MANCHESTER 1824
The University of Manchester

TOWARDS SAFE, FLEXIBLE, AND EASY SOFTWARE COMPARTMENTALISATION

A thesis submitted to The University of Manchester
for the degree of Doctor of Philosophy
in the Faculty of Science and Engineering

2024

By
Hugo Lefeuvre

Faculty of Science and Engineering
School of Engineering
Department of Computer Science
The University of Manchester

Contents

Word Count: 44,329

(Excluding Declaration, Copyright, Acknowledgements, Dedication, Appendices, and Bibliography.)

List of Tables

List of Figures

Abstract

Exacerbated by the ubiquity of software, software vulnerabilities increasingly threaten individuals, businesses, and critical infrastructure. One of the root causes of software insecurity is monolithic software design. By designing programs as single units of trust and privilege, the compromise of a single dependency, module, or software part instantly leads to that of the entire program.

Software compartmentalisation provides one way to address this problem. As a software engineering practice where developers break down programs into groups of isolated and distrusting compartments, software compartmentalisation ensures that each part of a program only runs with the smallest set of necessary privileges. In the event of a compromise, compartmentalisation contains the exploit, forcing attackers to deploy significantly more complex attacks to breach software systems.

Yet, despite having proved its worth in the field, software compartmentalisation is still not a popular practice: outside major software applications such as web browsers, mainstream programs remain vastly monolithic. We are missing out on the benefits of software compartmentalisation, at a time when we need them more than ever.

This thesis makes three steps towards mainstream software compartmentalisation.

In a first part, we investigate how to make compartmentalisation faster and safer by specialising compartmentalisation and protection profiles of entire systems towards application workloads. To achieve this we propose FlexOS, a modular operating system which enables users to easily implement highly-specialised compartmentalisation policies, along with a semi-automated method to explore this design space. We show that FlexOS opens for a vast design space, and that this modularity does not come at the cost of performance compared to existing OSes with fixed safety configurations.

In a second part, we investigate *interface vulnerabilities*, the confused deputy vulnerabilities which arise at insufficiently hardened compartment boundaries. We taxonomise these issues, showing that there exist no complete mitigations, and that they affect all known compartmentalisation approaches. We propose ConfFuzz, a fuzzer specialised to detect interface vulnerabilities at possible compartment boundaries, and use it to gather a wide data-set of 629 potential vulnerabilities in real-world software. Systematically studying these issues, we show, among others, that not all interfaces are affected similarly, that API size is uncorrelated with the prevalence of interface vulnerabilities, and that addressing interface vulnerabilities goes beyond writing simple checks.

In a third part, we perform a a large-scale systematisation of knowledge in the field of software compartmentalisation. By identifying and framing existing trends and approaches in software compartmentalisation, along with instances of compartmentalisation that made it into the mainstream, we aim to provide insights on the challenges we still need to tackle to bring the benefits of software compartmentalisation to the mainstream. We show that popularising software compartmentalisation and bringing research advances to the mainstream will require progress towards eliminating the need for developers to manually define compartmentalisation policies (or, when relevant, helping them doing so); towards better framing separation costs early on; to designing abstractions that will stand the test of time and progress; and to better challenging our threat models, particularly in light of interface safety issues.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

i. The author of this thesis (including any appendices and/or schedules to this thesis) owns certain copyright or related rights in it (the "Copyright") and s/he has given The University of Manchester certain rights to use such Copyright, including for administrative purposes.

ii. Copies of this thesis, either in full or in extracts and whether in hard or electronic copy, may be made **only** in accordance with the Copyright, Designs and Patents Act 1988 (as amended) and regulations issued under it or, where appropriate, in accordance with licensing agreements which the University has from time to time. This page must form part of any such copies made.

iii. The ownership of certain Copyright, patents, designs, trade marks and other intellectual property (the "Intellectual Property") and any reproductions of copyright works in the thesis, for example graphs and tables ("Reproductions"), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property and/or Reproductions.

iv. Further information on the conditions under which disclosure, publication and commercialisation of this thesis, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in the University IP Policy[1], in any relevant Thesis restriction declarations deposited in the University Library, the University Library's regulations[2] and in the University's policy on Presentation of Theses.

[1] See http://documents.manchester.ac.uk/DocuInfo.aspx?DocID=24420.
[2] See http://www.library.manchester.ac.uk/about/regulations/.

Acknowledgements

Of course, this has not been an entirely smooth ride! As it should be, I encountered rejection, competing works, slowly realised I know less than I thought I knew, and understand even less than that, encountered CPUs that triple-fault themselves, conceived many unsafe signal handlers, fought for survival with the linker, tracked down many performance bugs, and created many more. Goodness, it's a good thing I wasn't alone.

Yes indeed, concluding this thesis, I truly am indebted to *a lot* of people. I am indebted to those who were by my side when papers were rejected, who helped me realise I ignored and misunderstood, who were with me when the CPU was wrong, when the signal handler raced, when the linker wasn't right, and when the performance was even less right than that. And most importantly, I am indebted to all those who helped me create *great* bugs.

I will begin with my supervisor Pierre Olivier. I have been extremely lucky to have you. Throughout my PhD you have always been there for me, and I never lacked anything. I often wanted to venture into areas where our expertise was limited, and you helped me find out when it was right, and when it wasn't. You helped me see the bright side of things, and you didn't loose hope when I did. Thank you for bringing me books when I was sick in SD. I will read the Sanderson, I promise!

My mentors Felipe Huici and Costin Raiciu. Thank you for bringing me into research, and for teaching me all I needed to get there. You were an essential part of the subtle alchemy of my supervision. I did not realise how lucky I was when I joined NEC!

My friends and co-authors Vlad Bădoiu, Alexander Jung, Gaulthier Gain. Together, we spent such great nights creating the most delightful bugs – why, we broke nearly every single part of the OS! I will always look back to these deadline pushes with nostalgia.

My many co-authors, friends, mentors, and students, who worked with me on various papers throughout my PhD. Among others Nathan Dautenhahn, Marios Kogias, David Chisnall, Simon Kuenzer, Razvan Deaconescu, Alistair Kressel, Stefan Teodorescu, Sebastian Rauch. Thank you, I learned so much from all of you.

My colleagues and friends from the APT group at the University of Manchester, Nikolaos Kyparissas, Luca Peres, Igor Wodiany, Konstantinos Iordanou, and many others. Thank you for all the coffee breaks, random chats, dinners, beers, etc. You definitely made me want to go to the office! Nick, thank you for bringing a little bit of Manchester with you to Cambridge. We should resume our dinners once this madness is over.

My colleagues and mentors from the Confidential Computing group at Microsoft Research Cambridge, Adrien Ghosn, Istvan Haller, Sylvan Clebsch, Matthew Parkinson, Boris Köpf, Manuel Costa, (and again David Chisnall and Marios Kogias), and many others. Thank you for making this internship so wonderful. My interactions with the whole of the Confidential Computing group left a lasting impression on me.

The Unikraft and Debian communities. These are far too many people to list here. To all my Debian friends: you made me love systems. To all my Unikraft friends: you made me love systems even more.

My friends from France, Germany, Canada, from the UK, and from everywhere else. Thank you for being with me, even when I disappeared for months (years?) working on this thesis. Michael, thank you for our (ir)regular calls, it meant a lot to me.

My family. As a kid, I asked you a lot of questions. Maman, how long does school last? Well, I was horrified when you told me how many years I had left until I would finish high school. Goodness, this thing will never finish. Maman, Papa, does anyone on this earth know everything? Ah, OK, well, that's unfortunate. But "les chercheurs", pushing the boundaries of human knowledge…now that sounds *cool*. Thank you for answering all my questions. Thank you for pushing me to ask more questions.

Lastly, my wife, 佳佳. I am so lucky to have you. Thank you for being with me at the office on week-days, week-ends, days and nights, when my papers were accepted, when they weren't, when I was sick, angry, hungry. This thesis is as much yours as it is mine. I love you.

Manchester, on April 12, 2024 *Hugo Lefeuvre*

Note on Funding. Funding for each of Chapters 3 to 5 are detailed in their "Acknowledgements" sections. I am especially grateful to NEC Labs Europe, MSR Cambridge, and the University of Manchester for awarding me fellowships which enabled this PhD.

Dedication

Pour Mamie Lu.

Chapter 1

Introduction

Software is everywhere: healthcare, transportation, finance, energy, defence, communications, agriculture, and more. Through this vast deployment, promised to go on in the coming years, software has led to massive improvements in the living standard of billions of humans. Yet, this progress is double-edged: it made us more than ever vulnerable to malicious actors. Exacerbated by the ubiquity of software, software vulnerabilities now threaten individuals, businesses, and critical infrastructure [337]: over the past years, all across the world, cyber threats caused service loss and data leaks in hospitals and local governments, food and fuel supply chain disruptions, leaks of enterprise and personal data, among others [254, 151, 433]. More than ever, software security is in the spotlight.

One of the root causes of this problem is the way we design software. Typically, software is designed and implemented assuming it does not contain bugs. When a security vulnerability is found, developers conceive and deploy a patch addressing it. This reactive process is repeated over the lifetime of the software. Though effective to address known bugs, this approach fundamentally cannot mitigate *future, unknown* flaws, merely maintaining the illusion that the software has no bugs. This is problematic, as we know that any reasonably complex software system *will* have many bugs over its lifetime. Alas, before they are patched, bugs ("zero-days") represent an easy attack vector for attackers without which "there is almost no defence" [140]. Without proactively considering these future, unknown flaws in the way we design software, attackers will always be one step (bug) ahead.

Software compartmentalisation provides one way to address this problem. An embodiment of the principle of least privilege [392], software compartmentalisation (also

referred to as "privilege separation" in the literature [373]) is a software engineering practice where developers break down a program into groups of isolated and distrusting components (called *compartments*), each only privileged to control the resources they require to function. Software compartmentalisation comes in opposition with the standard practice of developing programs as a single unit of privilege. If part of a compartmentalised program is compromised (e.g., through a bug), attackers will be confined to the restricted privileges of the compromised compartment, and will need to deploy further attacks to compromise the rest of the system.[1]

Over the past decades, compartmentalisation[2] has been the target of a vast body of research, which demonstrated its ability to contain memory safety issues [145, 384], to provide general fault isolation or resilience [311, 346], as well as to isolate untrusted dependencies [110, 344], thwart supply-chain attacks [438, 213], side-channels [256, 345, 338], and many other threats.[3] Outside research, compartmentalisation showed a concrete impact on bug exploitability where it was pushed to production. In the OpenBSD community, the compartmentalised architecture of OpenSSH and slaacd successfully mitigated otherwise critical code-execution flaws [20, 68, 15, 16]. Similar observations were made for Nginx workers [11, 18], and web browser site isolation [163, 447, 378]. For all these reasons, compartmentalisation has recently attracted renewed interest from the industry and national funding agencies [182, 253].

Still, to this day, despite their benefits, compartmentalisation techniques remain rarely adopted outside security-aware developer circles. Even when it comes to isolating cryptographic secrets, a rather obvious application of compartmentalisation long pushed by many works [258, 225, 395, 227, 133, 315, 434, 147, 166, 141, 415, 226, 296, 313], approaches proposed in research are not adopted in practice by popular cryptography libraries. We are missing out on the security benefits compartmentalisation can bring, at a time when systems security has become more important than ever.

Why so? The answer is likely multifaceted. The performance cost of compartmentalisation is still an issue, most approaches remaining well over the 5-15% performance overhead typically mandated as acceptable, e.g., by funding agencies [182]. Security too is still an issue: beyond the difficulty to achieve fine-granular separation in practice, the concrete security benefits of compartmentalisation remain widely unquantified and unstudied. This lack of formal and quantifiable security guarantees makes compartmentalisation harder to popularise. Finally, as we show later in this thesis, the concept of

[1] We provide a more complete definition of software compartmentalisation in Section 2.1.1.
[2] For conciseness, we will often refer to software compartmentalisation as simply *compartmentalisation*.
[3] We discuss these applications of software compartmentalisation in greater detail in Section 2.1.2.

software compartmentalisation itself remains unclear to the mainstream and even to our wider research community: due to the scale of compartmentalisation research, few experts have a global overview of the progress which has been made, and which paths remain to be taken to bring compartmentalisation benefits to the mainstream.

On the performance side, achieving overheads below 5-15% will require practitioners and researchers to go well beyond simply optimising cross-compartment switch costs. One of the ways to meet these performance requirements is specialisation [200]: to achieve such low overheads, we should fully leverage workload-specific security/performance trade-offs. We should make it possible to tune isolation policies at a fine grain (granularity, boundary placement, guarantees provided), at the scale of the entire operating system; as well as to leverage the most appropriate isolation primitives, hardware- or software-based, for each task. On the technical side, this calls for compartmentalisation frameworks that are more *flexible* to be able to implement such highly-specialised policies; but also for automated or semi-automated design-space exploration methods to help non-expert users leverage this flexibility in practice. We explore approaches to both problems in Chapter 3 with FlexOS, a modular Operating System (OS) which enables users to implement highly-specialised compartmentalisation policies, and a semi-automated design-space exploration method based on partially-ordered sets.

On the security side, achieving strong and tangible security properties in the general case with compartmentalisation, and ideally security *guarantees*, will require a better understanding of the vulnerabilities that affect compartmentalised software. Most concerning are the confused deputy vulnerabilities [234] that arise at compartment boundaries [244, 344]. Regrettably, there exists no comprehensive study dedicated to these vulnerabilities, and most compartmentalisation works in research and in practice simply scope them out. We need to understand how widespread these vulnerabilities are, particularly when retrofitting compartmentalisation into previously monolithic codebases, what design patterns lead to these problems, how they impact the security benefits of compartmentalisation, which counter-measures exist, and how effective they are. We explore these questions in Chapter 4 with a large-scale study of compartment-interface vulnerabilities using ConfFuzz, a fuzzer specialised to detect confused deputy vulnerabilities at compartment interfaces.

Finally, to focus compartmentalisation research on a track that will bring it to the mainstream, we critically need a large-scale systematisation of knowledge effort in the field of software compartmentalisation. This effort should bring a theoretical substrate for compartmentalisation, characterising existing approaches into a framework that can highlight solved problems and gaps in the state of the art, that can systematise

mainstream needs and how research approaches address (or do not address) them, and provide insights on future software compartmentalisation challenges. We explore these questions in Chapter 5 with a comprehensive systematisation of knowledge effort, concluding this manuscript with a look forward going far beyond this thesis.

1.1 Research Aims and Objectives

In this dissertation, we aim to contribute to the field of software compartmentalisation in three ways: 1) reducing compartmentalisation performance costs through specialisation and flexibility; 2) making a step towards stronger security properties by better understanding the new classes of security vulnerabilities that arise in compartmentalised systems and how we should go about to solve them; and 3) bringing a global overview on compartmentalisation approaches, highlighting gaps, mainstream needs, and future challenges. Specifically, we aim to address the following research questions (**RQ**):

RQ1 *How can we enable users to easily and safely specialise the compartmentalisation and protection profile of the entire system towards their application workloads?* We are interested in understanding implications on operating system design: how should this specialisation be materialised in software components throughout the entire system stack, and with which programming abstractions? How should these abstractions be designed to support as many isolation mechanisms, hardware and software, as possible? Beyond design considerations, which performance benefits can we achieve with such flexibility? Conversely, does such flexibility come at any performance cost?

RQ2 *How can we guide users to navigate the design space enabled by flexible systems?* Such specialisation opens for a large configuration space which may be difficult for non-expert users to explore manually. Still, fully automated design-space exploration is difficult, due to the lack of good metrics for quantifying security. We are thus particularly interested in semi-automated exploration techniques to help users make relevant configuration choices.

RQ3 *How widespread are confused deputy vulnerabilities when compartmentalising unmodified applications, what is their impact on the security properties achieved, and how effective are existing protections?* Although the problem of confused deputies

in compartmentalised software is long known [234, 397, 373], most compartmentalisation works in research and in practice scope them out, and no comprehensive study have been dedicated to the problem. Answering these questions is necessary to achieve adequate and principled mitigations. We are interested in both theoretical and quantitative answers. On the theoretical side, which classes of confused deputies are possible, what are their possible impacts, and which existing protections can mitigate them? On the quantitative side, what is the prevalence of these classes of vulnerabilities in real-world compartmentalisation scenarios, and what is their concrete impact?

RQ4 *Which software design patterns characterise stronger or weaker security domain interfaces, and what is the complexity of refactoring boundaries to address these weaknesses?* Going beyond **RQ3**, we are interested in determining to what extent software design patterns lead to stronger or weaker compartment boundaries, what these patterns mean in terms of refactoring costs, and how they should impact the way we design interfaces for distrust. We are also interested to understand whether simple metrics such as the size of interfaces, or the quantity of shared data, are correlated with the vulnerability of interfaces.

RQ5 *How should a theoretical framework for compartmentalisation look, to characterise past and future works?* A large-scale systematisation of software compartmentalisation is needed to highlight solved problems and gaps in the field. We are interested in identifying and framing existing trends and approaches in software compartmentalisation, along with key characteristics to differentiate them.

RQ6 *How does the state of the art in research relate to existing approaches and needs in the mainstream, and what does this tell us about where compartmentalisation research should go next?* Building on **RQ5**, we are interested in studying compartmentalised software that made it to the mainstream and its characteristics, to identify discrepancies, and gaps, with existing research trends. Looking forward, what problems should compartmentalisation research be focusing on?

1.2 Research Contributions

This thesis answers our research aims through the following core contributions:

- The design, implementation, and evaluation of *FlexOS*, a modular OS design

whose compartmentalisation and protection profile can easily be tailored towards a specific application or use-case at build time. Presented in Chapter 3, this contribution addresses **RQ1**.

- A semi-automated technique named *partial safety ordering*, which uses partially ordered sets to describe the probabilistic security degrees of FlexOS configurations and identify the safest ones under a given performance budget. Presented in Chapter 3, this contribution addresses **RQ2**.
- A theoretical definition, systematisation, and taxonomy of compartment-interface vulnerabilities and existing efforts to address them. Presented in Chapter 4, this contribution addresses **RQ3**.
- The design and implementation of *ConfFuzz*, an in-memory fuzzer that automatically detects compartment-interface vulnerabilities in existing software at arbitrary interfaces, along with a quantitative study of compartment-interface vulnerabilities found by ConfFuzz applied to 39 real-world application compartmentalisation scenarios. Presented in Chapter 4, this contribution addresses **RQ3**.
- A systematic study of interface vulnerability patterns backed by our ConfFuzz data set, along with a series of interface design guidelines intended to ease the development and adaptation of new and existing interfaces with compartmentalisation in mind. Presented in Chapter 4, this contribution addresses **RQ4**.
- A comprehensive systematisation of existing partitioning approaches, compartmentalisation abstractions, and mechanisms, and their limitations. Presented in Chapter 5, this contribution addresses **RQ5**.
- A study of real-world compartmentalised software that highlights the characteristics of approaches that made it to the mainstream, and what should be learned from them. Presented in Chapter 5, this contribution addresses **RQ6**.
- A discussion of challenges to be tackled to make compartmentalisation a truly mainstream practice. Presented in Chapter 5, this contribution addresses **RQ6**.

1.3 Thesis Overview

This thesis is presented in a *journal* format. This means that the core materials of this thesis, Chapters 3 to 5, are self-standing works published, or under submission for publication, in peer-reviewed scientific venues. The rationale for submitting this thesis in

a journal format is that all contributions of this manuscript have been, or are destined to be independently peer-reviewed and published in scientific conferences. Rewriting them in a standard-format manuscript would represent an effort counter-productive to maximising the amount and overall quality of the contributions of this manuscript.

Overall, Chapters 3 to 5 (along with Appendices A and B) are included largely unchanged in their published or submitted form. A small number of edits have been performed to increase the consistency across chapters, improve clarity, and visual value. In order to maintain a coherent flow across the different publications which constitute this thesis, we prepend each of Chapters 3 to 5 with a thesis context paragraph, which positions each publication in the bigger picture of the thesis. Since authors other than the author of this thesis contributed to each of the publications, we also clarify, for each chapter, the extent of our contributions in a "Contributions of the Author" section.

Finally, we note that Chapters 3, 4, and 5 are included in their original language, US English. Chapters 1, 2 and 6 are written in UK English as per convention of the Doctoral Academy[4]. Next, we provide an overview of this manuscript, detailing the publications relevant to each chapter.

Chapter 2 provides background on the topic of software compartmentalisation. We introduce, scope, and further motivate compartmentalisation and the problem of interface hardening in compartmentalisation. On this basis, we motivate the contributions presented in Chapters 3 to 5.

Chapter 3 presents FlexOS, our approach to specialisation for security, along with our semi-automated design-space exploration strategy. As part of this, we motivate and describe the design of FlexOS as a modular library operating system, and of its mechanism-agnostic compartment abstraction. We briefly describe the implementation of FlexOS on top of Unikraft with support for two hardware-enforced isolation mechanisms, Intel Protection Keys for Userspace (PKU) and the Extended Page Table (EPT), and evaluate it on popular cloud applications. We present our semi-automated design-space exploration method based on partially-ordered sets, and evaluate it on the FlexOS configuration space. This chapter is derived from the following publication:

[299] H. Lefeuvre, V-A. Bădoiu, A. Jung, S. Teodorescu, S. Rauch, F. Huici, C. Raiciu, P. Olivier; *FlexOS: Towards Flexible OS Isolation*; ASPLOS 2022.

Chapter 4 presents a study of interface vulnerabilities in compartmentalised software. We define interface vulnerabilities, classify them, and discuss how existing protections can (or fail to) mitigate them. We then describe the design and implementation

[4]This notably leads to *compartmentalisation* being spelled differently (with "s" vs. "z") across the thesis.

of ConfFuzz, a fuzzer specifically designed to detect interface vulnerabilities at arbitrary compartment boundaries. We apply ConfFuzz to 39 application compartmentalisation scenarios to uncover a data-set of 629 potential vulnerabilities, which we dissect to extract insights on the prevalence of interface vulnerabilities, their causes, impact, and the complexity to address them. This chapter is derived from the following publication:

[298] H. Lefeuvre, V-A. Bădoiu, Y. Chien, F. Huici, N. Dautenhahn, P. Olivier; *Assessing the Impact of Interface Vulnerabilities in Compartmentalized Software*; NDSS 2023.

Chapter 5 presents a systematisation of knowledge study in the field of software compartmentalisation. We carefully define and frame software compartmentalisation. On this basis, we systematise 191 compartmentalisation efforts spreading the last 60 years into a consistent and structured field. We confront this study with an analysis of 60 compartmentalised programs that did reach the mainstream. Based on the study's insights, we conclude with a discussion of open problems to address for compartmentalisation to become a truly widespread engineering practice, and for research advances to make it into production. This chapter is derived from the following publication:

[N/A] H. Lefeuvre, N. Dautenhahn, D. Chisnall, P. Olivier; *SoK: Deconstructing Software Compartmentalization*; Under Submission.

Chapter 6 concludes this thesis by summarising our contributions, and what we should learn from them. We draw future works, and discuss the longer-term implications of our insights on the field.

1.4 Selected Publications

The contributions presented in this thesis fit in a broader ecosystem of research conducted with a wider group of co-authors over the past three years. Three of these publications [300, 299, 298] (along with one under submission) constitute the main three chapters of this thesis. Two other publications [301, 302] led by the author of this thesis within the timeframe of the thesis have been included in Appendices A and B. The rationale for including these works as appendices instead of main chapters is that they explore research problems which do not harmoniously fit in the main storyline of the manuscript. We expand on this in a context paragraph in each of Appendices A and B.

To provide a complete picture of this research ecosystem, a selected list of publications realised within the timeframe of this thesis is provided next in thematic order:

Specialising Operating Systems for Security.

[283] J.A. Kressel, H. Lefeuvre, P. Olivier; *Software Compartmentalization Trade-Offs with Hardware Capabilities*; PLOS 2023.

[299] H. Lefeuvre, V-A. Bădoiu, A. Jung, S. Teodorescu, S. Rauch, F. Huici, C. Raiciu, P. Olivier; *FlexOS: Towards Flexible OS Isolation*; ASPLOS 2022; in Chapter 3.

[300] H. Lefeuvre, V-A. Bădoiu, S. Teodorescu, P. Olivier, T. Mosnoi, R. Deaconescu, F. Huici, C. Raiciu; *FlexOS: Making OS Isolation Flexible*; HotOS 2021; this is an earlier workshop release of the content included in Chapter 3.

Specialising Operating Systems for Performance.

[353] P. Olivier, H. Lefeuvre, D. Chiba, S. Lankes, C. Min, B. Ravindran; *A Syscall-Level Binary-Compatible Unikernel*; IEEE Transactions on Computers, 2022.

[285] S. Kuenzer, V-A. Bădoiu, H. Lefeuvre, S. Santhanam, A. Jung, G. Gain, C. Soldani, C. Lupu, S. Teodorescu, C. Răducanu, C. Banu, L. Mathy, R. Deaconescu, C. Raiciu, F. Huici; *Unikraft: Fast, Specialized Unikernels the Easy Way*; EuroSys 2021.

Making it Easier to Design and Specialise Operating Systems.

[302] H. Lefeuvre, G. Gain, V-A. Bădoiu, D. Dinca, V-R. Schiller, C. Raiciu, F. Huici, P. Olivier; *Loupe: Driving the Development of OS Compatibility Layers*; ASPLOS 2024; in Appendix A.

[261] A. Jung, H. Lefeuvre, C. Rotsos, P. Olivier, D. Oñoro, M. Niepert, F. Huici; *Wayfinder: Towards Automatically Deriving Optimal OS Configurations*; APSys 2021.

Understanding and Solving the Problem of Interface Safety.

[168] Y. Chien, V-A. Bădoiu, Y. Yang, Y. Huo, K. Kaoudis, H. Lefeuvre, P. Olivier, N. Dautenhahn; *CIVSCOPE: Analyzing Potential Memory Corruption Bugs in Compartment Interfaces*; KISV 2023.

[301] H. Lefeuvre, D. Chisnall, M. Kogias, P. Olivier; *Towards (Really) Safe and Fast Confidential I/O*; HotOS 2023; in Appendix B.

[298] H. Lefeuvre, V-A. Bădoiu, Y. Chien, F. Huici, N. Dautenhahn, P. Olivier; *Assessing the Impact of Interface Vulnerabilities in Compartmentalized Software*; NDSS 2023; in Chapter 4.

Chapter 2

Background and Motivation

This chapter provides background to support and motivate the contributions presented in Chapters 3 to 5. We introduce, scope, and motivate software compartmentalisation, before providing background on the problem of interface safety and its challenges.

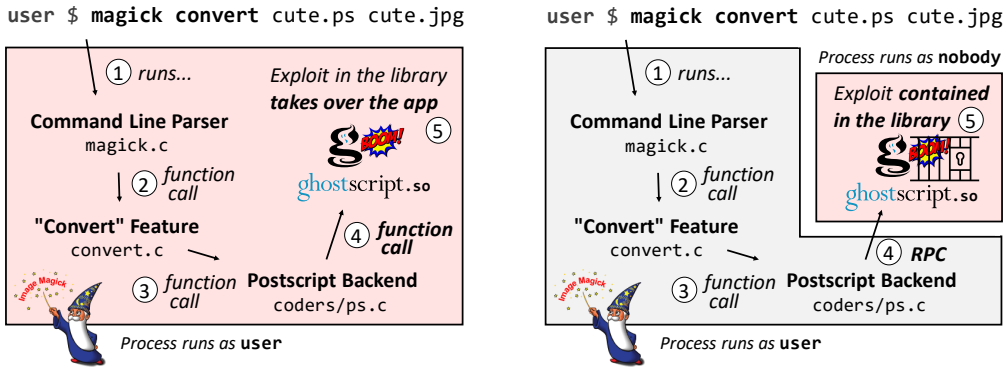
Note that, due to the journal format of this dissertation, Chapters 3 to 5 are self-contained and thus all present relevant background and related works. Still, this chapter is not redundant with the rest of this manuscript. Unlike the backgrounds of Chapters 3 to 5, which tend to be compact due to the page limit of conference publications, this chapter provides a more accessible and unified introduction to the topics of this dissertation for a general audience.

2.1 Software Compartmentalisation

In this first section, we introduce the concept of software compartmentalisation and illustrate it with a real-world example. We motivate the use of compartmentalisation in different contexts, and conclude presenting different approaches to software compartmentalisation from the literature. We motivate Chapters 3 and 5 based on this overview.

