virtual domains on multiple architectures. ASPLOS 2023, Association for Computing Machinery, p. 905–919.
- [107] ZHAO, K., GONG, S., AND FONSECA, P. On-Demand-Fork: A Microsecond Fork for Memory-Intensive and Latency-Sensitive Applications. In *Proceedings of the Sixteenth European Conference on Computer Systems* (New York, NY, USA, 2021), EuroSys '21, Association for Computing Machinery, p. 540–555. Patch submitted: <https://patchwork.kernel.org/project/linux-mm/patch/20210701134618.18376-1-zhao776@purdue.edu/>.