2.1.1 What is Software Compartmentalisation?

Software compartmentalisation is a software engineering practice where developers break down a program into groups of isolated and distrusting components, called *compartments*, each controlling only the resources they precisely require. Software compartmentalisation is an embodiment of the principle of least privilege [392]: by ensuring that each part of the program only runs with the least set of privileges necessary



(a) Monolithic Scenario

(b) Compartmentalised Scenario

Figure 2.1: Execution of `magick convert` on malicious input, in two scenarios: *monolithic* (left, ImageMagick's architecture), and *compartmentalised* (right). The malicious input traverses ImageMagick (① to ④) and is passed on to the Ghostscript library, which processes it and is compromised (⑤), e.g., through an arbitrary code-execution vulnerability [21, 22]. In the monolithic scenario, the whole of ImageMagick and `user` is compromised. In the compartmentalised scenario, the attacker only gains control of an unprivileged dummy user `nobody`, and must leverage additional vulnerabilities in ImageMagick or the kernel to escalate privileges.

to complete its job, we can reduce the damage caused by a bug or compromise. Concretely, if a software component is compromised in a compartmentalised program, for example through a zero-day bug [140], attackers will be restricted to the permissions of the compromised component. To escalate privileges and compromise the rest of the system, attackers will need to find and exploit additional software vulnerabilities.

Example. Let us illustrate this with a real-world example. ImageMagick [42] is a popular command-line image processing tool. When deployed, ImageMagick is often fed with untrusted graphics input, e.g., to convert a network-provided image from one file format to another, as is the case in Figure 2.1 (`cute.ps` → `cute.jpg`). ImageMagick relies on external libraries to process images: for example the Ghostscript [33] library for the PostScript [111] format. Due to the complexity of their task (parsing complex file formats), libraries such as Ghostscript are highly prone to security vulnerabilities: in 2023 alone, Ghostscript was affected by at least two code execution flaws [21, 22]. In a monolithic (non-compartmentalised) scenario, visualised in Figure 2.1a, such vulnerabilities have a high impact, as they allow an attacker to instantly take over ImageMagick, granting them the permissions of the user under which ImageMagick is running (`user` in Figure 2.1a). This is because the Ghostscript library is running in the

context of ImageMagick, with the permissions of ImageMagick. This is a typical case for software compartmentalisation: if we had instead designed ImageMagick to run Ghostscript in a sandbox compartment (e.g., in a separate process running under an unprivileged dummy user `nobody` as in Figure 2.1b), a compromise of the Ghostscript library would not grant the attacker critical permissions, forcing them to find and exploit additional software vulnerabilities to escape the Ghostscript library sandbox.

Although this example illustrates software compartmentalisation in an application, compartmentalisation can be applied to any program in the general sense: applications, OS kernels (microkernels [311], for example, are compartmentalised kernels), hypervisors [408], firmware [269], etc. Compartmentalisation can be *retrofitted* (i.e., integrated into formerly monolithic programs), or present in the initial design of a program.

Other isolation practices. In this thesis, we refer to software compartmentalisation as the practice of breaking down *one program* (e.g., one application, kernel, or hypervisor), similarly to a generalised definition of Provos et al. [373]'s *privilege separation*[1]. As we show throughout Chapters 3 to 5, this covers a vast and coherent body of work. Other isolation techniques [409] *across* programs or groups of programs are not here referred to as software compartmentalisation: e.g., isolation of applications on a commodity OS, whole-application sandboxing [192, 82, 217, 126], separation between user and kernel [165, 240], between virtual machines [131] hosting distinct programs, between stakeholders (confidential computing [391]), between users, or containers [335]. Though these isolation works share techniques and challenges with software compartmentalisation as we define it, we focus strictly on the latter for scope reasons.

2.1.2 Why Software Compartmentalisation?

In the previous section, we introduced the concept of software compartmentalisation together with a practical use-case. Next, we will further motivate software compartmentalisation by giving a more complete perspective on its applications (**A**). Drawing from this, we will discuss the threat models of compartmentalisation.

A1: Compartmentalising to contain memory-safety issues. Containing memory-safety issues is a popular use-case of software compartmentalisation: software parts which are more likely to feature memory-safety bugs (e.g., performing error-prone

[1] Provos et al. [373] define privilege separation for applications, but their definition can be extended to programs in the general sense.

tasks such as parsers), or components where bugs are most likely to be exploited (e.g., those facing the network) are isolated into an unprivileged compartment (*sandbox*) to limit the impact of future, unknown bugs. By virtue of memory isolation, memory-safety issues affecting these components are then contained within their originating compartment. Our running example (Figure 2.1) is a typical case of containing memory-safety issues: the error-prone Ghostscript library is compartmentalised to prevent potential memory-safety flaws from reaching to the ImageMagick application.

Somewhat counter-intuitively, containing memory-safety issues is also relevant for safe languages. Indeed, safe languages such as Rust, Go, or Java, enable users to express "unsafe" code, which too can be affected by memory-safety issues. This unsafe code can be compartmentalised to preserve safety guarantees [115, 317, 417, 128]. Similarly, programs written in safe languages may also need to interface with code written in an unsafe language, e.g., because they need to use dependencies which have not been re-written in a safe language. These unsafe dependencies too can be compartmentalised to protect safe code [290, 115, 383, 275, 128]. Lastly, prior work [334] showed that even composing different safe languages can cause memory-safety issues due to mismatches in the safety assumptions of different safe languages. Compartmentalising each part written in a different language can prevent such issues as well [334].

A2: Compartmentalising to protect against untrusted dependencies. Another application of software compartmentalisation is protecting a program against untrusted dependencies. Such cases take place when either the supply chain [289] or the dependency itself[2] are distrusted or considered compromised. In a monolithic program, the impact of relying on an untrusted dependency corresponds to granting attackers with permissions to execute arbitrary code, regardless of whether or not the dependency is written in a safe language. Here too, sandboxing can help mitigate the impact of a malicious or compromised dependency [316, 247, 213, 289].

A3: Compartmentalising to protect critical data and code. Programs often embed data which demands special protection. Such data includes program secrets (e.g., cryptographic keys, passwords and credentials, more broadly authentication data), as well as data critical to enforcing security mechanisms (e.g., shadow stacks [150], CPI

[2]Example: advertisement and analytics libraries, particularly on mobile devices, pose well-established privacy problems, which often causes them to be (at least partially) distrusted [316, 247]. Another case for distrust are software package registries such as npm [62], known to host packages which, under the cover of a legitimate functionality, expose malicious behaviour [478].

and CPS regions [287], randomisation information [405], or page tables in the kernel [198]). Similarly, programs often embed code sections which require elevated privileges, for instance performing a system call such as `setuid` or `setgid` to change user IDs, known to cause serious vulnerabilities when subverted [162]. In all cases, compartmentalisation can be employed to prevent illegitimate accesses to critical data or code. This type of compartmentalisation differs from **A1** and **A2** in that it is not directed at the part of the program which is distrusted, but at the one which is trusted. We refer to this model as *safebox*, by opposition to the sandbox model (**A1-2**). An example of this use-case is isolating OpenSSL to protect cryptographic keys [434].

A4: Compartmentalising to protect against transient execution attacks. Going beyond protecting program parts against conventional memory-based vulnerabilities, compartmentalisation can also be employed as a mitigation technique against intra-process Spectre attacks [156]. Prior works [256, 345] explored safeboxing critical data (similar to **A3**) to ensure that illegitimate speculative reads targeting valuable data fault before caching, either via hardware extensions [256], or by compiling compartment code to enforce specific control-flow properties [345]. Although we briefly discuss this use-case in Chapter 5, we generally consider Spectre and transient execution attacks out of scope in this thesis (cf. each chapter's threat model).

A5: Compartmentalising to make software more reliable. If one component of a monolithic program crashes, becomes unresponsive, or misbehaves, so does the entire program. Software compartmentalisation can be leveraged to increase program reliability: by introducing boundaries between components, individual program parts can be restarted without impacting the whole, provided program state is appropriately segregated. In software compartmentalisation, reliability has historically been pursued with microkernels [468, 311, 238, 278, 199], and with application compartmentalisation frameworks such as Google SAPI [35]. Although we discuss this use-case in Chapter 5, software reliability (and denial of service) is generally out of scope in this thesis.

The threat models of compartmentalisation. As illustrated throughout **A1-5**, the applications of compartmentalisation are numerous. For that reason, there is no single threat model in software compartmentalisation: depending on the use-case, the threat model will vary widely. For example, applications such as **A1** and **A2** typically target integrity and confidentiality, but some also consider availability, and yet others target integrity only. Usages such as **A5** generally target integrity, confidentiality,

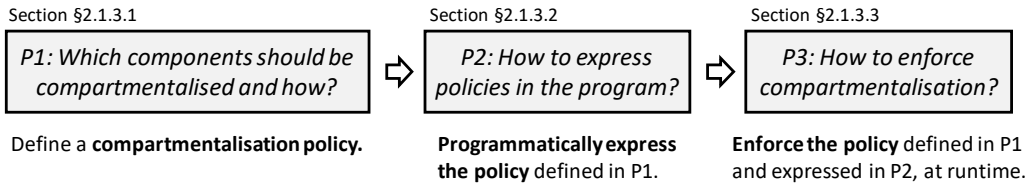


Figure 2.2: Overview of the three problems of software compartmentalisation as contributed in Chapter 5. We use this classification to structure this chapter.

and availability. One common rule is that compartmentalisation approaches generally must enforce at least integrity, as integrity is necessary to achieve other properties. In this thesis, Chapters 3 and 4 are focused on confidentiality and integrity. Chapter 5 generalises and discusses the different threat models of compartmentalisation in detail.

2.1.3 Approaches to Software Compartmentalisation

We have now introduced and motivated software compartmentalisation. Next, we will give an overview of approaches to compartmentalise software, and discuss how this state of the art motivates the contributions of this thesis, particularly Chapters 3 and 5.[3] To give a structured overview of existing compartmentalisation approaches, we adopt the classification we contribute later in §5.3. In this classification, visualised in Figure 2.2, we propose to see compartmentalisation as a set of three problems (**P**):

2.1.3.1 P1: Which components should be compartmentalised, and how?

When compartmentalising a program, developers must first define which components are to be separated and which properties should be enforced on the resulting compartments. The result is a *compartmentalisation policy*, which describes the compartmentalisation to enforce. Policies have a decisive impact on the security properties which can be achieved through compartmentalisation and on their associated performance cost. Concretely, defining a compartmentalisation policy consists of the following tasks:

1. Defining which parts of the program to separate. Developers must make boundary decisions on program control flow (where should compartment switches be placed?), as well as on program data (should a given object be shared? if so, should it be shared as a whole or at a finer-grain to better enforce least privilege?).

[3] Chapter 4 will be motivated by §2.2.

2. Defining which permissions to enforce. Developers must define permissions for private data (should integrity and/or confidentiality be enforced? or possibly others properties?) as well as shared data (should the compartment be allowed to read, write, or execute a given shared object?). When compartmentalising applications, developers must also define permissions at system boundaries (which system calls should the compartment be able to execute? which permissions should it have on OS constructs such as files or sockets?).[4] Finally, developers must also decide on the strategy to employ to react to permission violations (should the program be terminated, or the faulting component be restarted?).

We refer to the set of all possible compartmentalisations of a program (i.e., all possible decisions for each of steps 1. and 2.) as the *compartmentalisation design space*.

Example. Relating this to our running example, the result of this step when compartmentalising ImageMagick is deciding to (1) separate at the Ghostscript library API, at a coarse object granularity, using (2) a sandbox which enforces integrity and confidentiality on private data, read and write permissions for all shared objects, terminating on domain violations. Many other options could have been considered. Regarding control-flow boundaries, compartmentalisation could be performed at a finer grain within Ghostscript, or at a coarser grain at the level of the ImageMagick internal backend API. A safebox approach (as in **A3**) would be another possibility, compartmentalising only the few privileged parts of ImageMagick, such as those which perform system calls to operate on files. Regarding permissions, enforcing only integrity instead of integrity *and* confidentiality would be possible to trade off some security for performance. Instead of terminating the program on domain violations, it would also be possible to automatically restart a fresh instance of the Ghostscript compartment to increase resilience (as in **A5**), as little state is shared between ImageMagick and the isolated library.

Automating policy definition. Defining compartmentalisation policies has historically been a manual task, where experts design or refactor programs leveraging deep knowledge of their semantics. This is the case of microkernels [468, 311, 238, 278, 199] and of most cases of application compartmentalisation since the late 1990s (including influential examples such as qmail [137, 230], OpenSSH [373], or Postfix [231]). An important drawback of this approach is that manually defining compartmentalisation

[4] Similar needs arise when partitioning OS kernels and other types of programs, e.g., instead of system calls, hypercalls and/or hardware operations. For conciseness we do not discuss these here.

policies requires significant expertise, making it difficult for software compartmentalisation to become a truly mainstream software engineering practice. For this reason, in the last decade, researchers have explored semi-automated, as well as fully automated approaches to separate software. Influential approaches include:

- Semi-automated approaches based on code annotations, such as SOAAP [227], SeCage [320], or PtrSplit [318], which rely on developer-provided annotations describing shared, confidential, and/or sensitive objects or code portions. The tool automatically derives a policy based on these annotations.
- Other semi-automated approaches which rely on high-level developer instructions, such as BreakApp [438], ACES [171], Enclosure [213], CubicleOS [395], or Cali [133]. Developers provide the tool with instructions such as "place library X into a compartment", or "file F is distrusted", and the tool automatically derives a concrete policy based on these annotations.
- Yet other semi-automated approaches which combine annotations with high-level instructions, such as Program Mandering [319].
- Fully automated methods such as μSCOPE [384] or ProgramCutter [463], which automatically partition software solely based on data dependencies (i.e., without input from developers).

Although approaches to automatically define compartmentalisation policies are plenty, few aim to explore the compartmentalisation design space. Instead, most aim to help developers materialise a policy that they have already devised (e.g., via annotations or high-level instructions) without consideration for performance.[5] A small number of influential approaches consider performance/security trade-offs. SOAAP [227] leverages a performance model to evaluate the cost of a given policy. Based on this evaluation, developers are invited to iterate on their approach. Program Mandering [319] and μSCOPE [384] leverage probabilistic security metrics and a performance model to improve the quality of policies they automatically determine. Both target fully automated exploration leveraging probabilistic security metrics, which have their limitations. Overall, all approaches rely on performance models, which too have their limitations. Lastly, all target either applications [227, 319] or OS kernels [384], but not both, making them rather inappropriate for full system specialisation.

[5]The tool would typically expect the developer to independently evaluate the resulting policy and adjust their annotations or high-level policy accordingly.

Motivation for RQ2: More work is needed in security/performance design-space exploration techniques: avoid overly complex performance and security models while staying usable; specialise the full system at once.

2.1.3.2 P2: How to *express* compartmentalisation in the program?

Having defined a policy, we now need to express it in the program. Developers perform this using programming abstractions, which take many different forms.

Example. Let us express the policy defined previously in our running example, using traditional UNIX abstractions: processes and pipes. At program startup, we would modify ImageMagick to fork a child to host the untrusted Ghostscript library. Once forked, the child process would drop its privileges to run under an unprivileged `nobody` user (see Figure 2.1b). ImageMagick and the Ghostscript child would then setup a pipe to communicate. At program runtime, when ImageMagick needs to use features of the isolated library, it would, instead of a function call, perform a Remote Procedure Call (RPC, step ④ in Figure 2.1b) through the pipe in a message-passing fashion, serialising and marshalling appropriate arguments. The Ghostscript library child would then communicate the return value(s) into the pipe when done.

This same policy could be represented in many other ways with other programming abstractions. For instance, we could use an in-process compartmentalisation framework such as Hodor [236]. In this case, we would not fork a new process, as both compartments would run in the same process. We would not require a pipe, as Hodor cross-compartment calls feature function-call semantics. Instead, we would replace library function calls with Hodor cross-compartment calls, and share data between ImageMagick and the Ghostscript library using compiler annotations.

Abstraction semantics. As depicted in the above example, programming abstractions strongly influence the nature of the engineering effort of compartmentalising a program. But beyond that, their semantics also strongly influence the trade-offs developers can achieve through compartmentalisation. Particularly relevant are the semantics of cross-compartment calls, and of cross-compartment data sharing:

- Cross-compartment calls (or "compartment switches"), for instance, can be synchronous (as is the case with a function call), or asynchronous [170, 57]. The

granularity of entry points can enable developers to define compartment entry points at, e.g., any arbitrary instruction [166, 434, 371], at function boundaries [320, 333], or only at exported symbols of linkage modules [114]. The semantics of failure too matter: does a call failure result in a special return value (as is the case with a function call), an interruption of the program, or in an automatic restart/retry [35, 346]? Is there a timeout for unresponsive compartments [35]?

- Data sharing can be realised through shared memory or message passing [125]. Here too, many factors influence the properties of programming abstractions. When sharing memory, is ownership temporally restricted [395]? Can objects be shared with specific rights, e.g., read, or write only [327]? When doing message passing, are objects serialised to enforce any form of type safety [248]?

When retrofitting compartmentalisation, the more semantics differ from that of function calls and of a shared memory model, the more the more engineering effort will be required to compartmentalise. This is why automated systems generally comply with the semantics of function calls [133, 171, 438, 213].

Here, two points are particularly relevant for this thesis. First, the semantics of compartmentalisation programming abstractions dictate how compartments interact and compose with the broader operating system. Nearly all compartmentalisation abstractions target either the kernel or the application, but rarely both. This is restrictive: a more flexible user/kernel boundary, e.g., allowing to merge certain user and kernel compartments, could enable for drastic performance optimisations [167], particularly when considering single-purpose appliances [323, 285]. When targeting applications, abstractions behave orthogonally to processes, which leads to composition problems [173] and inefficiencies. Second, the semantics of compartmentalisation programming abstractions impact which mechanisms can be used to enforce the abstraction under the hood (as we discuss in the next subsection). Most abstractions are designed with a particular mechanism in mind, hindering the ability to leverage new mechanisms, which are released at a fast pace; for instance, the process abstraction and `fork()` semantics makes it hard to replace the page table [134].

Motivation for RQ1: More work is needed towards *flexible* compartmentalisation abstractions: among others, suitable for "holistic" separation at the scale of the entire operating system (as in: OS kernel + application), as well as mapping to a wide range of enforcement mechanisms.

2.1.3.3 P3: How to enforce compartmentalisation at runtime?

Lastly, we need to enforce the policy, defined in **P1** and programmatically expressed in **P2**, at runtime. Here too there are many different approaches which can rely on hardware and/or software.

Example. In §2.1.3.2, we implemented the policy with processes and pipes. At runtime, in a commodity OS such as Linux, the compartmentalisation would be enforced by the Memory Management Unit (MMU), and compartment switches would then be performed by switching the page table. Note that this same abstraction (processes and pipes) could be enforced with other mechanisms, such as hardware capabilities [454]. Had we implemented the policy with another programming abstraction, e.g., an intra-process compartmentalisation abstraction such as Hodor [236], the policy would be enforced using different technique, e.g., memory protection keys, which we discuss below.

Numerous classes of technologies can be used for enforcing software compartmentalisation. In hardware, they include physical separation [387], page-table-based approaches [44, 45, 296, 297] and protection keys [41, 46, 5, 400, 469], segmentation [208, 341], hardware capabilities [154, 440, 454, 343], bounds-checking [186, 89, 396, 273, 281], tagged-memory-based architectures [31, 86, 412, 455, 246, 386, 257, 188], and others. In software, they include software capabilities [237, 158], software fault isolation [445, 155, 467, 472, 342, 229, 260, 259], memory encryption [281], and others. Providing a taxonomy or full discussion of the different classes is out of scope for this background section (we do so in Chapter 5). Instead, we will provide background on a few technologies relevant in this thesis:

- *Protection Keys for Userspace [46] (PKU)* is a protection key mechanism present in Intel and AMD CPUs offering low-overhead intra-address-space memory isolation.[6] PKU leverages unused bits in the page table entries to store a *memory protection key*, enabling up to 16 protection domains. The PKRU register then stores the protection key permissions for the current thread. On each memory access, the MMU compares the key of the target page with the PKRU and triggers a page-fault in case of insufficient permissions. Other mechanisms in the protection key family [41, 46, 5, 400, 469] behave similarly. We use PKU in Chapter 3.

[6] Intel PKU is often referred to as Intel Memory Protection Keys (MPK), a family of mechanisms which also includes Intel Protection Keys for Supervisor (PKS), a mechanism similar to PKU for use in Ring 0.

- *Extended Page Table (EPT)* is a hardware-assisted virtualisation mechanism present in Intel CPUs (with similar implementations available from other manufacturers). In this paper, we refer to EPT as a synonym for Virtual Machine (VM) isolation. Unlike Intel PKU domains, VMs feature entirely distinct address spaces, making EPT a heavier and more costly mechanism. We use EPT in Chapter 3.

- *CHERI hardware capabilities* are a hardware extension to the MIPS, RISC-V, and ARMv8 ISAs. CHERI extends architectures with hardware capabilities, an architectural data type akin to fat pointers [288] extending conventional pointers with bounds and permission information. Capabilities are made unforgeable by tagged memory. CHERI capabilities can be used as a compartmentalisation technology, featuring a performance overhead ranging between that of protection keys and EPT. We sketch the use of CHERI in Chapter 3 (which was published before Arm Morello, the first SoC prototype of CHERI, was made available [120]) and implement this sketch in Kressel et al. [283].

2.1.3.4 P1-3: Wrapping up

Throughout this section we provided background on the entire compartmentalisation stack, from defining a policy (**P1**), expressing it in programs (**P2**), to enforcing it at runtime (**P3**). From this overview, we make two key observations.

Systematisation. **P1-3** span many aspects of computer science: security, operating systems, programming languages, hardware and architecture, software engineering (and, to a lesser extent, embedded systems and formal methods). For this reason, researchers rarely have an overview of all of **P1-3**. In fact, solutions to each part of the stack are often contributed by researchers from different communities with different backgrounds.[7] Unfortunately, "divide and conquer" shows its limits. It is perilous to approach either end of the stack in isolation, at the risk of being off target: e.g., designing policies (or tools to generate policies) which cannot be represented efficiently with existing abstractions, abstractions which do not map to enforcement methods, or hardware which does not efficiently enforce typical partitioning needs.

Despite of this, little work has been dedicated to systematising software compartmentalisation knowledge. In fact, even the classification of **P1-3** is a contribution of

[7] **P1** for instance, is a software engineering, security, and programming languages (PL) problem. **P2** is a traditional problem in operating systems and security. **P3** is a security and PL problem (when approached in software), or security and hardware/architecture (when approached in hardware).

this thesis presented in Chapter 5. Works closest to a systematisation of software compartmentalisation are Shu et al. [409], which surveys general isolation, and Acar et al. [110] which systematises general Android security research, including application compartmentalisation. Both feature a significantly more general scope than software compartmentalisation, and thus do not fulfil the need for systematisation we highlight. Both also pre-date many recent works in compartmentalisation, being from 2016.

> **Motivation for RQ5:** More work is needed in systematising software compartmentalisation works across the stack, to highlight the gaps and challenges of each of **P1**/**P2**/**P3**, and how efforts across the stack should coordinate to address them.

Research vs. practice. Though not a mainstream practice, compartmentalisation has seen deployment outside research since the late 1990s. Some of these efforts have been extensively covered in research, e.g., browser site isolation [163, 447, 378], browser library isolation [344], or OpenSSH separation [373]. However, outside these few cases, most efforts outside academia and major open-source projects have not been widely discussed in the research world.[8] For instance, vast efforts over the last two decades have been deployed into compartmentalising the OpenBSD userland [65], much of which has never been mentioned in research. This raises questions, such as who is compartmentalising outside research, what are the characteristics of these compartmentalisation efforts, and what discrepancies or gaps does this highlight with existing research trends?

> **Motivation for RQ6:** Systematisation efforts should extend to existing compartmentalised programs outside research, to understand their characteristics and what should be learned from them.

2.2 Interface Safety and Compartmentalisation

In this second part, we give background on the problem of interface vulnerabilities in compartmentalisation, illustrating using our running example from the previous section. We motivate for more work studying these vulnerabilities, as done in Chapter 4.

[8] There a likely multiple reasons for this. Compartmentalisation efforts outside research and major open-source projects are not widely publicised, as they do not lead to research publications or news outlets. When led at a small scale by isolated communities, they may go entirely unnoticed.

```c
/* ImageMagick callback exposed to the Ghostscript library.
 * This function is called by the Ghostscript library to transmit
 * outputs emitted by the PostScript document to ImageMagick. */
static int MagickDLLCall GhostscriptDelegateMessage(
  void *handle, const char *message, int length) {

  /* Interface vulnerability: unchecked usage of
   * sandbox-provided pointer/bounds information */
  (void) memcpy(*handle, message, (size_t) length);
  (*handle)[length] ='\0';
} /* ... abbreviated / simplified ... */
```

Listing 1: Example of real-world interface vulnerability: ImageMagick callback lets the Ghostscript library perform arbitrary writes outside its sandbox. This example, taken from Chapter 4 (cf. Listing 3), was found with ConfFuzz.

2.2.1 What are Interface Vulnerabilities?

A central problem of software compartmentalisation is validating all control and data flows at compartment boundaries [373]; we refer to this as ensuring *interface safety*. Improper validation results in a wide range of *interface vulnerabilities*, which allow attackers to escalate privileges by compromising other compartments. Simply isolating software components is not enough: component interfaces must be hardened into solid trust boundaries to achieve strong security properties.

Example. Let us illustrate this using our running example from §2.1.1. Following the example's setting, we expect that attackers, having exploited a vulnerability in Ghostscript to take over the compartment, will remain confined within the sandbox. Yet, as shown in Listing 1, the interface between the two compartments is vulnerable: ImageMagick exposes a callback to the untrusted Ghostscript library, which allows it to perform arbitrary write operations in ImageMagick, as often as it wants, and at any time, negating isolation properties entirely. This interface vulnerability is caused by ImageMagick (victim compartment) dereferencing sandbox-provided pointers (`handle` and `message`) and bounds information (`length`) without sufficient checking (in fact, here, without *any* checking), a typical consequence of retrofitting compartmentalisation without sufficient attention to compartment interfaces. This vulnerability, taken from Chapter 4, is a real-world flaw found by ConfFuzz, our interface vulnerability fuzzer, in the Cali compartmentalisation framework [133].

Interface vulnerabilities are a known problem relevant to many different fields. Generally, interface vulnerabilities are instances of confused deputy problems [234] (i.e., situations where a privileged component is tricked by an unprivileged component into misusing its privileges). This term is used in the literature [436]. In software compartmentalisation, interface vulnerabilities have also been referred to as "misuse of interfaces" [397], "improper use of legitimate interfaces" [201], "in-process abuse" [166], or "retrofitting pitfalls" [344]. A subset of compartment interface vulnerabilities was referred to as "Dereferences Under Influence" [244] (DUIs). In the domain of trusted execution environments, where interfaces vulnerabilities are a popular problem, they have been called "input validation errors" [157] and "COIN attacks" [270]. Their instantiation at the system-call API was referred to as "Iago" [161].[9]

This profusion of names is characteristic of a reality: although the problem of confused deputies in compartmentalised software is long known, it has not been directly studied. Of the above, the closest to such a study is Narayan et al. [344], which documents a list of interface vulnerabilities found while compartmentalising software. Although such efforts are valuable, a more systematic and larger-scale study is needed to better understand the problem and its implications on the security of compartmentalised software. Lastly, although more or less complete classifications have been proposed in the field of trusted execution environments [161, 436, 270, 157], these do not capture the specificities of interface vulnerabilities in compartmentalised software.

> **Motivation for RQ3:** More work is needed to better understand interface vulnerabilities: which classes of interface vulnerabilities are possible, and what are their impact(s)?

2.2.2 Interface Vulnerabilities are a Growing Concern

Although interface vulnerabilities have existed for as long as time-sharing computers, their importance has significantly increased in the past decade. As described throughout this chapter, new requirements in software security and new advances in enforcement technologies make it increasingly common to *retrofit* compartmentalisation into legacy software with minimal code changes, automatically, and at an increasingly fine grain. This comes in contrast with the historical approach of designing software as partitioned in the first place (or retrofitting with heavy changes), manually, and at a coarse

[9] Interested readers may refer to the glossary of Section 5.8 for a detailed discussion of these terms, including a more formal definition of confused deputy problem.

granularity. These new practices make compartmentalised software increasingly vulnerable to interface vulnerabilities:

- Retrofitting implies applying a distrust relationship at boundaries that were not initially thought for this threat model. Intuitively, this is more likely to cause interface vulnerabilities than designing boundaries for distrust in the first place.
- Automatically compartmentalising software will result in interface vulnerabilities: unless software is formally specified, compartmentalisation frameworks do not fully understand the semantics of programs and of their interfaces, and thus cannot guarantee the absence of interfaces vulnerabilities [397].
- Finer granularities of compartmentalisation can be seen as beneficial for security because they make it possible to more closely approach true least privilege [384]. On the other hand, by multiplying the number of compartments, finer granularities also multiply the number of interfaces, increasing the difficulty to ensure that these interfaces are safe.

This argumentation relies strongly on intuition. This is because the problem of interface safety in software compartmentalisation has not been quantified: where efforts have been made in quantifying the *benefits* of compartmentalising (i.e., the attacks that compartmentalisation can thwart), often when evaluating methods [319, 384, 333, 269], or even in self-standing studies [139], no such effort was made towards quantifying the new attacks compartmentalised software architectures open for. For instance, to what extent does retrofitting compartmentalisation induce more interface vulnerabilities than compartmentalising new programs? To what extent do finer granularities make interface protection efforts more complex? Perhaps as a result, these observations, despite their relative obviousness, are not widespread in the literature.

> **Motivation for RQ3:** More work is needed to quantify the problem of interface safety: what is the prevalence of interface vulnerabilities when retrofitting compartmentalisation? Are automated approaches likely to be more vulnerable than manual approaches? To what extent do finer granularities really exacerbate the problem of interface safety?

2.2.3 Thwarting and Detecting Interface Vulnerabilities

There have been relatively few attempts at designing tools and methods to find and address interface vulnerabilities in compartmentalised software. Overall, most works

throughout the stack of **P1**/**P2**/**P3** consider these vulnerabilities orthogonal in their threat model (or simply ignore the problem altogether), as developers were historically responsible to manually secure compartment interfaces when compartmentalising software. Yet, as we discussed, this model is shifting. Next, we will give an overview of the main approaches to protecting against interface vulnerabilities, and finding interface vulnerabilities in compartmentalised software.

2.2.3.1 Protecting Against Interface Vulnerabilities

Defences against interface vulnerabilities can be designed at each step of the compartmentalisation process (**P1**/**P2**/**P3**):

- At **P1**, compartment boundaries can be chosen to eliminate certain types of interface vulnerabilities, minimise the number of interface vulnerabilities, or maximise the ease to harden the interface. Most non-manual approaches focus on using data-flow analysis to vet that private data does not leak out of compartments. Safeboxing methods such as Glamdring [313], Shreds [319], or SOAAP [227] leverage annotated secrets and data-flow analysis to that aim. Beyond data-flow analysis to protect secrets, defences at **P1** have not been widely explored.

- At **P2**, programming abstractions can be designed to eliminate interface vulnerabilities by construction, e.g., forbidding certain constructs which are known to result in interface vulnerabilities. **P2** is where most defence approaches focus. Here, a number of approaches have been explored: enforcing invariants on interface definitions, e.g., restricting interface-crossing types [327, 344] or enforcing points-to ranges in interface-crossing pointers [327, 313, 344, 270]; forcing developers to check interface-crossing data [344], or checking automatically when possible [327, 313, 344]; or enforcing restrictions on cross-compartment control-flow, providing primitives to specify and enforce API call ordering [344].

- At **P3**, mechanisms can be designed to enforce properties that make certain interface vulnerabilities fundamentally impossible. Hardware capabilities such as CHERI [454] enforce spatial memory safety, which limits the possible impact of interface vulnerabilities. Pointer authentication [312] and bounds-checking techniques [186, 89, 396, 273, 281] can also provide properties which limit the impact of some classes interface vulnerabilities. Much like **P1**, defences at **P3** have not been widely explored. Although this topic is discussed in Chapters 4 and 5, designing new mechanisms is not a focus of this thesis.

Limitations. Overall, hardening interfaces against interface vulnerabilities remains a widely manual effort, and there remains much room for contribution in defences throughout the stack. In particular, defences at **P1** have been vastly under-explored, whether automatically or manually. Where **P1**-level defences vastly focus on leveraging data-flow analysis to isolate secrets [313, 319, 227], nearly none attempts to choose interfaces as to minimise the number of interface vulnerabilities, or maximise the ease to harden them programmatically. We argue that this is because the state of the art does not offer a systematic way to compare interfaces: e.g., provided two interfaces I_a and I_b, which of them constitutes a more appropriate separation boundary? Some heuristics, such as the quantity of shared data exposed [463, 319, 384], have been proposed, but they are based on anecdotal evidence and have not been confronted to the real world.

> **Motivation for RQ4:** This motivates for more work towards identifying the software design patterns which characterise stronger or weaker boundaries, along with the complexity of refactoring them.

2.2.3.2 Finding Interface Vulnerabilities

Approaches to find interface vulnerabilities in compartmentalised software supplement defences by finding those vulnerabilities which are not mitigated by deployed defences. Detection methods can be classified in two broad classes; *complete*, and *incomplete*:

- Complete (or *systematic*) methods can find all interface vulnerabilities of a certain type. They are typically based on static analysis. Complete methods tend to be computationally expensive or complex, leading to scalability issues with large codebases. Due to the difficulties of pointer alias analysis in languages such as C or C++, complete methods are also usually affected by false positives. DUI Detector [244] is a complete method which find all spatial interface vulnerabilities using a combination of static analysis and symbolic execution.
- Incomplete methods do not guarantee that they will find all vulnerabilities of a given type, at any point of their execution. On the other hand, because they can resort to dynamic analysis, they can often yield more detailed information and less false positives (or none at all). Emilia [179] is a fuzzer specialised to find Iago [161] vulnerabilities, a type of interface vulnerabilities in the context of trusted execution environments. Nyx-Net [401] is a classical network fuzzer which has been applied to fuzzing for interface vulnerabilities at the Firefox Inter-Process Communication (IPC) layer.

Overall, there are very few approaches of complete or incomplete type. Though methods employed in other fields (static analysis, symbolic execution, fuzzing) can be applied to detecting interface vulnerabilities, few works attempted to do so. Fuzzing, particularly, has been vastly under-explored in the context of compartmentalisation and interface vulnerabilities, and should be a promising technique to combine with existing defences. We explore this direction in Chapter 4.

2.2.3.3 Evaluating Defences and Detection Methods

As we observed earlier in Section 2.2.1, there is a lack of overview on the different classes of vulnerabilities that can arise at compartment interfaces, resulting in a lack of awareness on the problem of interface safety and its impact. But beyond awareness, this lack of overview makes it difficult to evaluate defences and vulnerability detection methods. Though none of the existing approaches discussed in Sections 2.2.3.1 and 2.2.3.2 cover the complete spectrum of interface vulnerabilities, it is difficult to evaluate *how incomplete* they are, and to even define how a complete protection or detection method should look. Overall this provides further motivation towards systematising and studying the problem of interface vulnerabilities: this thesis will not be about solving the problem, but rather about better understanding it to address it more completely and efficiently in the future.

> **Motivation for RQ3:** To be able to evaluate defences and detection methods, we need more work to better understand the problem of interface vulnerabilities, and particularly which classes of flaws can arise.

2.3 Summary

In this chapter, we provided background on the concept of software compartmentalisation, as well as on the problem of interface safety. In doing so, we motivated the contributions which we will present in the following chapters.

In a first part, we motivated for more specialised, holistic approaches to operating systems for software compartmentalisation. Overall, existing approaches rely on overly rigid programming abstractions, which make it difficult to partition with the whole system in mind, or to leverage a variety of enforcement mechanisms, limiting the trade-offs which can be achieved (**RQ1**). Building on this, we motivated for more work towards exploring the compartmentalisation design space automatically. Overall, few approaches

explore the performance/security trade-offs of software compartmentalisation. When they do so, they employ probabilistic performance and security models which are not always appropriate in practice. Making progress in this direction is all the more necessary as we are increasing the size of the design space (**RQ2**). We will explore these questions in Chapter 3.

In a second part, we motivated for a better characterisation of interface vulnerabilities in software compartmentalisation. Overall, although the relevance of interface vulnerabilities is increasing with recent advances in compartmentalisation, these flaws remain insufficiently studied. To show how important the problem is, be able to better evaluate past and future defences against interface vulnerabilities, and build more complete ones, we need to build a better and more systematic understanding of the problem (**RQ3**). Similarly, to be able to build better defences at each layer of the **P1**/**P2**/**P3** stack, we need to better understand the patterns which lead to interface vulnerabilities, and how to leverage them (**RQ4**). We will explore these questions in Chapter 4.

Over the course of the chapter, we also motivated for a large-scale systematisation of knowledge in the field of software compartmentalisation. Overall, little work has been dedicated to systematising software compartmentalisation knowledge. We need to better understand the approaches to software compartmentalisation works across the stack, to highlight the gaps and challenges of each of **P1**/**P2**/**P3**, and to determine how efforts across the stack should coordinate to address them (**RQ5**). This systematisation effort should be pushed beyond research to existing industry and practice works in compartmentalisation, to identify discrepancies, and gaps, with existing research trends (**RQ6**). We will explore these questions in Chapter 5.

Chapter 3

FlexOS: Towards Flexible OS Isolation

> "Which escape do you want to learn?
> There's the Art of the Big Dipper, which involves
> thirty-six transformations, or the Art of the
> Earthly Multitude, which involves seventy-two."
> "The more the merrier," considered Monkey.
>
> *Journey to the West*
> Wu Cheng'en

This chapter presents FlexOS, a modular operating system which enables users to implement highly-specialised compartmentalisation policies, and a semi-automated design-space exploration method based on partially-ordered sets. By exploring holistic and specialised approaches to compartmentalisation, we aim to show that a vast performance/security design space can be unlocked. In the bigger picture of this thesis, better tailoring compartmentalisation approaches to use-cases is an essential step towards faster and safer compartmentalisation. These contributions address research questions **RQ1** and **RQ2**, and are derived from Lefeuvre et al. 2022 [299].

Contributions of the Author

I designed and implemented FlexOS and the FlexOS Intel PKU backend. I performed most of the evaluation and analysis. I wrote most of the paper. Vlad-Andrei Bădoiu

ported the GCC ASan, UBSan, and Stack Protector features to work on FlexOS (*hardening*). Alexander Jung helped me run the 160 FlexOS configurations in Figure 3.6 using Wayfinder [261]. Stefan Lucian Teodorescu and later Sebastian Rauch designed and implemented the FlexOS EPT backend, under my supervision. Felipe Huici, Costin Raiciu, and Pierre Olivier provided feedback and contributed key edits throughout the paper.

Abstract

At design time, modern operating systems are locked in a specific safety and isolation strategy that mixes one or more hardware/software protection mechanisms (e.g., user/kernel separation); revisiting these choices after deployment requires a major refactoring effort. This rigid approach shows its limits given the wide variety of modern applications' safety/performance requirements, when new hardware isolation mechanisms are rolled out, or when existing ones break.

We present FlexOS, a novel OS allowing users to easily specialize the safety and isolation strategy of an OS at compilation/deployment time instead of design time. This modular LibOS is composed of fine-grained components that can be isolated via a range of hardware protection mechanisms with various data sharing strategies and additional software hardening. The OS ships with an exploration technique helping the user navigate the vast safety/performance design space it unlocks. We implement a prototype of the system and demonstrate, for several applications (Redis/Nginx/SQLite), FlexOS' vast configuration space as well as the efficiency of the exploration technique: we evaluate 80 FlexOS configurations for Redis and show how that space can be probabilistically subset to the 5 safest ones under a given performance budget. We also show that, under equivalent configurations, FlexOS performs similarly or better than existing solutions which use fixed safety configurations.

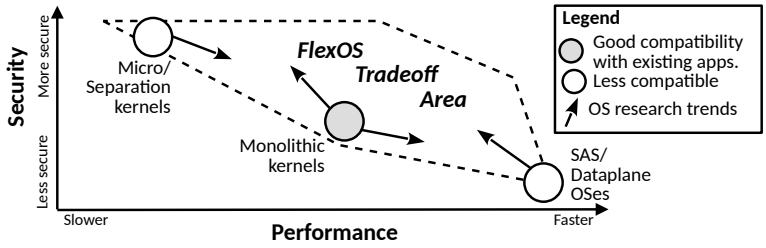


Figure 3.1: Design space of OS kernels.

3.1 Introduction

Modern OS architectures are heavily interlinked with the protection mechanisms they rely upon. OSes rigidly commit at design time to various high-level safety decisions, such as the use of software verification, hardware isolation, runtime checking, etc. Changing these after deployment is rare and costly.

The current OS design landscape (depicted in Figure 3.1) broadly consists of microkernels [238, 278], which favor hardware protection and verification over performance, monolithic kernels [144], which choose privilege separation and multiple address spaces (ASes) to isolate applications, but assume all kernel code is trusted, and single-address-space OSes (SASOSes), which attempt to bring isolation within the address space [158, 305, 237], or ditch all protection for maximum performance [323, 352, 285]. Making post-design changes to these high-level safety decisions is very difficult to implement. For instance, removing the user/kernel separation [324] requires a lot of engineering effort, as does breaking down a process into multiple address spaces for isolation [272]. Recently, the potential safety benefits hinted by the proposal to introduce Rust components in Linux [197] are questioned by the fact that the bulk of the kernel code will remain written in a memory-unsafe language [196].

The rigid use of safety primitives in modern OSes poses a number of problems. First, it precludes per-application OS specialization [200, 262, 325, 329] at a time when modern applications exhibit a wide range of safety and performance requirements. Prematurely locking the design into any combination of safety primitives is likely to result in suboptimal performance/safety in many scenarios. Effortless specialization for safety is further motivated by the fact that today's applications are made up of multiple components showing different degrees of trust and criticality, and as such requiring various levels of isolation. Furthermore, new isolation mechanisms [174, 400, 454, 177, 121, 120], with the ability to complement or replace traditional ones, are regularly being proposed by CPU manufacturers. When multiple mechanisms can be used for the same

task, choosing the most suitable primitive depends on many factors, and should ideally be postponed to deployment time. Finally, when the protection offered by a hardware primitive breaks down (e.g., Meltdown [314]), it is difficult to decide how it should be replaced, and generally costly to do so.

This leads us to the following research problem: *how can we enable users to easily and safely switch between different isolation and protection primitives at deployment time, avoiding the lock-in that characterizes the status-quo?*

Our answer is *FlexOS*, a modular OS design whose compartmentalization and protection profile can easily and cost-efficiently be tailored towards one application or use-case at build time, as opposed to design time. To that aim, we extend the Library OS (LibOS) model and augment its capacity to be specialized towards a given use case, historically done for performance reasons [200, 262, 325, 329], towards the *safety* dimension.

With FlexOS, the user can decide, at *build time*, which of the fine-grained OS components should be placed in which compartment (e.g., the scheduler, TCP/IP stack, etc.), how to instantiate isolation and protection primitives for each compartment, what data sharing strategies to use for communication between compartments, as well as what software hardening mechanisms should be applied on which compartments. To that aim, we abstract the common operations required when compartmentalizing arbitrary software behind a generic API that is used to retrofit an existing LibOS into FlexOS. This API limits the manual porting effort of kernel and application legacy components to the marking of shared data using annotations. These annotations, alongside other abstract source-level constructs, are replaced at build time by a code transformation step that instantiate a given FlexOS safety configuration.

The design space enabled by the system, illustrated on Figure 3.1, is very large and difficult for non-expert users to explore manually. This leads to our second research question: *how to guide the user navigating the vast design space unlocked by FlexOS?* To answer this, we propose a semi-automated exploration technique named *partial safety ordering*, using partially ordered sets to describe the probabilistic security degrees of FlexOS' configurations and identify the safest ones under a given performance budget.

We have implemented a prototype of FlexOS with support for Intel Memory Protection Keys (MPK)[1] and VM-based isolation[2], as well as a wide range of hardening mechanisms (CFI [109], ASan [403, 425], etc.). Our evaluation on four popular applications demonstrates the wide safety versus performance tradeoff space unlocked by

[1] And specifically its implementation for userspace, Intel Protection Keys for Userspace (PKU), as Intel Protection Keys for Supervisor (PKS) was not available at the time of this writing.

[2] Here also referred to as Extended Page Table (EPT) isolation.

FlexOS: we evaluate over 160 configurations for Redis and Nginx. We also show the ease of exploring different points in that space: our semi-automated exploration technique can probabilistically subset the 80 Redis configurations to the 5 safest ones under a given performance budget. Finally, we demonstrate that under equivalent configurations, FlexOS performs better or similarly to baselines and competitors: a monolithic kernel, a SASOS, a microkernel, and a compartmentalized LibOS.

3.2 Flexible OS Isolation: Principles, Challenges

FlexOS seeks to enable users to easily and safely switch between different isolation and protection primitives at deployment time. This section formalizes the fundamental design principles required to achieve this, the challenges that arise from them, and how we address them.

3.2.1 Principles

P1) ***The isolation granularity of FlexOS' components should be configurable.*** The compartmentalization strategy, i.e., the number of compartments and which components are merged/split into compartments, has a major impact on safety and performance, thus it should be configurable.

P2) ***The hardware isolation mechanisms used should be configurable.*** There is a wide range of isolation mechanisms with various safety and performance implications. These should be configurable by the user. For the OS developer, supporting a new mechanism should not involve any rewrite/redesign and be as simple as implementing a well-defined API.

P3) ***Software hardening and isolation mechanisms should be configurable.*** Software hardening techniques such as CFI, or Software Fault Isolation (SFI), as well as memory safe languages such as Rust, bring different levels safety at a variable performance cost. They should be selectively applicable on the components they are the most meaningful for in a given use case.

P4) ***Flexibility should not come at the cost of performance.*** The OS runtime performance should be similar to what would be achieved with any particular safety configuration without the flexibility approach.

P5) ***Compatibility with existing software should not come at a high porting cost***, to maximize adoption.

P6) ***The user should be guided in the vast design space enabled by FlexOS.*** Given its very large configuration space, the system should come with tools helping the user identify suitable safety/performance configurations for a given use case.

3.2.2 Challenges and Approach

P1 and P4 raise the question of ***how to offer variable isolation granularities, and how to do so without compromising performance?*** Genericity is typically paid at the price of performance [329, 328, 285], and interface design may not be easily decoupled from the isolation granularity without performance loss [211]. In order to tackle this issue, we propose to rely on a LibOS design that is *already finely modularized while providing state of the art performance*, Unikraft [285]. The main idea is to consider Unikraft's level of modularization (micro-library) as a minimal granularity, using pre-existing interfaces as compartment boundaries. Then, in order to maximize performance and safety for a given use case, less granular configurations can be composed by merging select components into compartments. At build time when an isolation mechanism is selected, FlexOS uses code transformations to inline function-call-like cross-domain gates, avoiding the overhead of a runtime abstraction interface [206].

P2 and P5 bring the challenge of ***how to design an OS in which 1) isolation can be enforced by many hardware mechanisms and 2) the engineering cost of introducing a new mechanism is low?*** Technology agnosticism is already difficult in userland software, but core kernel facilities (interrupt handling, memory management, scheduling) introduce additional complexity that should be handled very differently depending on the underlying isolation technology. For example, some technologies share a single address space between protection domains (e.g., MPK [174]) while other use disjoint address spaces (e.g., ARM TrustZone [121], EPT). The main idea of FlexOS is to abstract existing isolation technologies and identify kernel facilities that require different handling depending on the technology, and design these subsystems so as to minimize the changes needed when implementing a new technology.

P5 asks ***how to limit the engineering costs of porting new applications/libraries?*** To allow compatibility with existing software, FlexOS extends an OS that offers a POSIX interface. That OS is compartmentalized by marking cross-component calls and shared data using an abstract API and, in its basic form, porting a new application requires the developer to use the same API to mark shared data (i.e., data passed to other components) with source-level annotations. This avoids the need to change the application design or major code rewriting. Such an approach is common among

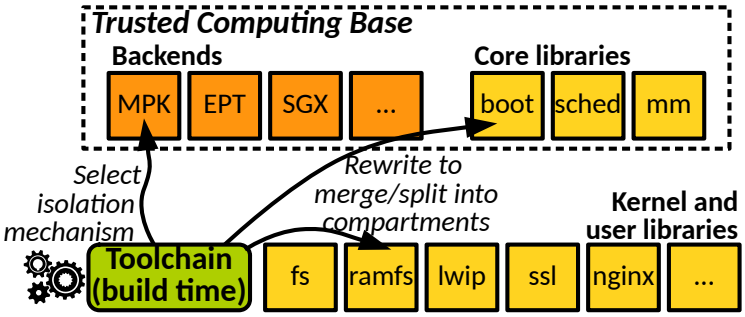


Figure 3.2: OS overview. The trusted computing base includes backends and core libraries. Backends are used by the toolchain to rewrite the libraries at build time.

state-of-the art compartmentalization frameworks [434, 236, 344, 400].

Finally, P1-P3 and P6 raise the question of ***how to help the user navigate the vast design space enabled by FlexOS?*** The introduction of safety flexibility increases the potential for safety/performance specialization, but selecting suitable configurations may be hard for a non-expert. For example, it can be difficult to reason about the safety implications of increasing the degree of compartmentalization vs. increasing the level of software hardening for a given configuration. To tackle that issue, we propose a method named *partial safety ordering*, using partial order relationships to probabilistically rank FlexOS configurations by safety and identify the safest ones for a given application under a performance budget.

Section 3.3 presents an OS design that satisfies P1-P5, and Section 3.4 gives key implementation points of a prototype we developed. Section 3.5 shows an approach to tackle P6. Finally, Section 3.6 presents an evaluation of our prototype.

3.3 Designing an OS with Flexible Isolation

We now provide an overview of FlexOS' main elements, going through its design, compartmentalization API, backend API, and finally its trusted computing base.

FlexOS is based on a modular LibOS, Unikraft [285] composed of a set of independent, fine grained libraries. In FlexOS, each library can be placed in a given compartment (an isolation domain), and it can be hardened via techniques such as Control-Flow Integrity (CFI), address sanitization and so forth. This safety configuration is provided at build time, in a configuration file provided by the developer, and FlexOS' toolchain produces an OS image with the desired safety characteristics. Listing 2 presents an example of such a configuration file that isolates libopenjpeg and lwIP in a separate

```yaml
compartments:
  - comp1:
    mechanism: intel-mpk
    default: True
  - comp2:
    mechanism: intel-mpk
    hardening: [cfi, asan]
libraries:
  - libredis: comp1
  - libopenjpeg: comp2
  - lwip: comp2
```

Listing 2: Example of a FlexOS configuration file that isolates libopenjpeg and lwIP in a separate compartment, with CFI and ASan enabled.

compartment with CFI and ASan enabled.

In contrast to Unikraft where all libraries are in the same protection domain and any library can directly call a function from another library, in FlexOS' source code libraries call external functions via *abstract gates*, and may share data with external libraries at the granularity of a byte using abstract code annotations. Gates and annotations form an API used to compartmentalize Unikraft into FlexOS, and represent metadata which is automatically replaced by our toolchain with a particular implementation at build time. Different implementations can leverage different isolation technologies, or flavors of a same technology. We refer to the API implementation for a given technology (MPK, EPT, etc.) together with its runtime library as *isolation backend*. This subsection gives a short overview of FlexOS' main design elements, which are then elaborated in the following subsections. Figure 3.2 depicts the components described in this subsection.

LibOS Basis. Achieving flexible isolation at a fine granularity implies a high degree of modularity. In practice, this modularity is not offered by typical monolithic general-purpose OSes [285]. A flexible isolation approach on the basis of Linux would require a first non-trivial "modularization" step [308] that may take years of engineering and careful redesign. Library OSes [285] and component-based OSes [148, 360] are a better starting point for flexible OS isolation because they often provide highly modular codebases with good application compatibility and high performance. Flexible isolation also suits well the specialization spirit of LibOSes, where the OS can be tailored for a given application/use-case. This was historically done for performance [200], and FlexOS enables specialization towards safety.

API and Build-time Instantiation. Unlike a typical LibOS, we design FlexOS in an *isolation-agnostic* manner. Cross compartment calls are made through abstract call gates that are instantiated at build time (arrows in Figure 3.2). Shared data is marked using compiler annotations, used at build time to instantiate a given data sharing strategy. Unlike linker-based approaches [395], FlexOS performs replacements using source to source transformations using Coccinelle [356, 294]. This has the advantage of allowing all compiler optimizations and gives FlexOS a clear performance advantage compared to historical approaches that relied on heavyweight runtime abstraction interfaces such as `COM` for Flux OSKit [206]. It also makes FlexOS' isolation approach easy to debug and understand by anyone who knows C: transformations can be visually inspected in a high-level language with usual file comparison tools.

3.3.1 Compartmentalization API and Transformations

Most isolation mechanisms (memory protection keys [174], trusted execution environments such as Intel SGX [177], or hardware capabilities [454]) restrict data access according to a set of current privileges, and provide a means to switch privileges and share data across compartments. Ensuring safety is equivalent to controlling privilege transitions, making sure that the system only ever enters "legal" couplings of executing code and data privileges. Other isolation approaches such as ARM TrustZone [121] or EPT/VMs consider compartments as entirely different systems (or "worlds"), enforcing a 1:1 system/compartment mapping. With this approach, systems never switch privileges, instead they communicate with other compartments via Remote Procedure Calls (RPCs) and shared memory. We design FlexOS' call gates and data sharing primitives to cater for both approaches. In FlexOS, the only requirement for an isolation mechanism is to (1) implement the concept of protection domains and provide a domain switching mechanism, and (2) support some form of shared memory for cross-domains communication. To the best of our knowledge, this applies to the vast majority of industry and research isolation mechanisms. This subsection gives an overview of FlexOS' compartmentalization approach, first focusing on the API with call gates and shared data, and then on build-time source transformations.

Call Gates. In FlexOS, cross-library calls are represented in the source code by *abstract call gates*. At build time, as part of the transformation phase, abstract call gates are replaced with a specific implementation. For instance, when the caller and callee

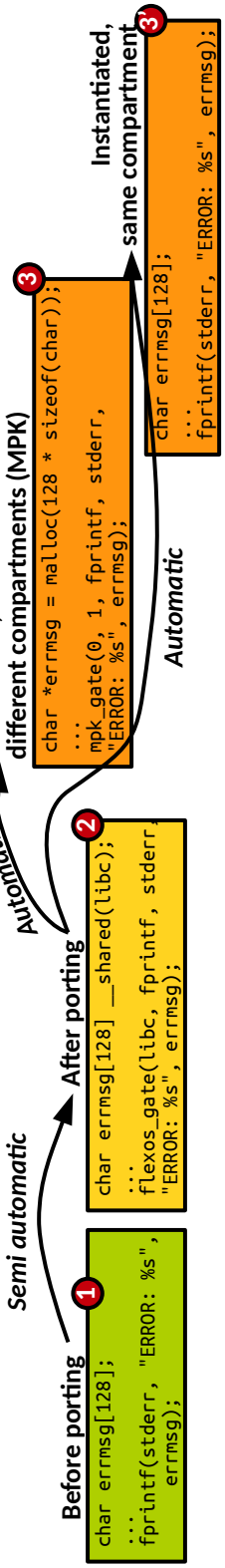


Figure 3.3: FlexOS code transformations. First, developers manually annotate shared data, and gate placeholders are semi-automatically inserted (direct calls: automatic, indirect calls: manual). At build time, API primitives are automatically replaced with the chosen mechanism. In the MPK case, shared data can for example be allocated on a shared heap. If the two libraries are in the same compartment, the result is similar to the code prior porting, resulting in zero overhead.

are configured to be in the same compartment, call gates implement a classical function call. When they are in different compartments, isolated for example by MPK, the call gate performs a protection domain switch before finally executing the `call` instruction. In a setting where libraries are isolated using VMs, the call gate performs a Remote Procedure Call (RPC). From the perspective of the compiler, caller, and the callee, call gates are entirely transparent as they implement the System V ABI calling convention. Unlike typical System V function calls however, call gates guarantee isolation of the register set and therefore save and zero out all registers not used by parameters. Figure 3.3 presents an example of gates from the porting (step ②) to the replacement by the toolchain (③ and ③').

The part of the process of porting existing user/kernel code to FlexOS consisting in marking call gates is automated: knowing the control-flow graph of the system, static analysis determines whether a procedure call crosses library boundaries, and if so, performs a syntactic replacement of the function call with a call gate instead. A corner case requiring programming effort is when a component calls another component through a function pointer. The callee cannot be determined statically, thus the programmer must annotate the possible pointed functions with the list of possible components they can be called from. The toolchain will then generate wrappers enclosing the implementations of the functions in question in the appropriate call gates. Our prototype implementation uses Cscope [8] and Coccinelle [356].

FlexOS call gates are not trampolines. Instead, they replace System V function calls entirely and are always inlined at the call site. An advantage of such approach is that call gates naturally provide an inexpensive (albeit incomplete) form of CFI, guaranteeing that libraries can only be entered through well defined entry points, known and enforced at compile time.[3]

Data Ownership Approach. FlexOS takes a code-centered [227] isolation approach. Each library is present only once and maps to a specific set of privileges. There is a slight tweak for backends that rely on several systems (TrustZone, VMs): for them, the trusted computing base (§3.3.3) is duplicated; one for each system, as each compartment must possess a self-contained kernel (§3.4.2).

FlexOS considers all static and dynamic data allocated by a library as private by default. Individual variables can then be annotated as "shared" with a specific group of libraries into *whitelists*, similarly to access control lists. In practice, the maximum

[3]Note: we refer to this in Chapter 5 as *Cross-Compartment Control-Flow Integrity* (CC-CFI).

number of isolated data sharing "zones" is limited by the underlying technology. Annotations are made with the keyword `__shared` as illustrated in Figure 3.3 step ②.

Compiler annotations are identical for all types of variables. However, under the hood, FlexOS differentiates between statically allocated variables, dynamically allocated heap variables, and dynamically allocated stack variables.

FlexOS' compartmentalization API itself does not dictate *how* variables have to be shared. Different mechanisms can require very different sharing approaches: while certain mechanisms such as MPK require shared data to be located in shared memory regions, others such as CHERI's hybrid capabilities [453] require compiler annotations that can be automatically generated in place of the FlexOS placeholder. Section 3.4 describes the implementation of the API for the two supported backends (MPK/EPT), and sketches implementations for an additional one (CHERI).

Identifying shared data represents the vast majority of the porting effort. It is necessary for both kernel libraries, user libraries, and applications. On the kernel side, this problem is simplified (but not eliminated) by the modularity of Unikraft's codebase. This issue is not specific to FlexOS and is widely explored in the literature. State of the art approaches (1) rely on manual code annotations [344], (2) perform static analysis at compile time to identify shared data automatically [133], or (3) perform a mix of static, dynamic, and manual analysis [227]. There is no silver bullet: manual code annotation (*explicit* sharing) can be non-trivial, but typically produces precise results that not only take into account what *is* accessed across modules, but also what *should be* shared from a security perspective. Static-analysis-based approaches (*implicit* sharing), on the other hand, are automatic, but conservative. These methods would be applicable to FlexOS, however automated shared data identification is not the main focus of this paper. The current prototype relies on manual annotations, and Section 3.4 details the porting effort for a number of applications and libraries.

Build-time Source Transformations. Before compilation, FlexOS' toolchain performs source transformations to (1) instantiate abstract gates, (2) instantiate data sharing code, (3) generate linker scripts, and (4) generate additional code in core libraries according to backend-provided recipes. The amount code generated in considerable. As an example, the toolchain modifies about 1K LoC for a simple Redis configuration. Figure 3.3 steps ③ and ③' presents an example of the porting-transformation process.

3.3.2 Kernel Backend API

Most isolation mechanisms require changes to a specific set of components in the kernel. The kernel facilities that can require special handling depending on the technology exclusively correspond to the core libraries (see Figure 3.2). In order to make such changes scalable, we designed core components to expose a *hook API* to isolation backends, allowing the core libraries to be easily extended with backend specific functionalities. For example, the MPK backend leverages the thread creation hook offered by the scheduler to switch a newly created thread to the right protection domain. These hooks come at no cost: since the instantiation is done at build time, the compiler is able to aggressively inline such calls.

Porting FlexOS to use a new isolation mechanism does not require redesign. In general, it is equivalent to (1) implementing gates for the particular mechanism, (2) implementing hooks for core components (see previous paragraph), (3) implementing linker script generation in the toolchain, (4) implementing Coccinelle code transformations, and (5) registering the newly created backend into the toolchain. In practice, developers can heavily reuse existing transformations for new backends.

3.3.3 Trusted Computing Base

Regardless of the isolation mechanism, certain components are so deeply involved in the OS' functioning that they will cause the entire system to violate its safety guarantees when compromised. These components are (1) the early boot code, (2) the memory manager, (3) the scheduler, (4) the first-level interrupt handler's context switch primitives, and (5) the isolation backend. We refer to these components as FlexOS' Trusted Computing Base (TCB), illustrated in Figure 3.2. Clearly, malfunctioning or malicious early boot code can setup the system in an unsafe manner, the memory manager can manipulate page table mappings in order to freely access any compartment's memory, the scheduler can manipulate sleeping thread's register states, and the backend provide incomplete isolation, etc. This is the case even when considering architectural hardware capabilities such as CHERI [184]. It comes as no surprise: this core set of libraries is historically the set of services that microkernel OSes provide [422]. FlexOS' TCB is small: around 3K LoC in the case of Intel MPK, and even less for VM/EPT.

Trust Model. The whole point of flexible isolation is to be able to achieve a wide range of trust models where different components (such as the network stack, parser libraries, etc.) can be considered untrusted and potentially compromised. Thus there is

no single trust model for FlexOS. In general, however, we assume that the TCB (see previous paragraph) is safe and error free. This is not an unreasonable assumption given the small size and the potential for formal verification (we have formally verified a version of our scheduler [300] using Dafny [304]). The hardware and the compiler are also part of the TCB. Note that the rest of the toolchain (Coccinelle included) is *not* part of the TCB as the code includes compile time checks that are able to detect invalid transformations. Finally we must also assume that interfaces correctly check arguments and are free of confused deputy/Iago [161] situations. This is not an unreasonable assumption within the core FlexOS codebase. Further, confused deputy and Iago attacks are probabilistically made more complex to execute in FlexOS due to the variability of the interface size; the system call API, for example, is divided into a variable number of sub-interfaces depending on the chosen configuration, and several compartments may need to be subverted for an attack to be successful.

3.4 Prototype

We present a prototype of FlexOS on top of Unikraft [285] v0.5, with Intel MPK and EPT backends. Modification to the Unikraft kernel represent about 3250 LoC: 1400 for the MPK backend, 1000 for EPT, and 850 for core libraries. In user space, changes to Unikraft's toolchain represents 2300 LoC. We port user codebases (Redis, Nginx, iperf, and SQLite) as well as most kernel components (the TCP/IP stack, scheduler, filesystem, etc.) to run as isolated components. This section presents the MPK and EPT backends, sketches a CHERI backend, and concludes with the porting effort.

3.4.1 Intel MPK Isolation Backend

MPK is a mechanism present in Intel CPUs offering low-overhead intra-AS memory isolation [46, 129, 400]. MPK leverages unused bits in the page table entries to store a *memory protection key*, enabling up to 16 protection domains. The PKRU register then stores the protection key permissions for the current thread. On each memory access, the MMU compares the key of the target page with the PKRU and triggers a page-fault in case of insufficient permissions. FlexOS associates each compartment with a protection key and reserves one key for a shared domain used for communications. If the image features less than 15 compartments, FlexOS uses remaining keys for additional shared domains between restricted groups of compartments. Any compartment

can modify the value of the PKRU, thus the MPK backend has to prevent unauthorized writes. This has previously been done via runtime checks [236] and static analysis [434]. In FlexOS, no code is loaded after compilation, hence static binary analysis coupled with strict W⊕X (Write XOR Execute) [421] is sufficient.

MPK Gates. For flexibility, FlexOS offers two different implementations of the MPK gate. The main one provides full spatial safety, similarly to Hodor [236]. The gate protects the register set and uses one call stack per thread per compartment. Each compartment has a stack registry that maps threads to their local compartment stack, making it fast and safe to switch the call stack. Upon domain transition, the gate (1) saves the current domain's registers set, (2) clears registers, and (3) loads function arguments. It then (4) saves the current stack pointer, (5) switches thread permissions, (6) switches the stack, and finally (7) executes the `call` instruction. Once the function has returned, operations are executed in reverse.

The second gate implementation shares the stack and the register set across compartments, similarly to ERIM [434]. It is conceptually very simple, switching the content of the PKRU before performing a normal function call. This lightweight implementation offers lesser guarantees but presents a lighter overhead, close to the raw cost of `wrpkru` instructions.

Data Ownership. FlexOS' MPK images feature one "data", "read-only data", and "bss" section per compartment to store private compartment static data. At boot time, the boot code protects these sections with the compartment's protection key.

Each compartment has a private heap, and a shared one is used for communications. Our prototype uses a single shared heap for all shared allocations, but this is not a fundamental restriction. Stack allocations are slightly more complex. Existing works convert shared stack allocations to shared heap allocations [236, 277, 133]. This approach is costly from a performance perspective: an allocation+free on the fast path for a modern allocator typically takes 30-60 cycles, and up to thousands of cycles on the slow path [263]. This is as expensive as entire domain transitions, and that for a single shared stack variable. While FlexOS supports stack-to-heap conversions, we propose another approach that addresses this issue, the *data shadow stack*.

Data Shadow Stacks. Stack allocations are much faster than heap allocations because the compiler is able to perform bookkeeping *at compile time*. At runtime, a single

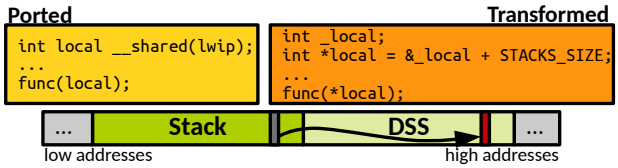


Figure 3.4: Data shadow stacks.

push instruction is needed, resulting in constant low cost. Data Shadow Stacks (DSS), illustrated in Figure 3.4, leverage this bookkeeping work for shared stack allocations.

When using the DSS, the usual stack size of threads is doubled. The upper part corresponds to the DSS and is put in the shared domain. The lower part is the traditional stack and remains in the compartment's private domain. For each shared variable `x`, we define the *shadow* of `x` as `&x + STACK_SIZE`. Thus, allocating space for a shared variable on the stack transparently allocates a shadow variable in the DSS. Before compilation, the toolchain replaces every reference to a shared stack variable with its shadow `*(&var + STACK_SIZE)` in the shared domain.

Allocations on the DSS are much faster than on a shared heap, since the DSS' bookkeeping overhead is null (stack speed), and the locality of reference high. The cost is a relatively small increase in memory usage (stacks are twice as large). The DSS mechanism is applicable to any isolation mechanism that supports shared memory, and is compatible with common stack protection mechanisms.

Control-Flow Integrity. Intel MPK does not enforce permissions on instruction fetches. Thus, if a compartment is compromised and the attacker ROPs into another compartment, a fault will not directly happen. The MPK backend is able to provide a form of CFI, ensuring that compartments can only be entered at well-defined points, a consequence of the hardcoding of gates as described in §3.3.1. If the control flow of a compartment is compromised and the attacker ROPs directly into another compartment C, then the system is guaranteed to crash when data local to C is accessed.

3.4.2 EPT/VM Backend

Virtualization has been used in many works to support isolation within a kernel [306, 348, 475, 349]. Hardware-assisted virtualization is widely supported and provides strong safety guarantees compared to MPK, at the cost of higher overheads. The EPT backend is an extreme case; compartments do not share ASes and run on different vCPUs. It shows that FlexOS is able to cater very different mechanisms under a common API.

FlexOS' EPT backend generates one VM image per compartment, each containing the TCB (boot code, scheduler, memory manager, backend runtime) and the compartment's libraries. Communications use a shared memory-based RPC implementation. Our prototype runs on QEMU/KVM patched to support lightweight inter-VM shared memory (less than 90 LoC).

EPT Gates. Upon domain transition, the caller places a function pointer and arguments in a predefined shared area of memory. All other VMs busy-wait until they notice an RPC request, check that the function is a legal API entry point, execute the function and place the return value in a predefined area of the shared memory. In order to support multithreaded loads, each RPC server maintains a pool of threads that are used to service RPCs. Using function pointers instead of abstract routine identifiers simplifies the RPC server's unmarshalling operation and does not prevent the RPC server from checking the pointer to ensure that it is a legal entry point. This optimization is possible since all compartments are built at the same time, hence all addresses are known.

Busy-waiting allows the EPT backend to minimize gate latency as opposed to VM notifications, but a similar implementation with `MONITOR/MWAIT` instructions would also be possible to minimize power consumption if calls are sparse. Overall, any of these tweaks can be implemented as gate variant to offer freedom to the user.

Data Ownership. The EPT backend relies on shared memory areas to share data (static and dynamic) across VMs. Areas are always mapped at the same address in the different compartments so that pointers to/in shared structures remain valid. Each VM manages its own portion of the shared memory area to avoid the need for complex (and error-prone or unsafe) multithreaded bookkeeping.

Control-Flow Integrity. Because it can enforce permissions on instruction fetches, the EPT backend is able to provide a form of CFI stronger than that of the MPK backend, ensuring that compartments can only be *left and entered* at well defined points. Indeed, the RPC server can control at entry that the executed function is legal, and compartments are not able to execute other compartments' code without RPC calls.

3.4.3 Supporting More Isolation Mechanisms

To check whether FlexOS can support other isolation backends, we discuss how we can leverage the CHERI hardware capabilities [454], an emerging isolation hardware mechanism. The CHERI ISA extension is available for ARMv8-A, which is supported

Table 3.1: Porting effort: size of the patch in lines of code (including automatic gate replacements) and number of shared variables.

Libs/Apps	Patch size	Shared vars
TCP/IP stack (lwIP)	+542 / -275	23
scheduler (`uksched`)	+48 / -8	5
filesystem (`ramfs`, `vfscore`)	+148 / -37	12
time subsystem (`uktime`)	+10 / -9	0
Redis	+279 / -90	16
Nginx	+470 / -85	36
SQLite	+199 / -145	24
iPerf	+15 / -14	4

by FlexOS. Among others, CHERI capabilities would extend FlexOS' trade-off space with the ability to address confused-deputy situations, reduce data sharing, and allow for a larger number of domains, something that is currently impossible for architectural (MPK) and performance (EPT) reasons. The backend would use boot-time hooks to initialize CHERI support, and scheduler hooks to perform capability-aware context-switching and thread initialization. Similarly to other backends, CHERI gates would save caller context, clear the relevant traditional and capability registers, install the callee context, and rely on the domain crossing instruction `CInvoke` and sentry capabilities [453] to perform protection domain jumps. As a first step, FlexOS should rely on the hybrid pointer model to maximize compatibility. Our API's shared data annotations would transform to `__capability` at build time to treat shared variables as a capabilities for efficient communications.[4]

3.4.4 Porting Effort

The porting process consists of two phases: call gate insertion (semi-automated), and shared data annotation (manual). The typical workflow, once gates have been inserted, is to run the program with a representative test case (e.g., a benchmark or test suite) until it crashes due to memory access violations. Crash reports point to the symbol that triggered the crash, at which point the developer can annotate it for sharing. In some cases, the crash can be a genuine violation; e.g., a library exposes internal state to external libraries, in which case the developer can decide to rework the library's API to address the privacy issue. This case is much less frequent and left at the developer's

[4]Note: We later implemented this proposal as part of Kressel at al. [283]. A substantial difference concerns the `__capability` annotation, which must be propagated to data derived from shared data.

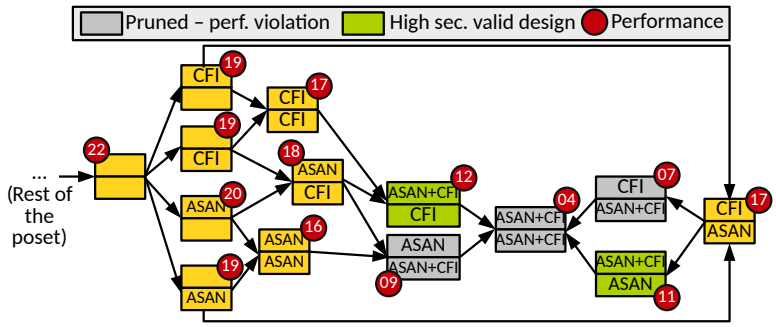


Figure 3.5: Partial view of the configuration poset for a fixed compartmentalization (2 compartments), varying per-compartment software hardening (CFI/ASan).

discretion. An example is `ramfs`, which is so deeply entangled with `vfscore` that blindly isolating it without redesign (an approach also referred to as *implicit* data sharing [269]) would impair performance with little additional security benefits, as a critical portion of the component's state would be shared. However, coupled with `vfscore`, both components can perfectly well be isolated from the rest of the system. This highlights a limitation of automated, implicit-sharing tools that blindly isolate this component [395]. Overall, the porting process is greatly simplified by common debugging tools: GDB and all usual debugging toolchains are supported. The debugging experience in FlexOS is not significantly different from Unikraft and most mainstream OSes, and we expect it to remain intuitive for anyone familiar with OS development. Depending on the amount of data shared with the outside world, the porting process ranges from 10 minutes (time subsystem, no data shared), to 2-5 days (filesystem, network stack). This porting cost is similar that of other compartmentalization frameworks [344]. Table 3.1 illustrates the porting effort with concrete numbers.

3.4.5 Software Hardening

The flexible isolation provided by FlexOS allows to enable or disable software hardening (SH) such as CFI, etc., on a per-component basis: isolating components without SH from components with it allows the latter to maintain the guarantees offered by SH. Moreover, many SH schemes work by instrumenting the memory allocator, and we use FlexOS' capacity to have an allocator per-compartment to enable flexible SH. This flexibility allows for example to alleviate the performance impact of SH by enabling it only for a subset of the system. Our prototype currently uses address sanitization (ASan), undefined behavior sanitization (UBSan), CFI, and stack protector.

3.5 Exploration with Partial Safety Ordering

In this section we present a design-space exploration technique, *partial safety ordering*, that aims to guide a user towards suitable configurations for a given use case by subsetting the vast design space enabled by FlexOS according to safety and performance requirements.

Given a performance budget, partial safety ordering attempts to find the most secure configurations among those enabled by FlexOS. Quantifying safety is challenging: it is impossible to give each configuration an absolute safety score that would allow to completely order them; for instance, is the safety of a configuration with 3 compartments, MPK isolation and no hardening better or worse than another one with 2 compartments, EPT isolation and CFI hardening?

Nevertheless, the safety of *some* configurations is programmatically comparable. Consider three configurations, C_1 with no isolation and no software hardening; C_2 with two compartments protected by a given mechanism with a given data sharing strategy and no hardening; and C_3 adding CFI for each compartment on top of C_2. In terms of (probabilistic) safety, we have the following relationship: $C_1 \leq C_2 \leq C_3$. With that in mind, it is thus possible to organize all configurations into a *partially ordered set* (poset), that can be viewed as a Directed Acyclic Graph (DAG) for which each node represents a configuration, and a directed edge between nodes n_1 and n_2 indicates that the level of safety of n_1 is probabilistically superior to that of n_2. The safety of nodes on the same path is comparable, while that of nodes on different paths is not.

Figure 3.5 presents a subset of the configuration poset corresponding to fixed choices for a compartmentalization strategy with 2 compartments, an isolation mechanism, and a strategy of data sharing. This subset of the poset represents the variation of the last feature, the software hardening, for which we assume only CFI and ASan for the sake of simplicity. Each configuration is depicted by a node indicating, for each of the two compartments, which hardening mechanism is applied: none, CFI, ASan, and CFI+ASan. We construct the poset partially depicted on Figure 3.5, ordering safety with the assumption that safety probabilistically increases with 1) the number of compartments; 2) data isolation (isolated vs. shared stacks, dedicated shared memory areas per pair of communicating compartments vs. shared areas accessible from everywhere, etc.); 3) stackable software hardening; and 4) the strength of the isolation mechanism.

Given such a poset, we can label each node with its performance characteristics (circles in the figure denote fictional performance numbers), and prune those that don't meet minimum requirements (gray nodes), ultimately yielding a set of configurations

that offer the best guarantees for a given performance budget. This set corresponds to the *maximal elements* of the poset, i.e., sinks of the DAG (green nodes in the figure).

Partial Safety Ordering in Practice. In practice, users provide the toolchain with a test script (e.g., `wrk` for Nginx) and a performance budget (e.g., at least 500K requests per second). Users are free to define performance as they may deem suitable depending on their needs: application throughput, tail latency, runtime, etc. Any metric is suitable as long as it remains comparable across configurations and runs. With this in hand, the toolchain generates the unlabeled poset. Then, it labels it by automatically measuring the performance of each configuration. The toolchain does not have to run all configurations: assuming monotonically decreasing performance, it can safely stop evaluating a path as soon as a threshold is reached. In practice, we observe that this significantly limits combinatorial explosion. The result is a set of the most secure configurations for the given budget, which the user can use to choose the most suitable one for a given use case. Ultimately, we expect this process to significantly trim the design space and allow the user to make an informed and relatively effortless choice.

This approach assumes that the user is able to get representative feedback on the application's performance, and users will not be able to use FlexOS' exploration facilities if they are not able to properly benchmark their application. However, we expect this situation to be quite rare: in the vast majority of cases, users will be able to at least minimally test their applications. These results can be used to exclude configurations that are too costly and test the best candidates in production using lightweight performance measurement systems, e.g., blue-green deployments.

Skipping Exploration. Some developers might already come with a particular isolation strategy in mind. In that case the developer can skip this exploration phase by providing a configuration file as shown in Section 3.3. In this case, the developer leverages FlexOS' flexibility and not its exploration facilities. We note, however, that this "expert" approach has its limits: applications evolve over time and a compartmentalization approach that is deemed optimal at a given time may not be suitable in the future [227]. In this case, an exploration system such as FlexOS' can be of use for the expert to easily reconsider their approach in light of changing software.

3.6 Evaluation

We aim to demonstrate the vast performance/safety design space enabled by FlexOS, assess the efficiency of the partial safety ordering exploration technique, and compare FlexOS' performance with the literature. To this end, we present an overview of the performance obtained with numerous safety configurations on three popular cloud applications (Redis, Nginx, and SQLite), as well as iPerf, a standard network stack benchmark. We demonstrate our design-space exploration technique with Redis and Nginx. Then, we compare selected SQLite configurations with Linux, CubicleOS [395], a (non-flexible) compartmentalized LibOS, the SeL4 [278]/Genode [204] microkernel, as well as Unikraft [285]. Finally, we study raw isolation overheads in FlexOS: DSS efficiency and cross-compartments call gate latencies.

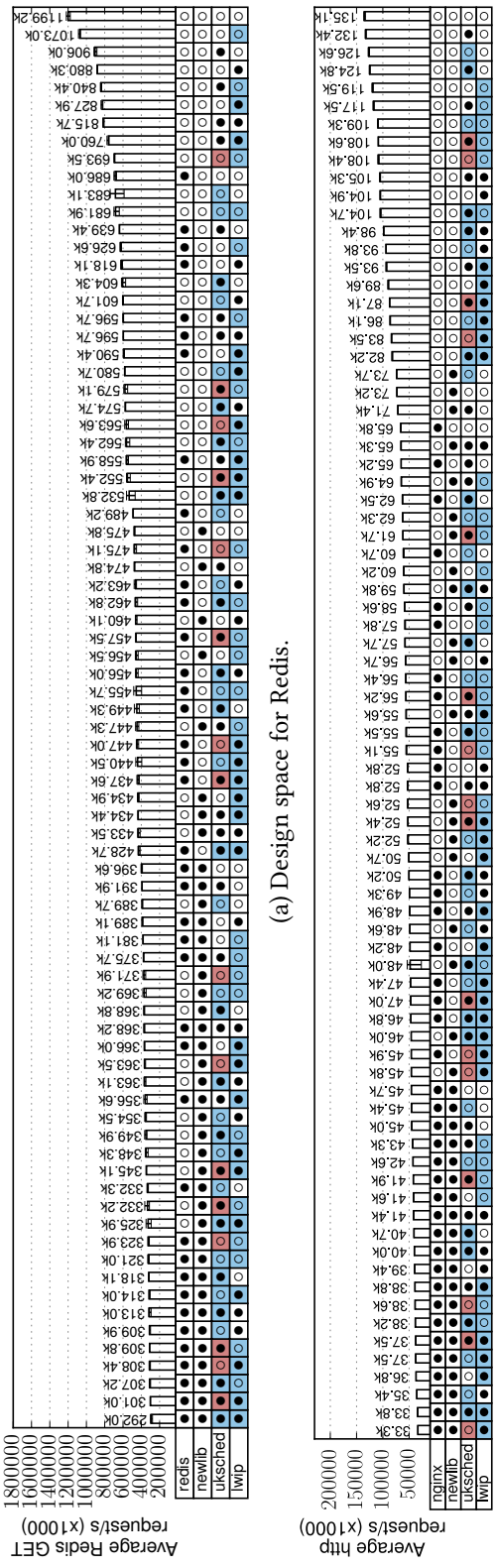
We run experiments on an Intel Xeon Silver 4114 @2.2 GHz. For each experiment we use 4 cores from the same socket, isolated with *isolcpu*: 2 cores for the client (iPerf client/`redis-benchmark`, `wrk`) on the host, 1 core for the QEMU process, and 1 core per FlexOS' vCPU. Hyperthreading is disabled.

3.6.1 Design-Space Exploration: Redis, Nginx

We automatically generate and run a large set of configurations for Redis and Nginx using the Wayfinder [261] benchmarking platform. We fix the isolation mechanism to MPK with DSS and vary: the number of compartments (1-3), compartmentalized components (TCP/IP stack, libc, scheduler, application), as well as per-compartment software hardening (stack protector, UBSan and ASan), for a total of 2x80 configurations.

Redis. The results are on Figure 3.6a (top), plotting for each configuration Redis' GET throughput. Overall we observe that FlexOS enables for a very wide range of safety configurations with significant performance variation: there is one order of magnitude of difference between the configuration yielding the lowest throughput (292K requests/s) vs. the highest one (1.2M requests/s).

Unsurprisingly, the configuration that disables isolation and hardening gives the highest throughput. Conversely, configurations with many compartments/hardening perform worst. Still, in between these two extremes, creating more compartments and enabling hardening has a variable impact on performance. For example, with two compartments and no hardening, isolating lwIP from the rest of the system leads to an 11% performance hit, while that number reaches more than 43% when the scheduler is the isolated component — indicating extensive communication between user code and



(a) Design space for Redis.

(b) Design space for Nginx.

Figure 3.6: Redis (top) and Nginx (bottom) performance for a range of configurations. Components are on the left. Software hardening can be enabled [●] or disabled [○] for each component. The white/blue/red color indicates the compartment the component is placed into. Isolation is achieved with MPK and DSS.

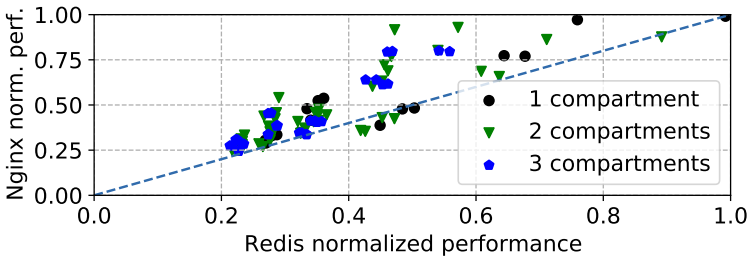


Figure 3.7: Nginx versus Redis normalized performance.

the scheduler. The same is true for hardening: with a single compartment, enabling hardening on the scheduler has a 24% performance cost, while that cost is 42% when hardening the Redis application code.

The complexity of maximizing safety and performance becomes more clear when isolating several components: isolating lwIP from the scheduler from the rest only differs from a few percentage points from isolating lwIP together with the scheduler from the rest. Such "isolation for free" effects are caused by communication patterns; lwIP does not directly communicate with the scheduler, hence the "cut" is not on a hot path, and merging them in a same compartment brings little performance benefits. Thus, the performance does not entirely depend on the number of compartments or the number of components with hardening enabled, but rather *what* particular components are isolated/hardened, and their communication patterns. Such effects can be leveraged to maximize safety and performance.

Nginx. The results are on Figure 3.6b (bottom), plotting for each configuration Nginx' HTTP throughput. Overall we observe that results span over the same range of overhead as Redis (0-4.1x). However, overheads do not follow the same distribution: 9 configurations have less than 20% overhead in the Nginx case, but only 2 for Redis. Similarly, 32 configurations have less than 45% of overhead, only 20 for Redis. This can be explained by looking more closely at individual configurations. Compared to Redis, isolating the scheduler is much less expensive (6% versus 43% for Redis), and the same goes for hardening (2% versus 24% for Redis). The costs, however, become similar as more hardening and isolation boundaries are added because of bottleneck effects.

This different distribution of costs is made more clear by Figure 3.7 which compares the relative performance of configurations for Nginx and Redis (same dataset as Figure 3.6). These differences show that isolating and hardening the *same components*

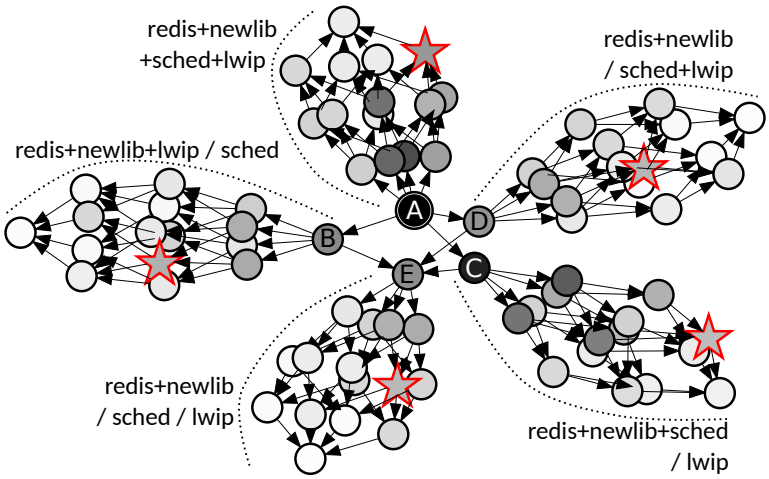


Figure 3.8: Configurations poset for the Redis numbers (Figure 3.6a). Stars are the most secure configurations with performance ≥500K requests/s.

on two networked applications results in uneven, difficult to predict slow-down. Existing approaches assume a one-size fits all safety configuration are therefore suboptimal; in contrast, FlexOS enables users to easily navigate the safety / performance trade-off inherent in their application.

3.6.2 Partial Safety Ordering

We applied this technique on the Redis numbers from Figure 3.6a. We construct the poset presented in Figure 3.8, where each node is a Redis configuration, i.e., a column from Figure 3.6a. The node's color intensity indicates the configuration's performance, black being the fastest (1.2M requests/s) and slower configurations becoming gradually white (pure white representing 292K requests/s). The fastest configuration is the one with no isolation and no hardening (Ⓐ on Figure 3.8). Other nodes in the center of the plot represent compartments addition, still with no hardening: separating from the rest of the system either the scheduler Ⓑ, lwIP Ⓒ, or Redis+newlib Ⓓ, and a 3 compartments scenario Ⓔ. From these 5 basic compartmentalization strategies come out 5 "branches". The nodes in each branch represent various combinations of per-component software hardening. The nodes' color evolution indicate the variable performance impact of creating new compartments and stacking software hardening on components.

We set a minimum required performance of 500K requests/s, and let partial safety ordering identify the safest configurations satisfying that constraint, indicated with

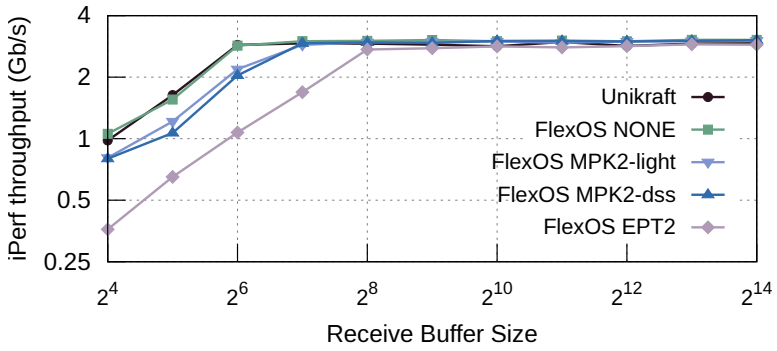


Figure 3.9: Network stack throughput (iPerf) with Unikraft (baseline), FlexOS without isolation, with two compartments backed by MPK (*-light* = shared call stacks, *-dss* = protected and DSS), and with two compartments backed by EPT.

stars on Figure 3.8. In this case, the technique can prune the configuration space from 80 to 5 configurations, helping the user easily pick the most appropriate one.

3.6.3 Batching Effects: Network Stack Throughput

We port a simple iPerf server to FlexOS and use it to measure the network performance of our system. We fix the compartmentalization to the following scenario: the iPerf application code is placed within a compartment, and the rest of the system (including the network stack) is placed in a second compartment. We apply no software hardening, and configure the iPerf server to pass buffers of varying sizes when calling `recv` on the socket. We measure the achieved throughput using an iPerf client for FlexOS without isolation, with MPK (sharing or protecting the call stack), as well as EPT. We run vanilla Unikraft as baseline.

The results are on Figure 3.9. FlexOS without isolation performs similarly to Unikraft, confirming that users "only pay for what they get". FlexOS' isolation slowdown manifests for small payload sizes, for which the domain crossing latency is an important bottleneck in the request processing time. Depending on the buffer size, EPT isolation is 1.1-2.2x slower than MPK with DSS, which is itself 0-1.5x slower than the baseline without isolation. MPK with shared stacks bears a 0-1.3x slowdown. Although MPK with DSS pays the price of a stack switch (see Table 3.11b), it is more secure than fully sharing the stack and still faster than fully isolating it while moving shared data to the heap (see Figure 3.11a). Batching effects clearly manifest as the payload size increases: MPK's performance quickly becomes similar to to baseline starting from 128 B. EPT's

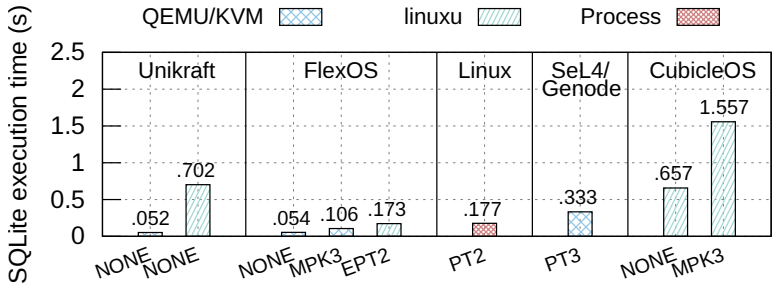


Figure 3.10: Time to perform 5000 INSERT queries with SQLite on Unikraft, FlexOS, Linux, SeL4 (with the Genode system), and CubicleOS. The isolation profile is shown on the x axis (NONE: no isolation, MPK3: MPK with three compartments, EPT2: two compartments with EPT, PT2/3: two/three compartments with page-table-based isolation).

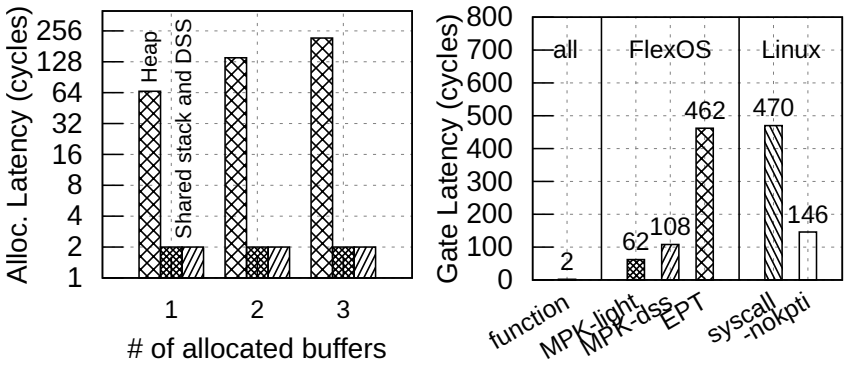
isolation being more costly, the payload size needs to be 256 B or above so that its performance to reach about 90% of the baseline's. These results illustrate that, depending on the size of the payload and the frequency of domain crossings, all backends can constitute a valid solution to a given problem.

3.6.4 Filesystem Intensive Workloads: SQLite

We evaluate the performance of FlexOS with filesystem intensive workloads and compare it to vanilla Unikraft, Linux, SeL4 [278] with the Genode [204] system, and CubicleOS [395]. Though both FlexOS and CubicleOS extend Unikraft, the former runs in a standard QEMU/KVM VM while the latter is implemented on top of *linuxu* [95], Unikraft's userland debug platform. The Unikraft baseline number thus cover both cases. We evaluate two scenarios: one with two components (EPT2, PT2), where the filesystem is isolated from the application, and one with three components (MPK3, PT3), where the filesystem is isolated from the time subsystem from the rest of the system. This benchmark performs 5000 sequential INSERT queries. To increase filesystem pressure, each query is in a separate transaction. The results are shown in Figure 3.10.

Compared to the baseline, FlexOS without isolation adds no overhead, and MPK3 adds an overhead of 2x. This is still significantly faster than the userland Linux version which performs a large number of system calls, highlighting the benefits of the LibOS basis. Somewhat surprisingly, FlexOS with EPT2 performs almost identically to Linux. This is because the system call latency is almost identical to the EPT2 gate latency on this system (see Figure 3.11b). Compared to SeL4, FlexOS is 3.1x faster with MPK3, and 2x faster with EPT2.

Compared to CubicleOS, FlexOS is an order of magnitude faster. This is due to (1)



(a) Allocation latencies (b) Gate latencies

Figure 3.11: FlexOS latency microbenchmarks.

CubicleOS relying on *linuxu*, i.e., running in Ring 3 and performing Linux system calls for privileged operations[5], (2) CubicleOS not implementing MPK support and relying on Linux `pkey_mprotect` system calls (making domain transitions orders of magnitude more expensive and the TCB thousands of times larger), and (3) CubicleOS' *trap-and-map* approach (which FlexOS avoids with shared data annotations). Even compared to its baseline without isolation, CubicleOS with MPK3 adds an overhead of 2.4x, about 30% more than FlexOS. CubicleOS without isolation is faster than the Unikraft linuxu baseline; this is because it uses the Lea [295] memory allocator which behaves better than Unikraft's TLSF [331] allocator in this benchmark.

3.6.5 Overheads: Stack Allocations, Gate Latencies

In FlexOS, stack data can be shared via heap allocations, using the DSS (trading space for performance), or sharing the stack entirely (trading safety for performance). To illustrate the benefits of the DSS, we measure, for each approach, the execution time of a function that allocates 1 to 3 shared 1-Byte stack variables and returns immediately.

The results are on Figure 3.11a. Heap-based stack allocations are one to two orders of magnitude (100-300+ cycles) slower than typical stack allocations (constant 2 cycles). This is not surprising, since general-purpose allocators typically feature unbounded execution time. This cost increases with the number of variables, since each

[5]The *linuxu* [95] Unikraft debug platform allows Unikraft developers to interact with Unikraft as a normal Linux ELF binary, considerably simplifying the debugging of subsystems that can run in userspace. It was, however, never meant for performance-sensitive workloads or practical deployments. Privileged operations that require system calls in linuxu include, for instance, scheduling, which makes a considerable number of calls to `rt_sigprocmask` throughout the execution of the benchmark.

variable triggers a separate call to malloc. The DSS matches the shared stack in performance, confirming that it combines the safety of isolation with the performance of traditional stack allocations. The memory footprint increase due to the DSS is modest as FlexOS uses small stacks (8 pages). For example, an instance with Redis (8 threads), has a space overhead of 288 KB. The DSS is a data sharing strategy and does not remove the need to perform stack switches.

Another source of compartmentalization overhead is gate latency. To illustrate the raw performance of FlexOS' gates we measure the gate latency of MPK stack-sharing gates (*-light*), normal MPK gates, and EPT gates. We compare them with the latency of a function call, and of a Linux system call (with and without Kernel Page-Table Isolation/KPTI [222], *-nopkti*). The results are shown in Figure 3.11b. MPK light gates are 80% faster than normal MPK gates, and 7.6x faster than EPT gates, as they correspond to the cost of raw `wrprku` instructions. EPT latencies are similar to system call latencies without KPTI, illustrating the practicability of the EPT backend.

3.7 Use Cases for Isolation Flexibility

FlexOS enables developers to seamlessly experiment with various safety configurations for their OS. An obvious use-case we presented throughout this paper is the specialization of the OS' safety strategy for a given application: manually or semi-automatically selecting, among the vast design space unlocked by FlexOS, the most suitable configuration for a particular use case with given safety/performance constraints. Still, there are many other ways in which this flexibility can be used; we detail some of them next.

Quickly Isolate Exploitable Libraries. Consider the period between the full disclosure of a vulnerability and the release of its fix, or the embargo period when vulnerabilities are disclosed only to affected vendors, but not to the general public; these periods can last for weeks up to years during which vulnerable software runs in the wild. With FlexOS, it takes seconds to create a new binary that isolates a vulnerable library into its own compartment (e.g., EPT + hardening) to at least mitigate the effects of exploits; an automated system could be created to respond to known vulnerabilities by recompiling production software to isolate certain libraries, similar to Self-Certifying Alerts [176]. Such flexibility improves over the state of the art by avoiding a loss of functionality (e.g., compared to Senx [250]), and providing excellent resistance to polymorphic variations of vulnerabilities (e.g., compared to filters [175]).

Quickly React to Hardware Protections Breaking Down. Recent hardware vulnerabilities [314, 279] showed that hardware-backed isolation mechanisms are not foolproof. The corresponding fixes may require significant engineering and redesign efforts (e.g., KPTI for Meltdown), leading to long vulnerability windows. FlexOS is not immune to hardware vulnerabilities by design. In this case, however, its ability to easily switch between protection techniques comes handy: by supporting a wide range of isolation primitives relying on a range of different hardware, switching the isolation mechanism from a vulnerable to a non-vulnerable one is just a matter of rebuilding the LibOS with a different configuration (snippet in §3.3), i.e., the engineering cost is nil.

As Secure as You can Afford. Consider a service provider who wishes to offer the best possible security as long as its server can keep up with the client load. A natural approach would be to run the safest combination that copes with peak load, as we suggested in our Redis evaluation; this means that in periods of low load the system has idle compute power.

With its capacity to quickly switch safety configurations, FlexOS enables another approach: to run, at any time, the safest configuration that can sustain the actual load. This makes attacks much harder as long as the system is under-loaded, but gracefully switches off defenses as load increases to respect SLA. Another approach is to couple this with software load balancers to triage users into likely benign or malicious, sending them to machines running faster or safer software, accordingly.

Dealing with Crashed Software. Vulnerabilities are a fact of life, and the standard approach is to quickly restart crashed software and to examine the faults in the background. When such a crash is detected (e.g., memory error), with FlexOS it is wiser to start a safer configuration of the same software, to ensure that any vulnerability is not turned into an exploit.

Incremental Verification. Individual components of FlexOS can be verified and isolated from the rest of the system. In this way, one can obtain strong guarantees on preconditions and ensure that verified properties hold even when mixed with unverified components, something that isn't possible with monolithic operations systems [308]. Over time, the entire system could be verified, gradually increasing the guarantees of the system.

Deployment to Heterogeneous Hardware. The flexibility of FlexOS mechanisms can also come in very handy when deploying on heterogeneous hardware. Some servers might offer MPK support for example, others CHERI, others only the classical MMU. In every case, Chrysalis is able to get the best from the available hardware without major rewrite, and without requiring insider knowledge from application developers.

3.8 Related Work

Improving OS safety. Previous work proposed to address the safety issues of monolithic OSes by reducing the TCB through separation [387, 116], micro-kernels [216, 238], and safe languages [142, 346, 251, 323, 180]. In SASOSes, internal isolation may be traded off for performance [264, 276, 352, 285], provided with traditional page tables [158, 305, 237, 349], or intra-AS hardware isolation mechanisms [395, 307, 417, 351]. Other research efforts strive to speedup IPC in microkernels [224, 336], or redesign monolithic OSes entirely [235, 420, 155, 306, 145, 348, 183].

Overall, each of these approaches is a single or a few point(s) in the OS safety/performance design space and lacks the flexibility of FlexOS to automatically specialize for safety or performance. LibrettOS [349] allows a LibOS to switch between SASOS and microkernel modes, but remains limited to a small subset of the safety/performance design space.

Compartmentalization Frameworks. Several compartmentalization frameworks have been proposed recently [434, 236, 344, 400, 227, 318, 133, 395]. Contrary to FlexOS, none focuses on flexible isolation. Regarding application porting, most [434, 236, 344, 400] rely on code annotations. A few studies provide various degrees of porting automation through data-flow analysis [227, 318, 133], but are typically bound to numerous limitations due to the complexity of breaking down monolithic codebases. Nevertheless, some of their principles can be applied to increase the degree of automation of FlexOS' porting process – something we scope out as future works. CubicleOS [395] proposes a *trap and map* mechanism to limit the porting effort, but this comes at a high cost, is specific to MPK, and is not entirely automated. Further, as shown in our evaluation, CubicleOS' reliance on Unikraft's *linuxu* leads to suboptimal performance.

3.9 Conclusion

The isolation strategy of today's OSes is mostly fixed at design time. This lack of flexibility is problematic in many scenarios. We propose FlexOS, an operating system whose isolation strategy is decoupled from its design. We augment the historical capacity of the library OS to specialize towards performance with the ability to specialize for safety: fundamental decisions such as the compartmentalization granularity and which isolation mechanism to use are deferred to build time. FlexOS ships with a semi-automated exploration strategy helping the user navigate the vast configuration space the system unlocks. FlexOS is available online at `https://project-flexos.github.io`[6] under an open source license.

In our future work, we intend to add more isolation backend implementations to FlexOS including CHERI and SGX, as well as support for more software hardening techniques. Another direction is to create a formal basis to help users navigate the safety configuration space. This would enable, among others, embedding formally verified components in FlexOS configurations while preserving their proven properties.

Acknowledgements

We would like to thank the anonymous reviewers, and our shepherd, Gerd Zellweger, for their comments and insights. A similar thanks goes to our colleague Marc Rittinghaus for his insights. We are immensely grateful to the Unikraft OSS community for their past and ongoing contributions. This work was funded by a studentship from NEC Labs Europe, EU H2020 grants 825377 (UNICORE), 871793 (ACCORDION) and 758815 (CORNET), as well as the UK's EPSRC grants EP/V012134/1 (UniFaaS) and EP/V000225/1 (SCorCH). UPB authors were partly supported by VMWare gift funding.

[6] See Appendix C.

Chapter 4

Assessing the Impact of Interface Vulnerabilities in Compartmentalized Software

> If the enemy leaves a door open,
> you must rush in.
>
> *The Art of War*, 11.65
> Sun Tzu

This chapter presents a large-scale study of *Compartment-Interface Vulnerabilities* (CIVs), the confused deputy vulnerabilities which arise at compartment boundaries, along with ConfFuzz, a fuzzer specialised to detect CIVs at compartment interfaces. By better understanding these vulnerabilities, we aim to draw attention to the problem and pave the way for effective, adequate, and widespread mitigations. In the bigger picture of this thesis, better understanding and solving CIVs is an essential step towards more quantifiably secure compartmentalisation. These contributions address research questions **RQ3** and **RQ4**, and are derived from Lefeuvre et al. 2023 [298].

Contributions of the Author

I designed and implemented ConfFuzz. Using ConfFuzz, I gathered most of the data set presented in the paper. I analysed the data presented, and wrote the majority of the paper. Vlad-Andrei Bădoiu and Yi Chien helped me running ConfFuzz on part of the

39 scenarios presented in the paper (V-A. Bădoiu: 5-10 scenarios, Y. Chien: 1 scenario). Vlad-Andrei Bădoiu helped me run ConfFuzz on FlexOS [4] (not included in the paper for space reasons). Felipe Huici and Nathan Dautenhahn provided feedback on the paper. Pierre Olivier provided feedback on the paper and contributed edits.

Abstract

Least-privilege separation decomposes applications into compartments limited to accessing only what they need. When compartmentalizing existing software, many approaches neglect securing the new inter-compartment interfaces, although what used to be a function call from/to a trusted component is now potentially a targeted attack from a malicious compartment. This results in an entire class of security bugs: Compartment-Interface Vulnerabilities (CIVs).

This paper provides an in-depth study of CIVs. We taxonomize these issues and show that they affect all known compartmentalization approaches. We propose ConfFuzz, an in-memory fuzzer specialized to detect CIVs at possible compartment boundaries. We apply ConfFuzz to a set of 25 popular applications and 36 possible compartment APIs, to uncover a wide data-set of 629 vulnerabilities. We systematically study these issues, and extract numerous insights on the prevalence of CIVs, their causes, impact, and the complexity to address them. We stress the critical importance of CIVs in compartmentalization approaches, demonstrating an attack to extract isolated keys in OpenSSL and uncovering a decade-old vulnerability in sudo. We show, among others, that not all interfaces are affected in the same way, that API size is uncorrelated with CIV prevalence, and that addressing interface vulnerabilities goes beyond writing simple checks. We conclude the paper with guidelines for CIV-aware compartment interface design, and appeal for more research towards systematic CIV detection and mitigation.

4.1 Introduction

The principle of least privilege has guided the design of safe computer systems for over half a century by ensuring that each unit of trust in a system can access only what it truly needs to fulfill its duties: in this way, system designers can proactively defend against unknown vulnerabilities [454]. Software compartmentalization is a prime example where unsafe, untrusted, or high-risk components are isolated to reduce the damage they would cause should they be compromised [392].

Recent years have seen the appearance of an increasingly large number of new isolation mechanisms [177, 121, 120, 454, 400, 359] that enable fine-grained compartmentalization. This resulted in compartmentalization works targeting finer and finer granularities, such as libraries [462, 434, 236, 344, 400, 318, 133, 395, 299, 114], modules [252, 114, 399], files [114], and even functions/blocks of code [227, 451, 417, 112]. In that context, major attention was dedicated to compartmentalizing existing code, since rewriting software from scratch to work in a compartmentalized manner is costly and complex [227]. With recent developments on compiler-based compartmentalization, frameworks offer to apply isolation at arbitrary interfaces for a low to non-existent porting cost [462, 133, 318, 112].

Unfortunately, breaking down applications into compartments means that control and data dependencies through shared interfaces create new classes of vulnerabilities [436]: in order to provide safe compartmentalization, it is not only necessary to ensure spatial memory isolation but also to design interfaces with distrust in mind. For example, objects passed through APIs can be corrupted to launch confused deputy attacks [327, 244], data structures can be manipulated to control execution or leak data through Iago attacks [161, 179], called components can modify return values or indirectly access shared data structures to launch new forms of exploit, etc.

Even though interface-related vulnerabilities (denoted *Compartment-Interface Vulnerabilities* / CIVs in this paper) were previously identified to various extents in the literature [327, 161, 244, 436], almost all modern compartmentalization frameworks [462, 434, 236, 400, 318, 281, 359, 133, 395, 417, 300, 299, 112] neglect the problem of securing interfaces, and rather focus on transparent and lightweight spatial separation. Since CIVs are already problematic for interfaces hardened from the ground up (e.g., the system call API [293, 161]) with well-defined trust-models (kernel/user), their impact on safety is likely to be even greater when considering arbitrary interfaces and trust models that materialize when compartmentalizing existing software that was not designed with the assumption of hostile internal threats. Worse still, the complexity of

safeguarding interfaces increases as more fine-grain components are targeted.

Beyond this lack of consideration, CIVs remain misunderstood; we ask the following research questions: *how widespread are CIVs when compartmentalizing unmodified applications? What are the API design patterns leading to them? What is the concrete impact of CIVs on the safety guarantees brought by compartmentalization, and what is the complexity of addressing them?* In order to achieve CIV mitigations that are generic and principled, we stress the need to formalize and quantify the problem.

This paper provides an in-depth study of CIVs. We taxonomize CIVs into a coherent framework, and systematize existing efforts to address them, highlighting categories that need attention in future research. In order to study existing CIVs in real-world scenarios, we propose ConfFuzz, an in-memory fuzzer specialized to detect CIVs at possible compartment boundaries. ConfFuzz automatically explores the complexity of compartment interfaces by exposing data dependencies leading to vulnerabilities. Contrary to existing fuzzers, that inject malformed data in a single direction (e.g., a library), ConfFuzz can show the degree to which data flowing through an interface can be manipulated to harm either direction of a cross-compartment call. We apply ConfFuzz to a corpus of 39 compartmentalization scenarios, many of which previously proposed as use-cases of 12 existing research and industry frameworks. We uncover a wide dataset of 629 potential vulnerabilities. We systematically study these issues, extracting numerous insights on the prevalence of CIVs, their causes, impact, and the complexity to address them.

At the highest level, our results confirm how important the problem of CIVs should be to compartmentalization research: in many cases, CIVs seriously reduce or even fully negate all benefits of compartmentalization, and that even when the interface is extremely simple: we demonstrate an attack to extract isolated keys in OpenSSL, a common application of compartmentalization, and a decade-old vulnerability in sudo's authentication API. Beyond this, we note the following high-level insights: 1) CIVs are present in almost all existing interfaces, but at significantly varying degrees: for instance module APIs are much more vulnerable than library interfaces, and some interfaces are entirely CIV-free; 2) the complexity of objects crossing the interfaces imports more than the size of the API itself, and most of an API's CIVs can often be tracked down to a handful of objects; 3) fixing CIVs goes further than writing a few checks, and often requires reworking interfaces and partially redesigning existing software. We conclude with an appeal for more research towards systematic CIV detection and mitigation, hoping that this study can encourage future works to consider the issue of interfaces.

To sum up, this paper makes the following contributions:

- A systematization and taxonomy of CIVs and existing efforts to solve them (§4.3).
- ConfFuzz, an in-memory fuzzer that automatically detects CIVs in existing software at arbitrary interfaces (§4.4).
- A systematic study of the CIVs found by ConfFuzz applied to 39 real-world compartmentalization scenarios, backing insights with concrete data (§4.5).
- A series of interface design guidelines intended to ease the development/adaptation of new/existing interfaces with compartmentalization in mind (§4.6).

4.2 Motivation

The problem of secure interface design is not new [234]. The Linux system call interface, for example, is the result of years of organic evolution towards a strong boundary that preserves the integrity of the kernel in the presence of untrusted applications. Alas, designing strong interfaces in an adversarial context is notably hard: interface-related vulnerabilities are still regularly reported against the system call interface [293, 282, 267], even after decades of hardening. The task is even harder when assuming mutual distrust; the system call API, to take the same example, is notably weak at protecting the application from the kernel [161, 179], and requires extensive shim interfaces [106, 159, 370, 218, 207] to sanitize untrusted inputs and outputs.

Modern compartmentalization frameworks enable users to easily enforce spatial and temporal separation between components of existing software. Typically, code is either *sandboxed*, where a software component prone to subversion is restricted from accessing the rest of the system (e.g., image processing libraries), *safeboxed*, where sensitive data is only accessible to a component while maintaining high privilege (e.g., libssl), or *separated* into mutually distrusting subsystems [292]. Depending on domain crossing frequencies, compartmentalization promises good vulnerability containment at a modest cost [454].

Unfortunately, as shown by previous works [244, 436, 344] and highlighted in this paper, simply isolating software components is not enough: if cross-compartment interfaces have not been designed as trust boundaries (e.g., when compartmentalizing existing software), a wide range of *Compartment-Interface Vulnerabilities* (CIVs) arise. Reasoning about the safety of an interface is complex due to data dependencies exposed through the use of that interface by actors that previously belonged to a single trust domain, but under compartmentalization distrust each other. That complexity increases

```
// ImageMagick callback exposed to libghostscript
static int MagickDLLCall GhostscriptDelegateMessage(
  void *handle, const char *message, int length) {

  /* CIV: unchecked dereference/usage of
   * sandbox-provided pointer/bounds information */
  (void) memcpy(*handle, message, (size_t) length);
  (*handle)[length] ='\0';
} /* ... abbreviated / simplified ... */
```

Listing 3: ImageMagick callback lets libghostscript perform arbitrary writes outside the sandbox.

with that of interface-crossing data flows. CIVs arise when developers would like to avoid trust in a component, and encompass traditional confused deputies [234], Iago vulnerabilities [161], or Dereferences Under Influence (DUIs) [244]. We define CIVs more formally in §4.3.

Take the example of library sandboxing in ImageMagick as done by the Compiler-Assisted Library Isolation (Cali) [133] framework. Here, libghostscript is sandboxed because it is notoriously prone to high-impact vulnerabilities. Cali automatically sandboxes the library by applying compiler-based techniques to detect data shared between the application and the library, and place them in a shared memory region, before running application and library in separate processes. When the application needs to execute a function of the library, Cali performs the function call in the library compartment. Whenever the library needs to execute an application callback, Cali executes the callback in the application compartment.

This approach might seem sufficient: it makes it harder for attackers to escape the sandboxing of libghostscript. In practice, however, as shown in Listing 3, ImageMagick exposes a callback to libghostscript that allows the untrusted library to perform arbitrary writes in the application's compartment as often as it wants and at any time, negating spatial isolation entirely. This vulnerability, identified by our tool ConfFuzz, is caused by ImageMagick (victim compartment) dereferencing sandbox-provided pointers (`handle` and `message`) and bounds information (`length`) without sufficient checking.

Even though CIVs particularly affect new fine-grain compartmentalization frameworks such as Cali, they are not a specificity of these frameworks; as we show in §4.5, CIVs also affect long-standing, production-grade sandboxing approaches such as the worker/master separation in Nginx.

Clearly, while a strong compartmentalization framework capable of reliably enforcing spatial and temporal isolation is necessary, it is insufficient to offer tangible security benefits: software must also be adapted to fit distrust scenarios by vetting information that crosses compartment interfaces.

In the remainder of this paper, we propose the first systematization and taxonomy of CIVs and existing defenses, introduce ConfFuzz, a tool to automatically detect CIVs at potential compartmentalization boundaries, and use it to provide an in-depth study of real-world CIVs found with ConfFuzz.

4.3 Compartment-Interface Vulnerabilities

In this section, we provide the first definition and taxonomy of CIVs, along with a systematic review of existing defenses and their shortcomings. We define three main classes of CIVs, subdivided in a total of 8 subclasses. We relate each subclass with existing mitigations and discuss their limits, summarized in Table 4.1. Here we use the term *corrupted* to refer to data voluntarily malformed by a malicious compartment.

> **Definition:** A *Compartment-Interface Vulnerability* (CIV) is an instance of the general confused deputy [234] problem, where a compartment is its own deputy, and fails to adequately vet the use of the interface it exposes to other compartments, as well as its usage of other compartments' interfaces.

Malicious compartments can leverage a CIV to mount data and control-based attacks, confusing a victim compartment into leaking and altering its private data and addresses, executing code, etc. Many CIVs arise due to incorrect or missing checks and sanitization of data flowing through the interface; our taxonomy provides a comprehensive list of causes. Type confusion [233], DUIs [244], and Iago [161] are all part of the CIV spectrum. For instance, Iago vulnerabilities are CIVs at the system-call boundary, and DUIs match DC1 and DC2 (§4.3.2).

4.3.1 Cross-Compartment Data Leakage (DL)

DL1: Exposure of Addresses. A victim compartment may leak memory addresses internal to a compartment, allowing an attacker, among others, to break Address Space Layout Randomization (ASLR) in the victim and locate critical objects. In Listing 3, address leaks may help libghostscript to know where to point `handle` to. DL1 can stem

Table 4.1: Compartmentalization mechanisms and frameworks that consider certain CIV classes. A ● indicates that a CIV class is fully addressed; a ◐ indicates a *partially* addressed CIV class; a ○ means fully vulnerable. An asterisk * indicates that the fix is not generic: the method makes assumptions about the source code being compartmentalized, or the use-case.

Mitigation Approach \ CIV Class	*DL1/Exp.Addr*	*DL2/Exp.Dat*	*DC1/Corr.Pt*	*DC2/Corr.Ind*	*DC3/Corr.Obj*	*TV1/API.Ord*	*TV2/Corr.Sync*	*TV3/Race*
TYPE-BASED CHECKS: RLBOX [344]	◐*	○	●*	◐	◐*	◐*	○	○
HARDWARE CAPABILITIES: CHERI [454]	○	○	●*	●*	◐*	○	○	○
UNDEFINED MEM. SANITIZERS [414]	◐	◐	○	○	○	○	○	○
BOUNDS-CHECKING TECHNIQUES [421]	○	○	○	●*	◐*	○	○	○
ASLR-GUARD [321], FG-ASLR [271]	◐	○	○	○	○	○	○	○
ANNOT. + DF-ANALYSIS: SOAAP [227]	○	◐	○	○	○	○	○	○
POINTER AUTHENTICATION [330, 312]	○	○	●*	○	○	○	○	○
API SEM. SANITIZATION: APISAN [470]	○	○	○	○	○	◐	○	○
TOCTTOU PROTECTION: MIDAS [138]	○	○	○	○	○	○	○	●*

from interface-crossing uninitialized data structures (or fields thereof) and compiler-added paddings [355], as well as data over-sharing between compartments. RLBox [344] proposes to address DL1 with pointer swizzling [458], ensuring that interface-crossing pointers can only address the sandbox. Generalized to arbitrary compartmentalization scenarios (e.g., by forcing interface-crossing pointers to address shared regions), this solves leaks due to oversharing, but does not address leaks due to uninitialized memory or padding. A near-complete protection can be achieved by combining RLBox with uninitialized-memory use detectors such as MemorySanitizer (MSan) [414]. RLBox is not generic, as it requires strong types which are not available in all languages (e.g., C), and non-trivial manual refactoring that forbids certain C/C++ idioms. More generic but weaker protection can be achieved with ASLR hardening techniques [321, 271].

DL2: Exposure of Compartment-Confidential Data. A victim compartment may leak compartment-confidential data to a malicious compartment. The impact depends on the nature of the leakage; typical targets include cryptographic secrets, or user data. Leaks stem from over-sharing, as well as uninitialized shared objects containing data from previous allocations. SOAAP [227] proposes to address DL2 with manually annotated sensitive data, leveraging data-flow analysis to guarantee that annotated objects never cross trust boundaries. This approach does not prevent leakages due to uninitialized memory/padding, and is notoriously prone to human error: past attempts at compartmentalizing OpenSSL failed because of misidentification of private data [379]. Similarly to DL1, more complete protection can be achieved by combining SOAAP with uninitialized memory use detectors.

4.3.2 Cross-Compartment Data Corruption (DC)

DC1: Dereference of Corrupted Pointer. A victim compartment may dereference a pointer corrupted by a malicious compartment. The impact spans that of all classical spatial vulnerabilities: malicious actors may gain read, write, or execute capabilities in the victim's context, or cause Denial of Service (DoS). In Listing 3, corrupted pointers `handle` and `message` grant the untrusted compartment full write permissions. DC1 vectors include shared objects, cross-compartment function call arguments and return values. RLBox [344] proposes to address DC1 with pointer swizzling [458], with the same limitations as in DL1. Hardware memory capabilities such as CHERI [454] address DC1 by making it impossible to forge pointers. Although promising, this technology is still

at a research prototype stage [120]. Its protection is not generic, requiring porting/annotations to fully address DC1, leaving certain C/C++ idioms unsupported. Generally, pointer authentication techniques such as ARM Pointer Authentication (PA) [312] can also address DC1, but may require porting in a compartmentalized context.

DC2: Usage of Corrupted Indexing Information. A victim compartment may use indexing information (size, offset, index) corrupted by a malicious compartment. The impact includes DoS, and that of buffer overflows, and underflows, depending on the context. Typical vectors are, similarly to DC1, shared data, cross-compartment function call arguments, and return values. RLBox [344] proposes to employ compiler-based techniques to force experts to write checks prior accessing interface-crossing data (and in particular indexing information). This ensures that humans will sanitize the API, but does not offer correctness guarantees. Hardware memory capabilities [454] address DC2 by offering bounds safety for C/C++, with the limitations mentioned in DC1. Generally, bounds-checking techniques [421] can address DC2.

DC3: Usage of Corrupted Object. A victim compartment may use an object corrupted by a malicious compartment. Examples include corrupted strings lacking `NULL` termination or including arbitrary format string parameters, corrupted OS/libc constructs such as `FILE*`, corrupted integers causing numeric errors [340], etc. Corrupted objects may be control or non-control data [164]. DC3 impact includes, in addition to DoS, information leaks, or exposing read, write, or execute primitives. Vectors are the same as DC1 and DC2. The fundamental difficulty to address DC3 is that the validity of an interface-crossing object is entirely dictated by the semantics of the API and of its users. The difficulty to extract this information systematically is a well-known problem [470]. RLBox [344] partially addresses DC3 with automatic validity checking and copy, for the types that allow it (e.g., strings), and forces manual sanitization for others, with the drawbacks mentioned for DC1. Even for types that allow automatic sanitization such as C-style strings, checks remains partial (checking `NULL`-termination, but not format string parameters). Hardware memory capabilities [454] address part of the *symptoms* of DC3 with full spatial memory safety, but cannot offer complete protection: not all control and data attacks that could be mounted on DC3 rely on spatial memory safety vulnerabilities.

4.3.3 Cross-Compartment Temporal Violations (TV)

TV1: Expectation of API Usage Ordering. A victim compartment may expose functions (or callbacks) to other compartments and assume call ordering, without enforcing it. For example, a compartment may expose two functions `init()` and `work()`, expecting `init()` → `work()`. Malicious compartments may call `work()` first. The immediate impact can be any form of undefined behavior in the victim, spatial or temporal depending on the context, such as DoS, uninitialized pointer dereferences, use-after-frees, synchronization bugs, etc. RLBox [344] proposes tooling to enforce callback ordering, but still requires manual detection and patching of TV1. This poses further difficulties when a compartment can be concurrently queried. This could be coupled with API semantics inference techniques such as APISan [470] to lighten the manual effort, but these techniques remain incomplete. Generally, there is a need for more research in identifying and enforcing compartment API usage ordering.

TV2: Usage of Corrupted Synchronization Primitive. A victim compartment may use corrupted synchronization primitives (e.g., mutexes, locks). The impact includes, beyond DoS (deadlock), that of any race-condition which could be leveraged to mount control-based attacks. TV2 vulnerabilities are a special case of DC3 where the corrupted object is a synchronization primitive. Existing frameworks do not offer solutions to this class of vulnerabilities: addressing TV2 is particularly challenging, as it requires redesigning the way distrusting compartments cooperate in multithreaded contexts.

TV3: Shared-Memory Time-of-Check-to-Time-of-Use. A victim compartment may check corrupted data in the shared memory. This may allow a malicious compartment to corrupt the value after the check and before the use, making the check useless. TV3 may lead to any previously mentioned CIV impact. TV3 vectors are shared objects with double fetches. Existing mitigations include forcing the copy of objects to a private region before checking, as done by RLBox [344] (with the genericity limitations mentioned in DL1), or forbidding concurrent modification altogether as done by Midas [138] (that targets only kernel-space).

4.3.4 Summary: CIV Protections are in their Infancy

No existing compartmentalization framework tackles all CIV classes. Techniques that can address a subset of CIVs only offer partial and/or non-generic solutions. Even the

most comprehensive system, RLBox [344], relies extensively on manual checking with no guarantees of check correctness. It is likely that combining all the techniques required to achieve state-of-the-art protection would result in an impractical performance overhead that contradicts the initial motivation of using lightweight isolation technologies for fine-grain compartmentalization. This observation motivates our study assessing the safety and complexity impact of CIVs.

4.4 ConfFuzz: Exploring CIVs with Fuzzing

4.4.1 Assumptions and Threat Model

We assume an application decomposed into compartments that are mutually distrusting. Compartments are defined as protected subsystems (as proposed by Lampson [292]), with private code, heap, and stack. Compartments communicate through interfaces. If a pointer is passed through an interface, the object it references is shared between the two compartments. A protection mechanism enforces spatial isolation: code cannot access private data or code from other compartments. The compartmentalization framework enforces cross-compartment control-flow integrity: one compartment can only call explicit entry points exposed by other compartments. These assumptions fit the vast majority of modern frameworks [462, 434, 236, 400, 318, 281, 359, 133, 395, 300, 299, 112].

We assume a completely compromised compartment which we refer to as the *malicious* compartment: an attacker can execute arbitrary code in its context. Compartments communicate through interfaces that respect the semantics of function calls, with variable degrees of sanitization. The malicious compartment attempts to misuse these interfaces to attack another compartment called the *victim*. The interface can be abused in either direction, according to the caller/callee role of the malicious/victim compartments:

Safebox. As the caller, the malicious compartment can abuse an interface *exposed by the victim* (callee). Vectors of corruption are function call arguments, data in shared memory, and return values of callbacks invoked by the victim to be executed in the context of the malicious compartment. This corresponds to a *safebox* scenario, in which a trusted subsystem (e.g., libssl) is protected from the rest of the system.

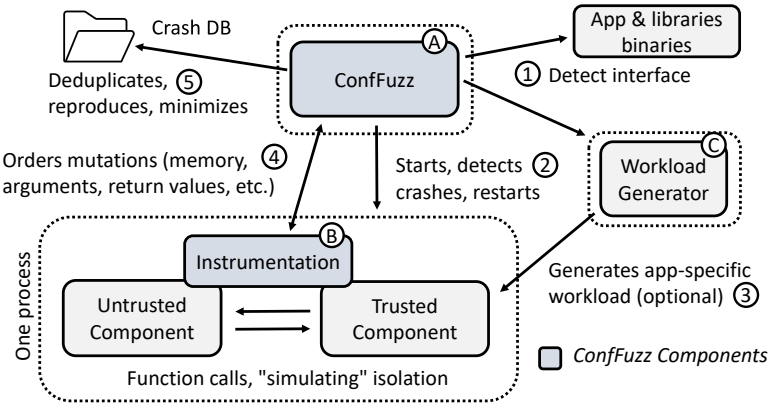


Figure 4.1: ConfFuzz architecture diagram.

Sandbox. As the callee, the malicious compartment can abuse an interface *invoked by the victim* (caller). Here, corruption vectors are return values, data in shared memory, and parameters of callbacks invoked by the malicious compartment to be executed in the victim's context. This is a *sandbox* scenario, in which an untrusted component (e.g., a 3rd-party library) is prevented from accessing the rest of the system.

4.4.2 Overview

To assess the impact of neglecting interface safety, we propose to fuzz monolithic (non-compartmentalized) software at possible arbitrary compartment boundaries, and analyze the set of CIVs we uncover. ConfFuzz is an *in-memory* fuzzer [418]: it instruments programs to hook into arbitrary interfaces, such as libraries, modules, functions, etc.

Because of their systematic nature, approaches based on static analysis may seem enticing to explore interface-related issues. However, related works leveraging such techniques fail to scale to more than simple programs [244]. Hence, we take a pragmatic approach and rely on fuzzing. Our goal is further different from existing in-memory/in-process/library/API fuzzers [404, 418] because our tool needs to fuzz both ways (safebox/sandbox). Hence, we develop ConfFuzz from scratch.

Two-Components Approach. Each run of ConfFuzz considers two communicating software components: a malicious, and a victim one. ConfFuzz simulates attacks towards the victim by automatically altering data crossing the interface between them; we call this *interface data altering*. For that, ConfFuzz hooks into the interface and fuzzes in both directions (sandbox/safebox), altering function call arguments, shared

data, and return values for direct interface calls and callbacks.

Architecture Overview. As shown in Figure 4.1, ConfFuzz is composed of a self-contained fuzzing monitor (Ⓐ), and Dynamic Binary Instrumentation (DBI, Ⓑ). This instrumentation, which sits between the malicious component and the victim in the application's process, leverages the Intel Pin [322] DBI framework. Using Pin lets us apply ConfFuzz to software with a low engineering cost, and hook at arbitrary interfaces unlike other approaches e.g., `LD_PRELOAD`.

First, ConfFuzz automatically identifies the interface between application and compartment components by analyzing debug information in the corresponding binaries (①). When starting the application (②), the fuzzing monitor dynamically injects instrumentation wrappers at the detected compartment interface. Following this, it optionally starts a workload generator (Ⓒ) to stimulate the application (③). At runtime, at each API call, the fuzzing logic in the monitor determines a set of alterations to perform, possibly through mutation of an existing set, and performs the alterations via the instrumentation (④).

The application runs with Address Sanitizer [403] (ASan) as bug detector. When ASan reports a crash in the victim, ConfFuzz deduplicates it based on the stack trace. If the bug is not known, ConfFuzz reproduces it, before minimizing the set of alterations performed to obtain the nucleus of alterations that trigger the bug (⑤). ConfFuzz is implemented in 4.5K LoC of C++, Bash and Python. The following subsections present ConfFuzz's fuzzing process steps in greater details.

4.4.3 Interface Detection and Instrumentation

ConfFuzz automatically handles the detection of the signature of a given target API. The tool gathers a list of API symbols: functions and callbacks used by the victim and malicious components to communicate, along with, for each of these, the number of arguments, the type and size of each argument, and the type and size of the return values if appropriate. We compile the target application with debug symbols, allowing ConfFuzz to use DWARF [195] metadata to retrieve interface and type information. The tool can automatically infer the list of functions composing the interface exposed by shared libraries, and the user can provide that list manually when targeting arbitrary interfaces. Most functions are instrumented when the application starts. Concerning callbacks, ConfFuzz automatically detects them at runtime by scanning API call parameters for function pointers, and instrumenting the identified functions on the fly. ConfFuzz

automatically infers what data is shared between the malicious and victim components, considering that all buffers referenced by pointers crossing the API are shared data.

Each symbol, including API elements and callbacks, is instrumented at entry and exit. At each of these events, the instrumentation checks for reentrance to protect against API calls performed by the compartmentalized component itself, notifies the fuzzing monitor with information about the function being executed and arguments/return value information, and allows the monitor to perform alterations: depending on the type of symbol and the fuzzing direction (safe/sandbox), altering argument values, return value, altering shared memory, etc. The instrumentation is kept as simple as possible and all the fuzzing logic runs in the monitor. Instrumentation and monitor communicate via a well-defined protocol using pipes.

4.4.4 Workload Generation and Coverage

ConfFuzz passively sits at internal API boundaries in an application, and users must determine an application-specific configuration and input workload that exercise the API. Finding a good set of inputs with high coverage is a problem shared across most fuzzers [377, 476]. In the data set considered in this paper, the time to understand the configuration system of an application and find an appropriate workload went from a few minutes to a few hours for a graduate student. We also explored the use of OSS-Fuzz [107] to generate workloads for the application, but found that hand-tuned workloads are generally better at precisely targeting the internal APIs that we intend to fuzz, which is critical for this study. Nevertheless, OSS-Fuzz should be considered in cases where the manual effort to create workloads should be minimized.

ConfFuzz is not coverage-guided. However, to provide an indication of how comprehensive the fuzzing of an API is with a given configuration and workload, ConfFuzz measures *API coverage*: the number of target API functions reached. This metric can be compared to the target API's size to understand the coverage of a given configuration and workload.

4.4.5 Interface Data Altering and Fuzzing Strategy

At each API crossing, the instrumentation notifies the monitor, which may proceed to alter interface data over the entire attack surface exposed to the malicious component. When fuzzing in sandbox mode, the malicious component may alter return values and callback arguments. In safebox mode, function call arguments and callback return values may be altered. In both cases, the malicious component may also alter data shared

Table 4.2: ConfFuzz data altering strategies for each CIV class. Each class of data alterations done by ConfFuzz is targeted at a particular type of CIV, as shown in this table.

CIV Class	Corresponding Data Alteration Strategy
DC1	*Alteration of pointer types* to invalid values (zero page, arbitrary unmapped areas).
DC2	*Alteration of integer types*: increments/decrements to trigger over/underflows, replacement to known limits such as `INT_MAX` (possibly at offsets) to trigger numeric errors, replacement with random values.
DC3	*Alteration of non-pointer types & of pointer targets*: decrements/increments at varying offsets in the object, replacement of bytes at various offsets in the object. *Replacement of pointers to the same type* (replay), and *Replacement of pointers to different types* (type confusion) to trigger more complex DC3 flaws.
TV1	*Non execution of API functions* for partial TV1 detection.
TV2	*Alteration of mutex/lock types* in shared memory (allows partial detection of TV2).
TV3, DL1, DL2	Not targeted.

between the malicious and victim components.

ConfFuzz probabilistically decides whether or not to alter data at an API crossing. In order to avoid revealing only shallow bugs that systematically crash in early fuzzing stages, we use a dynamic probability adaptation threshold: at first, the threshold is at 0, i.e., ConfFuzz alters data aggressively at all crossings. When the number of new crashes becomes scarce, ConfFuzz increments the threshold to find crashes further in the API usage. Concretely, based on a counter incremented at each API crossing, crossings that come before the threshold is reached see their data altered with a lower probability. This allows ConfFuzz to find stateful crashes as well.

ConfFuzz alters values in two ways: applying increments/decrements, and replacing the value altogether. ConfFuzz uses type information to drive alterations. For pointer values, ConfFuzz may perform replacements with other pointers of the same type, of different types, at varying offsets, at the zero page, on the heap, stack, data, text sections, etc. For integer values, ConfFuzz may perform replacements with known limits (e.g., `INT_MAX`). A detailed description of data alterations performed by ConfFuzz is provided in Table 4.2. While fuzzing, the fuzzer enriches an alteration corpus with values gathered during previous alterations. Values from the corpus are reused, possibly mutated, with a given probability.

4.4.6 Crash Processing and Bug Analysis

Crash Sanitization. Upon a crash, ConfFuzz compares the ASan stack trace with its database of known crashes. If the crash is a duplicate, no further analysis is performed, but information about the new occurrence is logged. ConfFuzz then checks whether

the new crash is a false positive. False positives arise when altered objects are sent back to the malicious component by the victim. In such cases the malicious component corrupts itself, yielding an invalid bug.

In order to detect false positives, ConfFuzz walks down the stack trace until it finds an entry referencing code belonging to a component. If that component is the one considered as malicious for this fuzzing run, the crash is considered a false positive. Even though such false positives are not valid crashes, ConfFuzz still attempts to minimize them, as this allows to detect non-viable data alterations that can be avoided later in order to minimize time wasted on false positives.

Reproduction and Minimization. After sanitization, ConfFuzz systematically attempts to reproduce crashes. Unfortunately, not all crashes are reproducible, as some might be due to particular non-deterministic factors such as scheduling effects, reliance by the application on random values/changing external inputs, etc. A non-reproducible crash cannot be further processed automatically by ConfFuzz. Still, information regarding such crashes are valuable for the analysis and are logged for manual inspection. On the other hand, if the crash can be reproduced, the monitor gradually minimizes it.

As part of the minimization step, ConfFuzz tries to understand the minimum set of alteration steps required to trigger a given bug. ConfFuzz gradually goes through each alteration performed in reverse order (since the last alterations performed tend to be the most likely to trigger the crash), and determines whether the alteration is sufficient to trigger the crash, necessary to trigger the crash (without it the crash cannot be reproduced, but it is not sufficient by itself), or superfluous. This results in a minimal set of attack primitives that the malicious component can perform to trigger the bug.

Impact Analysis of Crashes. After processing crashes, ConfFuzz performs initial triage. It harvests ASan-provided information: whether the crash is due to an illegal read, write or execution, allocator corruption, or to a `NULL` dereference, along with faulty addresses. Then, for R/W/X crashes, ConfFuzz tries to determine whether the vulnerability is arbitrary: for each alteration in the minimized steps, ConfFuzz mutates the altered value with increments/decrements, trying to reproduce the crash. If the crash is reproducible and the faulty address varies accordingly to the increment/decrement, the vulnerability is considered arbitrary.

Table 4.3: Bugs found for sandbox and safebox Trust Models (TM, fuzzing directions). *Refs.* links to studies that implemented an equivalent scenario. *Victims* gives the number of individual software components (application code, libraries, modules) that the malicious component managed to crash. *API Coverage* represents workload coverage: *callers* denotes the number of components calling the API, and *coverage* denotes how many functions of the fuzzed API are hit at runtime. *Impact* describes the type of bug: R/W/X fault, `NULL` dereference, or improper calls to the allocator (e.g., calling `malloc` with a negative value).

TM	Application	Compartment API	References	Crashes		Victims	API Coverage		Impact (of which arbitrary)				
				Raw	*Dedup.*		*Callers*	*Coverage*	*Read*	*Write*	*Exec*	*Alloc*	*Null*
Sandbox	HTTPd	libmarkdown	[344]	192	13	3	1	100% (4/4)	10 (8)	7 (7)	0 (0)	1	4
		mod_markdown		381	71	5	1	100% (1/1)	62 (52)	17 (14)	2 (1)	0	30
	aspell	libaspell		278	8	1	1	34% (48/141)	7 (7)	7 (7)	2 (1)	0	3
	bind9	libxml2 (write API)		0	0	0	1	86% (13/15)	0 (0)	0 (0)	0 (0)	0	0
	bzip2	libbz2	[462, 133]	16	5	1	1	62% (5/8)	5 (2)	1 (0)	0 (0)	0	0
	cURL	libnghttp2		61	7	2	1	50% (18/36)	3 (3)	5 (5)	0 (0)	1	3
	exif	libexif		400	7	1	1	10% (13/129)	3 (3)	0 (0)	0 (0)	0	5
	FFmpeg	libavcodec		316	20	3	4	31% (19/60)	13 (12)	12 (12)	0 (0)	3	7
		libavfilter		51	1	1	2	12% (2/16)	1 (1)	0 (0)	0 (0)	0	1
		libavformat		217	9	2	3	52% (10/19)	8 (7)	1 (1)	0 (0)	0	7
	file	libmagic		150	5	1	1	63% (7/11)	5 (2)	1 (1)	0 (0)	0	4
	git	libcurl	[252]	13	4	2	1	90% (18/20)	2 (2)	2 (2)	0 (0)	1	1
		libpcre		81	2	1	1	44% (8/18)	2 (2)	0 (0)	0 (0)	2	0
	Inkscape	libpng	[462]	66	3	1	1	46% (14/30)	2 (1)	2 (2)	0 (0)	0	1
		libpoppler	[227]	81	4	2	1	100% (9/9)	4 (3)	4 (4)	0 (0)	0	2
	libxml2-tests	libxml2 (write API)		0	0	0	1	100% (47/47)	0 (0)	0 (0)	0 (0)	0	0
	Lighttpd	mod_deflate		117	26	2	1	100% (6/6)	16 (11)	5 (0)	1 (1)	2	9
	Image Magick	libghostscript	[133]	67	14	2	1	100% (11/11)	4 (2)	1 (1)	0 (0)	3	9
		libpng	[462]	778	44	1	2	22% (17/77)	2 (2)	9 (9)	2 (0)	2	39
		libtiff	[462]	197	14	2	1	30% (13/43)	3 (3)	6 (6)	0 (0)	0	13
	Nginx	libpcre		144	10	1	1	93% (14/15)	8 (7)	3 (3)	0 (0)	6	2
		mod_geoip	[399]	276	25	2	1	35% (5/14)	21 (17)	4 (1)	1 (1)	1	10
	Okular	libmarkdown	[344]	64	5	3	1	100% (4/4)	3 (1)	0 (0)	0 (0)	1	2
		libpoppler	[227]	195	9	1	1	6% (24/379)	8 (6)	7 (7)	0 (0)	1	4
	Redis	mod_redisbloom		389	23	1	1	42% (8/19)	18 (13)	6 (4)	0 (0)	0	13
		mod_redisearch		381	21	1	1	54% (18/33)	15 (14)	14 (11)	0 (0)	0	12
	rsync	libpopt		167	8	1	1	90% (9/10)	4 (3)	2 (0)	0 (0)	0	5
	squid	libxml2		226	12	1	1	70% (7/10)	9 (5)	3 (3)	4 (1)	0	4
	su	libaudit		0	0	0	1	66% (2/3)	0 (0)	0 (0)	0 (0)	0	0
	Wireshark	libpcap		162	8	2	1	50% (20/40)	8 (3)	5 (5)	0 (0)	0	4
		libzlib		42	1	1	1	85% (6/7)	0 (0)	0 (0)	0 (0)	0	1
	Total:			5508	379	47	38	N/A	246 (192)	124 (105)	12 (5)	24	195
Safebox	cURL	libssl	[133]	198	27	1	1	25% (14/56)	18 (10)	5 (4)	1 (1)	0	17
	GPA	libgpgme		174	9	1	1	4% (3/72)	7 (2)	0 (0)	0 (0)	0	6
	GPG	libgcrypt	[133]	4221	105	1	1	15% (15/95)	64 (60)	4 (0)	0 (0)	77	20
	Memcached	internal_hashtable	[359]	4037	16	1	1	50% (6/12)	10 (5)	2 (0)	0 (0)	1	6
	Nginx	internal_libssl-keys	[359, 434, 225, 315]	599	46	1	1	50% (2/4)	32 (1)	28 (0)	0 (0)	0	22
		libssl	[133, 112, 252, 395]	346	39	2	1	11% (11/96)	16 (13)	19 (13)	2 (1)	0	26
	sudo	internal_auth-api		191	5	1	1	100% (5/5)	5 (4)	0 (0)	0 (0)	0	4
		libapparmor		97	3	1	1	100% (2/2)	2 (2)	2 (0)	0 (0)	0	2
	Total:			9863	250	9	8	N/A	154 (97)	60 (17)	3 (2)	78	103

4.5 A Large-Scale Study of Real-World CIVs

In this section, we use ConfFuzz to gather a large dataset of real-world CIVs, from which we extract insights on a set of research questions:

(Q1) What is the number of CIVs at legacy, unported APIs?

(Q2) What patterns lead to CIVs and are all APIs similarly affected by CIVs?

(Q3) What is the complexity to address these CIVs when compartmentalizing?

(Q4) What is the range of severity of the CIVs we uncover? In other words, without a fix, what can attackers do?

Table 4.3 shows our results. §4.5.1 and §4.5.2 give an overview of the methodology and results. The following sections provide in-depth analysis.

4.5.1 Methodology

Choice of Scenarios. Our corpus is composed of 25 applications and 36 library/module/function APIs, totaling 39 real-world sandbox and safebox scenarios. We choose scenarios that are meaningful security-wise, e.g., sandboxing image processing libraries because they are higher-risk, or safeboxing functions manipulating SSL keys because they are sensitive. Motivated by (Q2), we choose scenarios that encompass a diversity of API types (libraries, pluggable modules, internal APIs), and usages (image processing, text parsing, logging, key management, etc). We focus on highly popular applications. Several of these scenarios (16/39, see table) have been presented as use cases in 12 different compartmentalization frameworks, only 2 of which [344, 227] providing protection against certain CIV classes (discussed in §4.3). The smaller number of safebox scenarios reflects that, in general, libraries/modules tend to perform more untrusted operations that one might want to sandbox (e.g., request processing, complex data parsing) than trusted operations that one might want to safebox (e.g., cryptographic operations).

Comparison with Related Works. As discussed in §4.4.2, existing fuzzers are unfit by design to the investigation of CIVs, so we present no baseline. A relevant work in the domain of static analysis is DUI Detector [244], which could be used to detect classes DC1 and DC2 presented in §4.3.2. Unfortunately, its authors were unable to provide us with its source code. We therefore provide a textual comparison in §4.7. We evaluated the cost of DBI (Pin) in ConfFuzz on a representative set of applications (Nginx, Redis,

file, xmltest), and found it to be ≤65%, thanks to Pin's probe (JIT-less) instrumentation mode. These numbers match official data [187]. The overhead could be further reduced with compiler-inserted instrumentation, which we leave for future works.

4.5.2 Overview of the Data Set

Overall, ConfFuzz found 629 unique bugs deduplicated from 15,371 crashes. ConfFuzz uncovered bugs of 4 different classes as listed in §4.3: DC1-DC3, and TV2. In the *sandbox* mode, where ConfFuzz fuzzes from the (untrusted) component towards components calling it (e.g., application code, other libraries/modules), we found 379 CIVs. For 3 scenarios, ConfFuzz did not find any CIVs. We discuss these scenarios in §4.5.4. In the *safebox* mode, the fuzzer found 250 unique crashes. There are significant differences with the sandbox results impact distribution, which we analyze in §4.5.4. For both modes, the summed number of bugs of all impact types is larger than the total bug count (999 versus 629) because 224 bugs (1/3rd of the bugs) have more than one type of impact.

API End-Point Coverage. We observe unequal interface coverage across scenarios. Some scenarios such as HTTPd with libmarkdown show full API coverage, while others such as bzip2 with libbz2 feature poorer coverage. Generally, low coverage is due to ConfFuzz fuzzing a single software configuration or CLI option. In the case of bzip2 for example, ConfFuzz fuzzes decompression (`-d`), but not compression or archive testing (`-z`/`-t`). These entry points of libbz2 are left unexplored, and the user is notified. This is a common problem across fuzzers, addressable by varying configurations [377, 476], and fuzzing with more workloads, which we consider out of scope. Nevertheless, despite its simple exploration approach, ConfFuzz shows good ability to find relevant bugs. For example, in HTTPd with libmarkdown, a scenario from RLBox [344], ConfFuzz correctly discovered all CIVs addressed by RLBox, asserted the criticality of the bugs, and even one more due to library version differences.

4.5.3 Prevalence of CIVs

At the highest level, the results confirm our expectations: *CIVs are widespread among unmodified applications.* Indeed, all but 3 scenarios present CIVs. Looking into greater details, however, disparities appear: libraries present widely varying CIV numbers, ranging 0-105 for a single scenario. Disparities become even clearer when looking at the ratio of vulnerable over covered API elements, ranging 0%-100%, as shown in Figure 4.2. This observation is not a consequence of coverage disparities; we find that the number

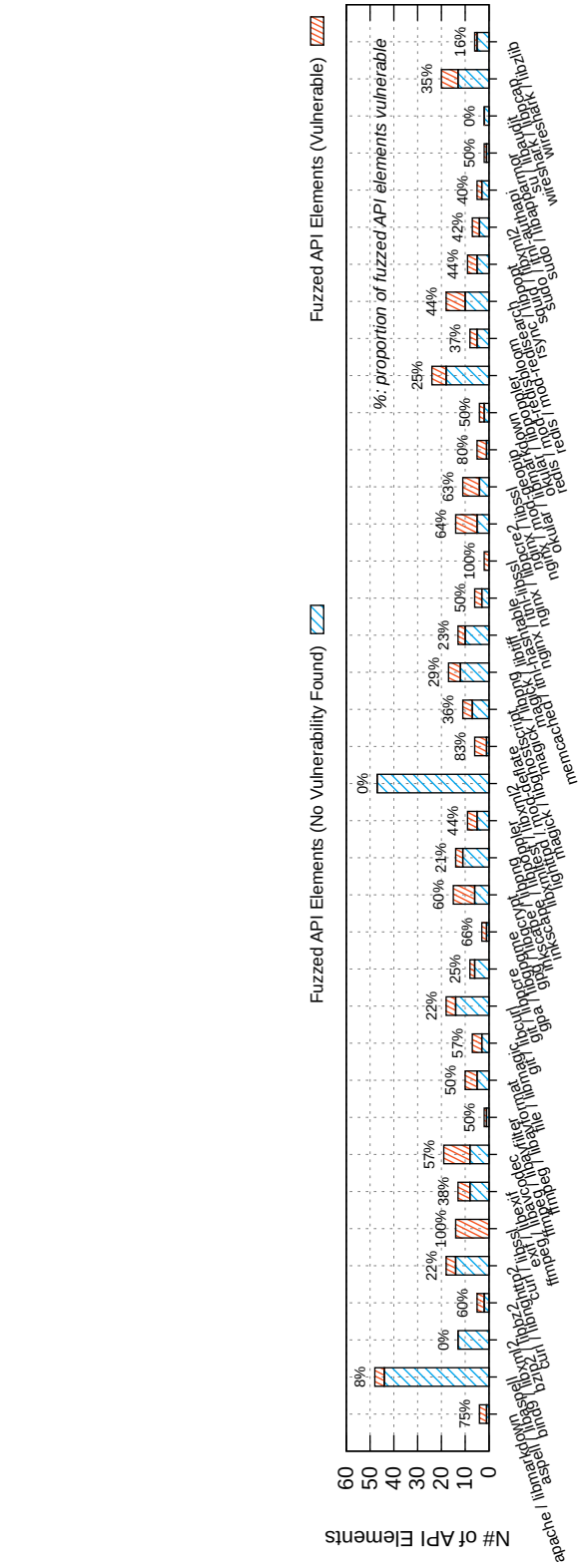


Figure 4.2: Proportion of covered vulnerable endpoints versus covered endpoints for each scenario (see Table 4.3 for coverage).

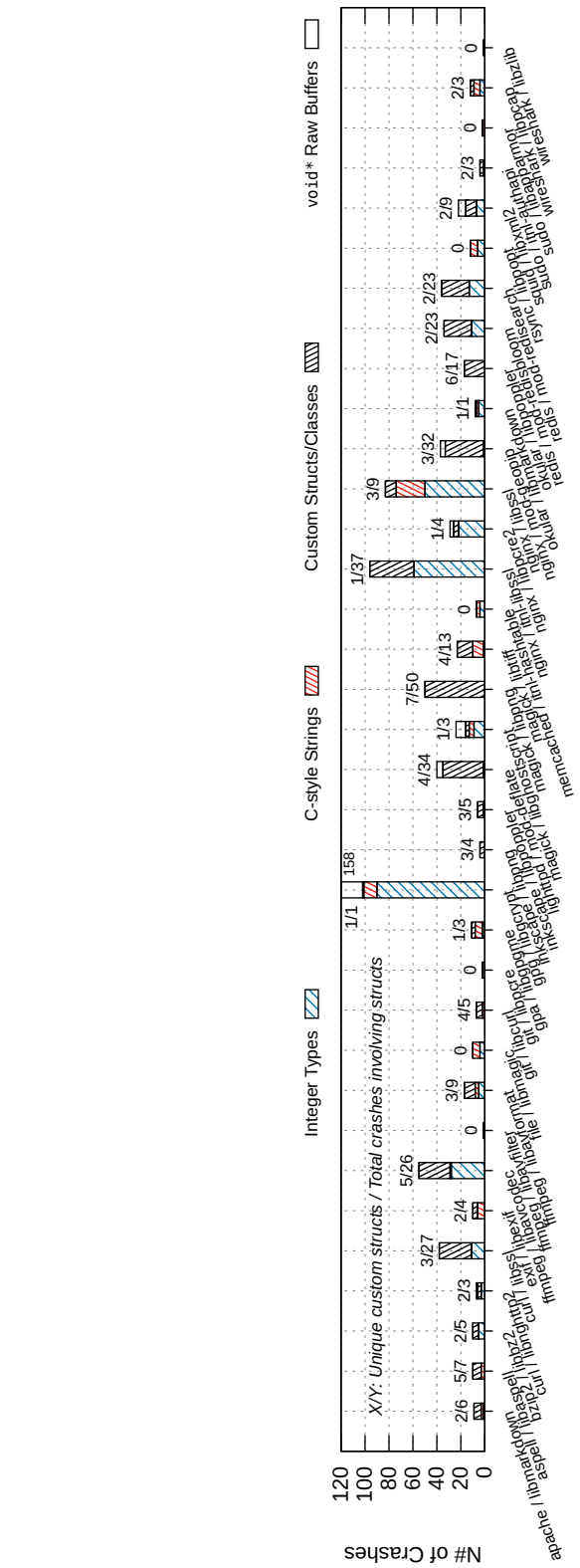


Figure 4.3: High-level type classes involved in CIVs for each scenario.

Table 4.4: Low-level CIV patterns from API-crossing types.

Type Class	Low-Level CIV Patterns Involving this Type Class
Integer Types	sizes and bounds (DC2), error codes and return values (DC3, Pattern 4), and generally control-flow manipulations (DC3)
Custom Structs	state data (DC3, Pattern 1), mutexes (TV2, Pattern 1), non-control data (DC3)
C-style Strings	version/error strings (Pattern 4), serialized data (all DC1 and DC3)
void buffers*	exposed allocator arguments (DC1, DC2, DC3, Pattern 5), and generally opaque data structures (DC1 and DC3, rare in this dataset)

of functions covered and the number of CIVs found are uncorrelated ($|r| < 0.09$). Similarly, there is no correlation between the size of compartment APIs and the number of CIVs found ($|r| < 0.1$). This observation further materializes when considering APIs that do comparable tasks, e.g., libbz2 and libzlib (compression), or libpng and libtiff in ImageMagick (graphics). In both cases, CIV counts vary by 3-5x for nearly identical coverage. This is most startling in ImageMagick where the workload (image format conversion) is identical in both cases, but the number of CIVs jumps from 14 to 44. These observations hint that the vulnerability of interfaces to CIVs is a factor of *individual patterns and structural properties* rather than of size or functionalities. We study these patterns in the next section.

Insight: Smaller APIs do not imply less CIVs in unmodified applications: API size and number of CIVs are uncorrelated.

4.5.4 Patterns Leading to the Presence or Absence of CIVs

Zooming in, the types/data structures triggering crashes are integer types, custom classes and structures, C-style strings, and raw `void*` buffers flowing through compartment interfaces. As shown in Figure 4.3, integer types and custom objects are the main culprits, being involved in 85% of CIVs, while only 33% of CIVs are due to C-style strings and raw buffers. Table 4.4 summarizes the low-level CIV patterns observed for each type. The following paragraphs give insights from a selection of these patterns and higher-level observations.

Pattern 1 – Modularity & Exposure of Internal State. Module APIs are some of the most vulnerable interfaces in the study; HTTPd, Nginx, Redis, and Lighttpd modules rank 2nd-10th in CIV count, clearly above average. HTTPd, particularly, ranks 2nd with 71 unique CIVs, even though its module API has a single function and entry point. Impact-wise, modules represent more than half of read CIVs, and 1/3rd of write

```
// A structure to represent the current request, shared with modules
struct request_rec {
    apr_pool_t *pool; // request's memory pool
    conn_rec *conn; // connection with the client
    server_rec *server; // request's virtual host
    request_rec *main; // main request pointer
    apr_thread_mutex_t *inv_mtx; // callback mutex
} // ... abbreviated, 75 fields in total ...
```

Listing 4: HTTPd structure shared with modules, exposing critical internals.

and execute CIVs, even though they only represent 5/39 of our scenarios. In short, the most modular interfaces of the dataset get *more bugs* and *worse bugs* on average.

We track down these observations to one common pattern across module APIs: *exposing the application's internals to maximize performance and flexibility*. Unlike library APIs, module APIs are designed to accommodate *generic needs with good performance*. We observe that these needs stand at odds with the requirements of compartmentalization: in order to achieve this, the application must expose its internals to the module, resulting in significantly less encapsulation than with traditional libraries. Take Apache modules as an example: HTTPd modules can register hooks into core operations of the server (e.g., configuration, name translation, request processing), and are systematically passed a request structure `request_rec`, visible in Section 4.5.4. This structure is highly complex, with over 75 fields, of which over 60% of pointers, many referencing other complex structures: memory pools, connection data, server data, and even synchronization structures (resulting in TV2 CIVs). Such shared structures are not only very hard to sanitize, they lead to oversharing when compartmentalizing because most modules do not need access to all fields.

These observations become even more clear comparing the two Apache scenarios: mod_markdown and libmarkdown. Here, the HTTPd Markdown module uses the Markdown library under the hood, thus we can isolate at the module boundary (mod_markdown), or at the library boundary (libmarkdown) to obtain comparable security properties. As expected, isolating at the library boundary yields fewer crashes than isolating at the module boundary (5.5x less crashes). Despite a larger number of entry points, the attack surface exposed by HTTPd to the library is far smaller than that of the module API due to encapsulation; the library does not have knowledge or access to HTTPd's internals. We make similar observations for Nginx, Redis, and Lighttpd.

These observations are not obvious: multiple studies propose privilege separation of modules [252, 399] without considering this problem, and other studies [308, 395,

299] build on the assumption of modularity being fundamentally good for compartmentalization. We stress that this is not always the case: the goals of being able to develop and evolve software parts independently to reduce engineering costs as sought by modularity [362] do not necessarily align with that of minimizing the privileges assigned to software parts as sought by compartmentalization. Overall, more research is needed to achieve fast, generic, and compartmentalizable modular APIs.

> **Insight:** Modularity ≠ Low compartmentalization complexity: boundaries exposing internal state are hard to protect.

Pattern 2 – CIV-Resilient Input-Only APIs. Several scenarios are entirely CIV-free: libxml2 write API, and su with libaudit. In the case of libxml2, even API tests with full coverage of all 47 endpoints do not yield a single CIV. We manually confirmed that no CIVs are possible at these APIs, reducing these observations to one common pattern: *designing the API like a write/input-only endpoint.* For instance, Bind9 uses libxml2 to store statistics onto the filesystem in XML format; it strictly forwards data to libxml2, which formats and stores it. The situation is similar in su with libaudit, used as a logging facility. su passes logs to libaudit, which processes them. There is no feedback loop and no data flow from the library to the application, and thus no CIVs. Note that this does not apply to squid, which uses a different API of the libxml2 family. While this restrictive pattern cannot suit generic sandboxed API needs, it shows that there are naturally robust APIs for privilege separation, and remains applicable to several other scenarios such as (de-)compression or image processing. We expand more on CIV-resilient APIs in §4.6.

> **Insight:** "Input-only" APIs are CIV-resilient by design, and can be found is several scenarios.

Pattern 3 – Corrupted Data Forwarding. In sandbox scenarios, the number of victim components (individual application/libraries/modules that the untrusted component managed to crash), as shown in Table 4.3, is >1 for 1/3rd of the cases. This is surprising, since only 4/39 scenarios have more than one caller component — 3 of them being FFmpeg. We tracked down these observations to one common pattern, where a trusted component T_1 receives corrupted input from an untrusted component U_1, and forwards it to another trusted component T_2. Component T_2 thus receives corrupted data from *trusted* component T_1. Take the example of curl, which features this CIV

```
ssize_t send_callback(nghttp2_session *h2,
    uint8_t *mem, size_t length, ...) {
  /* (...), send_underlying = libssl callback */
  written = ((Curl_send*)c->send_underlying)(data,
      FIRSTSOCKET, mem, length, &result);
} /* (...) simplified */
```

Listing 5: Fall-through CIV in libssl, with curl and libnghttpd2 isolated: curl transparently forwards corrupted data to libssl.

pattern in Listing 5. Here, curl's callback `send_callback` (T_1) receives corrupted input `mem` and `length` from libnghttpd2 (U_1), and transmits it to libssl (T_2) via `send_underlying`, resulting in a libssl crash. The untrusted HTTP parsing library thus manages to attack libssl even though the two libraries do not directly communicate. Intuitively, this pattern motivates to check as early as possible to avoid unsanitized data spreading. Unfortunately it is not always possible to perform checks at the untrusted boundary: T_1 is the recipient of the data and may not have knowledge of the semantics of the corrupted object. Thus, checks must be thought with the full system in mind: *it is not because a library only interfaces with safe or trusted components that it won't receive untrusted inputs.* This poses the question of "when to check, and when not to", to ensure that no check is missed, while limiting the amount of unnecessary vetting.

> **Insight:** The attack surface of a component exceeds interactions with directly reachable untrusted components; targeted attacks to non-adjacent components are realistic and checks must be thought with the full system in mind.

Pattern 4 – Error-Path CIVs. About 42% of the dataset's crashes require more than one alteration to manifest. In these cases, part of the alterations aim at diverting the control flow to reach a vulnerable location. We manually studied these multistep vulnerabilities and found that, in sandbox scenarios, many present a common pattern where the vulnerability is located in an error-handling path. In this case, ConfFuzz diverts the control flow to the error path (e.g., via a non-zero return value), so that the application reaches a location that makes use of another corrupted value. Take the example of Nginx with the GeoIP module isolated, as depicted in Listing 6. Nginx calls the module's post-configuration callback, passing it a pointer to its shared configuration object `cf`. Assume GeoIP corrupts `cf`. If the call succeeds (default behavior in GeoIP), Nginx proceeds with other operations making use of the configuration object,

```
static char * ngx_http_block(ngx_conf_t *cf, ...) {
  /* (...) simplified */
  if (module->postconfiguration(cf) != NGX_OK)
    /* cf used in caller after return: */
    return NGX_CONF_ERROR; ①

  if (ngx_http_variables_init_vars(cf) != NGX_OK) ②
    return NGX_CONF_ERROR;
} /* (...) many operations involving cf here */
```

Listing 6: Multi-alteration vulnerability with Nginx modules.

particularly at ②, triggering a crash. Unfortunately, crashes at this stage hide potential for corruption beyond ② or in the caller. Exploiting error-path CIVs, ConfFuzz found that by corrupting `cf` and returning an error value from `postconfiguration`, further crashes could be uncovered in `ngx_http_block`'s caller, after the return statement ①, avoiding ②. Beyond showing that ConfFuzz can easily uncover multistep crashes, this strengthens Pattern 3's observations: checks should be done as early as possible, ideally before any control flow operation. CIV checks should preempt other functional checks, e.g., error code handling, to avoid multiplying the size of the attack surface.

Pattern 5 – Allocator Exposure. The dataset features 102 allocator corruption bugs affecting a total of 14 scenarios, with 77 bugs coming from GPG with libgcrypt. We investigated this CIV class and reduced it to two related patterns, where (1) corrupted data flows reach parameters of allocators calls, or (2) trusted components expose untrusted components with a direct window to their allocator. In the first case we observe numerous DC1-corrupted pointers reaching `free()`, DC2-corrupted integers reaching `malloc()`, as well as cases within the libc due to DC3-corrupted `FILE*` pointers. The second case appears in GPG with libgcrypt, and almost systematically in sandbox scenarios with module APIs (HTTPd, Nginx). Here, the application presents the sandboxed component with a malloc-like API that allows it to allocate objects with the application's memory allocator. This results in numerous DC1-DC2 CIVs and explains the peak in `void*` related bugs for GPG in Figure 4.3. Allocator exposure is achieved with the intention to achieve higher performance (custom memory allocator), for introspection reasons (statistics, or record log information), or for correctness reasons (allocation/freeing is spread over both components). Vulnerabilities arising from this pattern are high impact: they allow malicious compartments to trigger arbitrary use-after-frees, heap Feng Shui [450], and other allocator exploitation techniques. Since this

pattern also defeats compartment heap separation, it is likely that DL2 CIVs will arise too. In all cases, allocator exposure is very difficult to get right in compartmentalized contexts, and should be avoided where possible. Removing allocator exposure can be straightforward if performed for performance reasons only, but may require significant redesign of the API if done for introspection or correctness reasons.

> **Insight:** Cross-compartment memory management is hard and leads to exploitable CIVs. Fixes may imply API redesign.

4.5.5 Security Impact of CIVs

At the highest level, our impact analysis confirms that CIVs are particularly critical from a security standpoint. We find that more than 75% of scenarios studied present at least one write vulnerability, and all safebox scenarios present at least one arbitrary read, which defeats attempts to protect secrets. We present an overview of impact results from Table 4.3, before focusing in depth on a selection of bugs from the data set.

As visible in Table 4.3, the distribution of impact is clearly in favor of read vulnerabilities, `NULL` dereferences, followed by write, allocator corruption, and code execution. This follows the distribution of typical patterns in applications. Most commonly, a victim component reads data referenced by a pointer provided by a malicious component (resulting in read vulnerabilities), or use a corrupted integer within a check (e.g., a success code) before accessing internal data (resulting in `NULL` dereferences if the application reads unallocated/uninitialized data with the call returning a success code). Less commonly, a victim write to a pointer provided by a malicious component (write impact). Memory allocator corruption bugs most commonly happen via Pattern 5 of §4.5.4, or when size parameters flow from an interface to an allocation site. The least common impact is execute, typically resulting from the victim executing a callback passed by a malicious component.

We find that more than 70% of read and 66% of write vulnerabilities are arbitrary, as well as half of execute CIVs. Thus, in the absence of countermeasures, if a subverted component can perform illegal R/W/X operations outside its compartment through APIs, it is likely to be able to do so at any address. Further, even though the proportion of execute impact is low (8/39 scenarios), it is probable that attackers will be able to mount attacks with arbitrary R/W CIVs to reach code execution. Next, we illustrate these observations with an analysis of concrete bugs from the dataset.

```
// CIV 1: option setting API leads to arbitrary R/W
ulong SSL_CTX_set_options(SSL_CTX *ctx, ulong op) {
  return ctx->options |= op;
}

// CIV 2: cross-API object SSL_CTX with function
// pointers leads to arbitrary execution
SSL *SSL_new(SSL_CTX *ctx) {
  /* ... */
  s->method = ctx->method;
  /* ... */
  if (!s->method->ssl_new(s)) // arbitrary execution
    goto err;
} /* ... */
```

Listing 7: Two libssl CIVs leading to arbitrary read, write, and execute impact. Both functions are exposed to the application as part of I_{ext}.

```
void aesni_ecb_encrypt(const uchar *in, uchar *out,
  size_t length, const AES_KEY *key, int enc);
```

Listing 8: Prototype of internal encryption API from I_{int}, vulnerable to key extraction CIVs. The body of the function is automatically generated in assembly, we thus omit it.

4.5.5.1 Case Study: OpenSSL Key Extraction

OpenSSL is a popular compartmentalization target, being both high-risk (highly complex codebase shipped in network-facing applications) and sensitive (holding secrets). We count at least 8 studies safeboxing it [225, 395, 227, 133, 112, 252, 315, 434] with attempts going beyond that of academic research [379]. Safeboxing approaches typically either compartmentalize OpenSSL in full and isolate at the external libssl API (I_{ext}), or compartmentalize at the libcrypto internal API of key-interacting primitives (I_{int}). I_{int} is generally viewed as more robust because of the reduced TCB and the ability to tackle bugs within libssl, such as Heartbleed [315].

We applied ConfFuzz to both interfaces. *In all cases, we are able to extract keys out of the safebox leveraging a single CIV uncovered by our fuzzer*; we present three of them.

Listing 7 illustrates two CIVs found for I_{ext}, where libssl is safeboxed as a whole. The first CIV affects libssl's primitives to set and get SSL options, part of the official libssl API. Here, an untrusted caller compartment can control `ctx` and `op`, enabling for arbitrary read and write operations via the bitmask. In the second CIV, libssl executes

```
int sudo_passwd_verify(struct passwd *pw, char *pass,
  sudo_auth *auth, struct sudo_conv_callback *cb) {
  /* ... abbreviated ... */
  sav = pass[8]; // read CIV
  pass[8] = '\0'; // write CIV
} /* ... abbreviated ... */
```

Listing 9: CIV and CVE-2022-43995 found by ConfFuzz in the sudo password backend, manifesting when passed a password with length below 8 Bytes.

callbacks provided by an untrusted object of type `SSL_CTX`, one of the three key vulnerable structures highlighted by Figure 4.3. These callbacks are very common across the libssl API and result in arbitrary code execution. Both vulnerabilities can easily be leveraged to extract safeboxed keys.

Listing 8 illustrates a CIV found for I_{int}, where keys are isolated at the internal interface. In this encryption primitive, callers control the `in` and `out` pointers, along with the key location `key`. By pointing `in` to the key and `key` to a known value, attackers can either cryptanalyze the key out of `out`, or use the decryption function to extract the key.

There are systematic problems that make robust safeboxing at the libssl API difficult. This large API makes heavy use of state structs such as `SSL*` or `SSL_CTX*`. Sanitizing such structures is hard; it is likely that, even provided counter-measures from §4.3, the result will approach that of a rewrite of OpenSSL. Safeboxing at I_{int} is less complex but still requires redesign: encryption and decryption primitives must be made stateful to store the location of valid keys, checking that input and output buffers do not overlap key locations. Key creation and loading must also be carefully validated.

> **Insight:** CIVs make it simple to extract SSL keys from an unmodified API safebox. Robust SSL key safeboxing requires redesign of the key API into a stateful entity.

4.5.5.2 Case Study: Sudo, Impact Beyond CIVs

The sudo [88] utility is a strong target for compartmentalization, as it is both high-risk (>100K LoC, many features) and sensitive (exploits lead to system privilege escalation). We considered several scenarios, one of them safeboxing the authentication API, which

manages password verification. Here, ConfFuzz found 5 CIVs, with read and `NULL` dereference impact. Investigating them, we realized that one CIV, shown in Listing 9, actually features R/W impact, and is reachable from user external input, i.e., it is also a vulnerability in non-compartmentalized contexts. This decade old issue manifests when users enter small passwords and was assigned CVE-2022-43995 [19] after we reported it.

4.5.5.3 Case Study: Nginx Master/Worker Interface CIV

We studied the applicability of ConfFuzz to other compartmentalization models such as the Nginx master/worker manual separation. Here we assume that a worker has been compromised (e.g., from the network), and attempts to escalate to master privilege level. In this model we found a decade-old CIV that allows a worker to trigger memory corruption in the master[1]. The vulnerability affects a reliability feature of Nginx: when a worker crashes, the master forcibly unlocks shared memory mutexes hold by the worker to prevent deadlock. A malicious worker may corrupt the mutex before crashing itself to force the master to dereference a crafted pointer. This particular CIV is low impact due to control constraints in the mutex unlocking routine – bytes will only be overwritten if they match the worker's PID. Nevertheless, CIVs at such interfaces present a real risk: less constrained bugs are realistic and may pose a real privilege escalation threat.

> **Insight:** CIVs also affect production-grade software and may be leveraged to mount privilege escalation attacks.

4.5.6 Conclusions

We stressed that CIVs widely affect unmodified software, but in varying proportions (Q1). Factors are structural; we elaborated on them with 5 central patterns and insights (Q2). We illustrated that API redesign will be necessary in many cases to achieve robust least-privilege enforcement (Q3). Finally, we showed that CIVs are impactful, exploitable, and elaborated with case studies on popular compartmentalization targets (Q4). Drawing from this, we discuss how to design interfaces that are by conception more CIV-resilient in §4.6.

[1] https://github.com/conffuzz/conffuzz-ndss-data/blob/main/docs/nginx.md

4.6 (Re-) Designing Interfaces for Distrust

We showed that interfaces are not equally affected by CIVs because of interface design patterns. Next, based on previous sections, we discuss interface patterns that *reduce* compartmentalization complexity, and how to leverage them to design strong compartment boundaries, or refactor existing ones. These patterns do not eliminate the need for CIV countermeasures as detailed in §4.3; in their absence, these patterns reduce the number of CIVs, and in the presence of countermeasures, these patterns help palliate their limitations. When refactoring, many of the items listed below require major software redesign. We believe it is a necessary price to pay to obtain firm safety guarantees from compartmentalization.

4.6.1 Resources (Memory, Handles) Must Be Clearly Segregated

Memory ownership must be clearly defined, with each component responsible for allocating and freeing memory in their region: components must not rely on an another component's memory allocator (see Pattern 5, §4.5.4). Similarly, system resource handles (or handles to any third-party-managed resources) must not be shared. Take the example of `FILE*`: when shared, it is hard to determine who should release the handle and when, requiring complex, ad-hoc, and error-prone virtualization [133, 252, 344]. Instead, components should acquire and release their own handle: e.g., for `FILE*`, components should exchange file paths and call `open()` on their own.

4.6.2 Copy API-crossing Objects

Shared objects should be systematically copied to prevent TV3 CIVs: it is hard to safely use objects that can be concurrently modified by untrusted compartments. More generally, concurrent use of objects across compartment boundaries should be avoided, as it requires cross-compartment synchronization, which in turn opens for TV2 CIVs.

4.6.3 Simplify API-crossing Objects

Compartment interfaces must not expose data that cannot be safely checked. This includes state information, which is hard to protect in the general case (Pattern 1, §4.5.4). In such cases, a layer of indirection can be added so that the object is not accessed directly, but through a set of primitives that can assert the safety of individual operations. If this is not possible, the interface is probably not a good compartmentalization boundary in the first place.

4.6.4 Trusted-Components Allocates

When a trusted component is passed a pointer to a buffer allocated by another component, it needs to either trust that component, or verify the pointer (which only privileged monitors can perform as it requires knowledge of the memory layout). Take the example of C-style strings: if a sandboxed callee allocates and returns through an API a string pointer, a trusted caller needs to verify the pointer's validity and the `NULL`-termination of the string. This problem can be eliminated by applying a *trusted-component allocates* policy, i.e., *caller-allocates* in sandbox scenarios, and *callee-allocates* in safebox scenarios. If the trusted component allocated the string buffer, it knows the maximum size of the string and can safely check for `NULL`-termination. In the case of mutual distrust, the involvement of a privileged monitor is necessary.

4.6.5 Trusted Interface Functions Must Be Thread-Safe

When a trusted compartment C_t exposes a function f_t (API function for safeboxes, callbacks for sandboxes) to an untrusted compartment C_u, C_t enables C_u to interfere with its control flow at any time. For example, C_u may interleave multiple calls to f_t and other API functions to trigger TV1 CIVs in C_t. Even when calls to C_t functions are serialized, these may perform callbacks back to C_u that will allow it to interleave other calls to C_t (a behavior that we observed with ImageMagick and libpng). Thus, trusted compartment functions must be designed *thread-safe* to support any concurrent calling. Alternatively, trusted interface calls should be strictly serialized and run to completion (no callbacks), a rather restrictive model.

4.6.6 Trusted Interface Functions Must Enforce Ordering Requirements

Similarly, if trusted interface elements $f_1 \cdots f_n$ have ordering requirements, then these must be clearly stated and enforced to further tackle TV1 CIVs. This may require safeboxed libraries to become stateful, where they previously relied on invoking undefined behavior if the caller did not respect ordering. When asynchronous behavior is suitable, event-loop-based designs may allow interface designers to shift as much control-flow leverage as possible out of the hand of attacker by processing the core of the callback in the main loop, in a way that is consistent with other external inputs (similarly to signal processing).

4.6.7 No Sharing of Uninitialized Data

API-crossing uninitialized data must be systematically zero-ed to avoid DL1-DL2 CIVs (§4.3.1). Even sharing of properly checked objects can be unsafe if they have not been zero-ed at initialization, since compiler-added padding might remain uninitialized. Where applicable, zeroing should be compiler-enforced.

4.6.8 CIV Checks First

As soon as one allows untrusted data to propagate unchecked through a compartment, it becomes hard to ensure that all checks are properly performed down the line (Pattern 4, §4.5.4), and encourages duplication of non-trivial checks, maximizing the likelihood of errors in present and future versions of the software. Worse, untrusted data might not even be used within but simply flow through a compartment to be used in another one, which might have variable trust assumptions on the compartment feeding it data (Pattern 3, §4.5.4). Copy and checks should therefore be performed on the data as soon as possible after crossing the API, and preempt all other functional checks.

4.7 Related Works

Finding API Vulnerabilities. DUI detector [244] leverages static binary analysis, symbolic execution and dynamic taint analysis to detect pointer dereferences made by a security domain under the influence of another through an interface. Due to performance and scalability issues (emulation and symbolic execution), such approaches are hard to scale to large programs, large interfaces, large numbers of programs, or, as is the case here, high bug counts. Further, compiler-based static approaches are unsuited to scenarios such as OpenSSL, where safeboxed components are implemented in pure assembly files. Other studies such as Van Bulck et al. [436], focusing on Trusted Execution Environment (TEE) runtimes, take a manual approach to identify interface vulnerabilities. Being manual, such approaches are limited in scope. In-memory fuzzing allows us to be faster and more scalable than static and manual approaches, enabling for a larger-scale CIV study. Fuzzing yields a subset of all CIVs present at an interface, which is a suitable limitation in our case.

Classical system call fuzzing [443] searches for kernel vulnerabilities at the system call API. This corresponds to *one specific safebox scenario* where the compartment API is the system call API. ConfFuzz is much more general, targeting arbitrary sandbox/safebox scenarios at arbitrary APIs. Similarly to system call fuzzing, Emilia [179]

fuzzes for Iago [161] vulnerabilities, hooking at system calls and altering their return values to simulate a malicious kernel, corresponding to a sandbox model. Here too, ConfFuzz is much more general as it 1) can hook into arbitrary interfaces; 2) supports bidirectional fuzzing (sandbox/safebox); and 3) fuzzes the *full compartment attack surface* (callbacks, return values, shared data, function arguments) – whereas Emilia only fuzzes return values.

Classical network fuzzing [366] too bears similarities with fuzzing for API vulnerabilities in compartmentalized software. However, classical network fuzzers do not capture the specificities of compartmentalized software and have, to the exception of Nyx-Net [401], never been applied to this aim. Nyx-Net is a classical network fuzzer which has been applied to fuzzing the Firefox IPC layer. Unlike ConfFuzz, Nyx-Net targets exclusively classical process-based compartmentalization, and requires custom hooks in the libc network API. ConfFuzz is more generic, in that it can plug at any intra-program API, automatically (without custom hooks). Still, Nyx-Net's snapshot-based approach could be applied to ConfFuzz as a performance optimization, which is outside of the exploratory goal of this study.

Finding API Misuses. APISan [470] studies existing software to infer semantic usage information for a given API (e.g., semantic relation on arguments/functions). Using that information, it searches for deviations to detect possible API misuses at the source code level. Such an approach is not suited to detect CIVs. First, CIVs can be present even when the semantics of an API are respected in the code: at runtime, a malicious compartment with code execution abilities can manipulate the program's execution in a way that does not respect the API semantics. Second, even if API semantics could be fully enforced at runtime, most CIVs would remain undetected because the semantics of unmodified APIs are generally unsuited to distrust, as we show in the paper. Nevertheless, as we highlight in §4.3, APISan's ability to infer API semantics may be leveraged to determine enforcement policies, provided a large enough set of API usage samples (usually available for popular APIs).

In-Memory Fuzzing. Unlike conventional fuzzing approaches that inject malformed data through a program's input channels (e.g., network), in-memory fuzzing [418] moves the fuzzer within the target using process instrumentation techniques. ConfFuzz is an in-memory fuzzer specialized for CIV fuzzing. ConfFuzz mainly differs from existing in-memory fuzzers [418, 404, 326] in that it 1) fuzzes in both ways (sandbox and safebox) – whereas existing in-memory fuzzers mostly correspond to safebox fuzzing,

and 2) targets a different attack surface, the *compartment* attack surface – which, unlike usual in-memory fuzzers, includes callbacks, return values, etc. To our knowledge, we are the first to use in-memory fuzzing to study CIVs in unmodified software.

Interface-Aware Compartmentalization. Compartmentalization frameworks provide a variable degree of support for protecting security domain interfaces. The vast majority of modern compartmentalization frameworks [462, 434, 236, 400, 318, 281, 359, 133, 395, 300, 299, 112] do not achieve more than basic ABI-level interface sanitization at security domain crossing, such as switching the stack and clearing registers. Combined with the fact that most also rely on relatively coarse-grain shared memory-based communication for performance reasons, this opens up a wide range of CIVs and was one of our motivations to develop ConfFuzz.

RLBox [344] is a sandboxing framework for untrusted C++ software components. RLBox sanitizes sandbox data flow in a partially automated way: using static analysis and C++ type information, the framework can add certain checks automatically. When not possible, RLBox outputs compiler errors to require human intervention. Similarly, SOAAP [227] relies on code annotations and employs static analysis to flag possible data leaks. Both approaches are prone to human error due to manual effort. The CHERI [454] hardware memory capability model promises strong and efficient compartmentalization by extending RISC ISAs with capability instructions. Certain CHERI features (e.g., unforgeable pointers/capabilities, byte-level memory sharing) eliminate or mitigate some classes of CIVs. Nevertheless, CHERI is still a prototype [120]. We discuss the benefits and limitations of all three systems in §4.3.

Several TEE runtimes have been proposed [106, 159, 370, 218, 207] to transparently shield enclaves from the outside world by maintaining a secure interface. However, as demonstrated by several studies [436, 179] this cannot eliminate all CIVs, motivating fuzzers such as Emilia [179] and ConfFuzz. Before TEEs, sandboxing frameworks protecting applications from a malicious OS such as InkTag [240] and MiniBox [310] attempted to prevent Iago attacks by vetting/managing the memory mappings requests made by the protected program.

4.8 Conclusion

Breaking down monolithic software into compartments without reasoning about newly created interfaces leads to Compartment-Interface Vulnerabilities. This paper contributes an in-depth study of CIVs. We presented a taxonomy of CIVs and classified

the state of existing defenses. We proposed ConfFuzz, an in-memory fuzzing approach to investigate part of the CIVs identified in our taxonomy, along with their impact in compartmentalized software. Applying ConfFuzz to 25 applications and 36 libraries, we uncovered a large data-set of 629 CIVs from which we extracted numerous insights on the prevalence of CIVs, their causes, impact, and the complexity to address them: we confirmed how important CIVs should be to compartmentalization research, and highlighted how API design patterns influence their prevalence and severity. We concluded by stressing that addressing these problems is more complex than simply writing a few checks, proposed guidance on compartmentalization-aware interface design and adaptation, and motivated for more research towards systematic CIV detection and mitigation. We open-sourced the code of ConfFuzz, and the data set generated as part of this paper: `https://conffuzz.github.io`.[2]

Acknowledgements

We thank the anonymous reviewers for their insights. We are also grateful to David Chisnall and Istvan Haller for their insightful feedback. This work was partly funded by a studentship from NEC Labs Europe, a Microsoft Research PhD Fellowship, the UK's EPSRC grants EP/V012134/1 (UniFaaS), EP/V000225/1 (SCorCH), the EPSRC/Innovate UK grant EP/X015610/1 (FlexCap), the EU H2020 grant agreements 871793 (ACCORDION) and 758815 (CORNET), and the NSF CNS #2008867, #2146537, and ONR N00014-22-1-2057 grants.

[2] See Appendix C.

Chapter 5

SoK: Deconstructing Software Compartmentalization

Four attempts – failure;
Fifth attempt – enthusiastic but failure.

The Romance of the Three Kingdoms
Luo Guanzhong

This chapter presents a large-scale systematisation of knowledge of the field of software compartmentalisation. By identifying and framing existing trends and approaches in software compartmentalisation, along with instances of compartmentalisation that made it into the mainstream, we provide insights on the challenges we still need to tackle to bring the benefits of software compartmentalisation to the mainstream. These contributions address research questions **RQ5** and **RQ6**. This paper is currently under submission at a scientific venue.

Note on *Systematization of Knowledge* (SoK) Papers

As described by IEEE S&P [91], "SoK papers are contributions to evaluate, systematize, and contextualize existing knowledge. SoK papers are not surveys, in that they must provide a new viewpoint on an established area, support long-held beliefs with evidence, or present a comprehensive taxonomy of such an area."

This makes this chapter a natural ending for this thesis. Going beyond what a survey or background chapter would provide, we aim to look back at the work realized not just in this thesis, or in the scope of this thesis, but at the scale of the entire field, to

look to the future and determine the right problems to prioritize to bring the benefits of software compartmentalisation to the mainstream.

Contributions of the Author

I gathered and analysed the data presented in this publication. I wrote the paper. Nathan Dautenhahn and David Chisnall provided feedback on various drafts. Pierre Olivier provided feedback and edited the paper.

Abstract

In order to prevent the exploitation of unknown vulnerabilities, developers can partition monolithic software into lesser privileged compartments. While heavily explored since time-sharing systems, and despite an increased attention in the 2000s following models such as OpenSSH and qmail, privilege-separating software is still not a common practice today. In this paper, we systematically examine and comprehensively systematize results from 192 software compartmentalization efforts spreading the last 60 years. We confront this study with an analysis of 60 compartmentalized software that did reach the mainstream, showing how modern software, if at all compartmentalizing, still vastly relies on manual separation, custom abstractions, and heavyweight legacy mechanisms, poles apart from the state of research. We conclude that popularizing software compartmentalization and bringing research advances to the mainstream will require progress towards eliminating (and, when relevant, supporting) the definition of compartmentalization policies; towards better framing separation costs early on; to designing abstractions that will stand the test of time and progress; and to better challenging our threat models, particularly in light of interface safety issues.

5.1 Introduction

Despite decades of effort from the security community, vulnerabilities still plague software. Every year brings new critical kernel [23, 14, 13] and browser [24, 17] exploits, severe vulnerabilities in dependency chains [12], etc. Thwarting them ultimately remains a game of cat and mouse; protections are either incomplete, too expensive to be deployed, or can be bypassed. The last rampart when protections fail or when flaws are unknown is the *principle of least privilege* [392]. By granting each entity only the privileges it needs, one can ensure that the compromise of one part will not imply that of the whole.

Software compartmentalization is a prominent implementation of the principle of least privilege. With compartmentalization, developers divide a software monolith into lesser privileged components. Software compartmentalization inherits from a large body of research, starting with processes [181]; including OS models such as microkernels [468, 311, 238, 278, 199], security and separation kernels [387, 388, 117, 456, 116], or capability OSes [237, 158]; all the way to fine-grain application compartmentalization in the 2000s [445, 272, 434, 344] following models such as OpenSSH [373], qmail [137], or Postfix [231].

Not only has compartmentalization shown promise for containing memory safety issues [145, 384] and providing general fault isolation/resilience [311, 346], it also proved able to isolate untrusted third parties [110, 344], thwart supply-chain attacks [438, 213], side-channels [256, 345, 338], protect cryptographic secrets [315, 320, 434] and shadow stacks [150], or isolate unsafe parts of safe languages [334, 417, 383, 275, 128].

Still, to this day, compartmentalization techniques are not widely adopted outside security-aware developer circles. Despite tangible interest in research, compartmentalizing is still not a common engineering practice. Even when it comes to isolating cryptographic secrets, long pushed by many works [258, 225, 395, 227, 133, 315, 434, 147, 166, 141, 415, 226, 296, 313], approaches proposed in research are not adopted in practice by popular cryptography libraries. We are missing out on the security benefits compartmentalization can bring, at a time when systems security has become more important than ever.

Research speculated that this is due to a lack of automation [463, 133], to limitations of mechanisms [359, 469], to excessive performance overheads [434, 350], or a lack of strong security guarantees [298, 244], among other challenges. All are likely part of the problem, and through hundreds of research works in the past decades, progress was made on every front. Yet, few experts have a global overview of this progress.

This lack of a systematic perspective leads to a mismatch between what software compartmentalization needs to progress towards the mainstream, and the focus/framing of research efforts: most do not tackle compartmentalization's key aspects as a whole and thus produce solutions that cannot be relevant to making it a mainstream practice.

This paper takes a systematic approach to address these observations. We carefully define and frame software compartmentalization. On this basis, we comprehensively systematize 192 software compartmentalization efforts spreading the last 60 years into a consistent and structured field. We confront this study with an analysis of 60 compartmentalized software that did reach the mainstream, showing how modern software, if at all compartmentalizing, still vastly relies on manual separation, custom abstractions, and heavyweight legacy mechanisms, poles apart from the state of research. Based on these insights, we conclude with a discussion of open problems to address for compartmentalization to become a truly widespread engineering practice, and for research advances to make it into production. Concretely, this SoK contributes:

- A comprehensive systematization of existing partitioning approaches, compartmentalization abstractions, and mechanisms, and their limitations (§5.3);
- A study of the characteristics of compartmentalized programs that made it to the mainstream, and what should be learned from them (§5.4);
- A discussion of challenges that should be tackled to make compartmentalization a truly mainstream practice (§5.5).

5.2 Software Compartmentalization

Compartmentalization is a well-defined term in information flow security [398, 446], but its modern use referring to program privilege separation as in *software* compartmentalization has not been clearly characterized. We first address this lack. In the following, we adopt definitions of *subject*, *object*, and *permission* from Miller [339] and Saltzer [392], which we summarize as follows: a subject is a unit of computation (e.g., a thread of execution, an assembly instruction), an object is a unit of privilege enforcement (e.g., a byte of memory), and permissions are actions subjects may perform on objects (e.g., read, write). We refer to a maximal set of subjects sharing identical permissions as a *protection domain*.[1] This allows us to define software compartmentalization:

[1]Interested readers may find a glossary in Section 5.8. Precise understanding of these terms is not necessary to understand this paper.

> **Definition:** A compartmentalization of a program P is the set of (1) a policy to separate P into two or more protection domains (called compartments), and (2) the enforcement of this policy at runtime.

With a sanely defined and enforced policy, compartmentalization can mitigate many classes of attacks [145, 438, 434, 128] by containing exploits in compromised domains. Compartmentalization can be applied to any program: applications, OS kernels (microkernels [311] are compartmentalized kernels), hypervisors [408], firmware [269], etc. Compartmentalization can be *retrofitted* (i.e., integrated into formerly monolithic programs), or present in the initial design of a program.

A central problem of compartmentalization is validating all control and data flows at compartment boundaries [373]; we refer to this as ensuring interface safety. Improper validation results in a wide range of confused deputy vulnerabilities [234] well-studied in previous works [298, 344, 244, 161, 397].

Scope. In this paper, we refer to software compartmentalization as applied *within* one program (for instance one application, kernel, etc.), similarly to Provos [373]'s *privilege separation*. As we show later, this covers a vast and coherent body of work. Other isolation techniques [409] *across* programs or groups of programs are not here referred to as compartmentalization: e.g., isolation of applications on a commodity OS, whole-application sandboxing [192, 82, 217, 126], separation between user and kernel [165, 240], between VMs [131] hosting separate programs, between stakeholders (confidential computing [391]), between users, or containers [335]. Though these isolation works share challenges with software compartmentalization as we scope it, we focus strictly on the latter for space reasons.

Compartmentalization Trade-Offs. Compartmentalization approaches strive to optimize some or all of four properties: the Ⓢ *security and safety* benefits of compartmentalization (properties enforced, interface safety guarantees); the Ⓟ *performance* of separated software (compared to a monolithic design); the Ⓒ *compatibility* of compartmentalization with existing software and programming idioms (e.g., to minimize the reimplementation effort needed by programmers to compartmentalize); and the Ⓤ *usability* of separated software, ensuring that non-expert end-users can correctly operate (e.g., configure, maintain, monitor) compartmentalized software.

Striking the right balance between these properties is central to achieve techniques that are relevant to real-world usages. There is no silver bullet [360, 361, 227, 384, 189,

299]; nearly all approaches discussed next adopt a different point in the design space. These trade-offs are thus at the core of this study. In the following we refer to these four properties through Ⓢ, Ⓟ, Ⓒ, and Ⓤ.

5.3 Deconstructing Compartmentalization

Though compartmentalization is only meaningful as a whole, with a soundly defined and enforced policy, it raises distinct challenges that are often solved separately:

(P1) How to determine the right policy to enforce?

➽ Addressed by *policy definition methods*.

(P2) How to express policies in software, programming models, and idioms?

➽ Addressed by *compartmentalization abstractions*.

(P3) How to enforce policies at runtime?

➽ Addressed by *compartmentalization mechanisms*.

Nearly all works from the literature target a subset of these challenges. For instance, SOAAP [227] addresses (P1), SMVs [243] (P2), and Donky [400] (P3). In fact, P1-P3 are rarely addressed all at once for scale reasons. We ground our systematization on this division of challenges. Next, we will discuss key characteristics common to P1-P3 (§5.3.1), before comprehensively systematizing each challenge (§5.3.2 - §5.3.4). Figure 5.1 provides an overview of this discussion.

Methodology. We manually filter the program of leading[2] security and systems conferences for 2003-2023 (9083 papers) through titles to obtain a list of potentially relevant works (923 papers). We choose 2003 as the point when application compartmentalization gained visibility in research with Provos et al. [373] and Kilpatrick [272]. We then inspect abstracts to reduce this list to 89 relevant papers. In order to cover works prior 2003 and from other venues, we inspect the references of each paper to identify 63 further works, and complete the list with 40 more from our knowledge of the industry and literature, totaling 192 works. This process is visualized in Figure 5.2. 105 of these works, featured in Tables 5.1 to 5.3, address at least one of P1-P3. Our goal is not

[2]Ranked A* by CORE 2023: http://portal.core.edu.au/conf-ranks/?search=&source=CORE2023

Figure 5.1: Overview of P1, P2, and P3 along with the main characteristics that differentiate existing solutions to each problem.

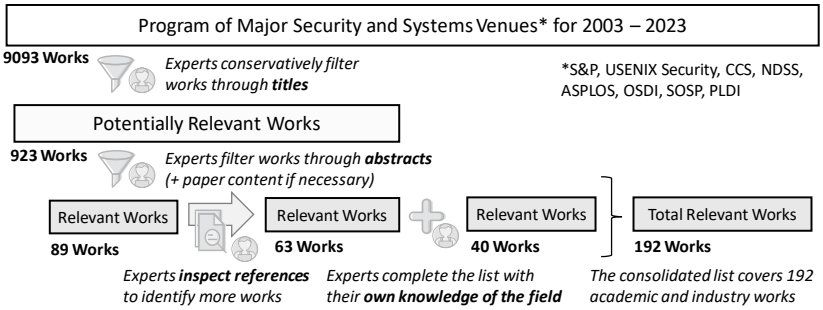


Figure 5.2: Visualization of the selection methodology.

exhaustivity (this is not a survey), but to capture a representative, substantial set of influential compartmentalization works. We define our data points by extracting, for P1-P3, key characteristics allowing us to differentiate these solutions (Tables 5.1 to 5.3). This results in 7 comparison points for P1, 8 for P2, and 7 for P3. For works spanning several categories (e.g., P1-P2), we include them in both, studying them from both perspectives as if they were separate works. Interested readers can read more about our methodology in Section 5.9.

5.3.1 Overarching Characteristics

We first introduce three key characteristics of compartmentalization approaches spanning P1, P2, and P3: the choice of subjects, target properties, and target trust model. These characteristics are central to qualifying any compartmentalization approach.

Choice of Subjects. The choice of subjects (*subject* as defined in §5.2) to compartmentalize is a key point to differentiate approaches. There are three categories: *code-centric*, *data-centric*, and *hybrid*:

- In code-centric (or *spatial*) approaches, subjects are defined as program instructions, and protection domains constitute code regions. For instance, the *libjpeg* image processing library can be put in its own protection domain [344].[3]
- In data-centric (or *temporal*, *horizontal* [454]) approaches, subjects are defined as temporal units of execution, e.g., a thread or a process. Protection domains may contain one or more of these subjects. For example, worker processes in a modern web server (see §5.4) constitute data-centric domains, all executing the same packet-processing loop in isolation.

[3]The example presented in Section 2.1.1 is another case of code-centric compartmentalization.

- These two strategies are not mutually exclusive: hybrid (or *object-oriented* [227]) approaches consider data-centric subjects bounded within code regions, e.g., a thread bounded to a specific library. Less popular than code- and data-centric approaches, hybrid approaches have been explored for multi-instance code-centric domains [327].

All three approaches suit different classes of applications. Spatial approaches are appropriate when distrust can be directed at a particular code unit (for example risky, or 3rd-party code), or when secrets are associated to specific data structures (e.g., secret keys, passwords, credentials). Conversely, temporal approaches are appropriate for programs handling mutually distrusting information flows, particularly with per-flow secrets, e.g., serving mutually distrusting clients as in web browsers or web servers. Hybrid approaches can accommodate mixed characteristics.

Target Properties. Compartmentalization approaches enforce various properties, including *integrity*, *confidentiality*, and *availability*. In this context, integrity guarantees that a subject cannot write out of its protection domain. Confidentiality guarantees that a subject cannot read out of its protection domain. Availability guarantees that a subject cannot prevent other protection domains from executing normally. Integrity is a prerequisite for all other properties: availability generally cannot be provided without integrity (without integrity, malicious domains can arbitrarily overwrite other domains), and similar issues arise when enforcing confidentiality without integrity [457]. Compartmentalization approaches can also enforce additional properties, typically to raise the bar against cross-compartment attacks. Among them, *Cross-Compartment Control-Flow Integrity* [395, 299] (CC-CFI) guarantees that the control-flow across compartments is correct: a compartment can only call compartments it would normally call according to the global Control-Flow Graph (CFG), and cross-compartment call sites can only call endpoints they would normally call according to the global CFG. Another one, *runtime re-compartmentalization* [360, 349] allows the policy to change at runtime, enabling it to achieve more suitable security or performance trade-offs.

Trust Models. Compartmentalization can enforce various trust models by assigning different subjects to domains, deciding for which domain to prioritize least-privilege, and which properties to enforce. All can be expressed as a composition of three key models: *sandbox*, *safebox*, and *mutual distrust*.

- The sandbox trust model reduces the privileges of an untrusted subject s_u in order to protect the rest of the system.[4] Least-privilege is enforced on s_u only. A popular use-case is protecting against vulnerable code, such as the request parser in a web server.

- The inverse of the sandbox model, the safebox model (or *vault* [400]), reduces the privileges of the whole system to protect a trusted subject s_t. Here, least-privilege is applied on everything but s_t. A typical use-case is the protection of cryptographic secrets [434].

- The mutual distrust model aims, for two disjoint sets of subjects S_1 and S_2, to eliminate the privileges of both parties on the other. Here we enforce least-privileges on both sets. A typical use-case is distrust among sandboxes to increase fault isolation [139].

All compartmentalization approaches rely on a *Trusted Computing Base* (TCB) [387] to enforce compartmentalization. The content of the TCB varies across approaches. Typically included are the CPU package and a compartmentalization runtime (also called *reference monitor* [118]), but the TCB can also cover a compiler, OS kernel, or additional libraries.

5.3.2 Policy Definition Methods (P1)

Definition: A *Policy Definition Method* (PDM) is a method to define a program privilege separation policy. PDMs identify subjects, objects, and permissions to enforce, and may be applied to existing, as well as from-scratch codebases.

We discuss the goals of PDMs, and how they reach them. This section is designed to be read along with Table 5.1.

Goals. Applied to policy definition methods, the properties from §5.2 translate to the following:

Ⓢ *Minimizing privileges.* Choosing boundaries to minimize the privilege of each protection domain; Accurately identifying the privileges needed by each protection domain to avoid over-privileged compartments.

[4] *Sandbox* can also refer to whole-program privilege reduction [192], e.g., with seccomp [82], or language-based techniques [217]. This matches our sandbox model, but not our definition of software compartmentalization (§5.2).

Table 5.1: Taxonomy of Policy Definition Methods. PDMs that also propose an abstraction are marked with *.

Separation Method	*Automation* ○◔◓●[1]	*Policy Language Type*		*Separation Granularity*	*Subject Selection: Spatial / Temporal*	*Analysis Approach*	*Language Specific*	*Additional Goals*	
		Annotations	*Placement Rules*					*Performance*	*Interface Safety*
Manual [243], [...]	○	N/A	N/A	Any	Any	N/A	○	N/A	N/A
Crowbar [141]	◔	○	○	Function	Spatial	Dynamic	○	○	○
MPDs* [360, 361]	◔	○	○	Component	Spatial	Hybrid	●	●	○
CubicleOS* [395]	◔	●	●	μLibrary	Spatial	Static[2]	●	○	○
Google SAPI* [35]	◔	●	●	Function	Spatial	Static	●	○	○
FlexOS* [299, 261]	◔	●	●	μLibrary	Spatial	Dynamic	●	●	○
RLBox* [344]	◔	●	○	Function	Any	Static	●	○	●
SOAAP* [227]	◔	●	○	Any	Any	Hybrid	●	●	●
SeCage* [320]	◓	●	○	Function	Spatial	Hybrid	●	○	○
PtrSplit* [318]	◓	●	○	Function	Spatial	Static	●	○	○
PrivTrans* [147]	◓	●	○	Function	Spatial	Hybrid	●	○	○
Glamdring* [313]	◓	●	○	Function	Spatial	Static	●	○	●
Shreds* [166], CAPACITY* [194]	◓	●	○	Any	Spatial	Static	○	○	●
DataShield* [153]	◓	●	○	Any	Spatial	Static	○	○	●
Swift* [170]	◓	●	○	Any	Spatial	Static	●	○	●
Jif* [471]	◓	●	●	Any	Spatial	Static	●	○	●
PM [319]	◓	●	●	Function	Spatial	Hybrid	●	●	○
KSplit* [248]	◓	●	●	Driver	Spatial	Static	●	○	○
Cali* [133]	◓	○	●	Library	Spatial	Static	○	○	○
CompartOS* [114]	◓	○	●	Linkage Unit	Spatial	Static	○	○	○
Enclosure* [213]	◓	○	●	Package	Spatial	Static	●	○	○
BreakApp* [438]	◓	○	●	Package	Spatial	Static	●	○	○
CompARTist* [247]	◓	○	●	Library	Spatial	Static	●	○	○
ACES* [171]	◓	○	●	Function	Spatial	Any[3]	○	○	○
ProgramCutter [463]	●	N/A	N/A	Function	Spatial	Dynamic	●	○	○
μSCOPE [384], SCALPEL [385]	●	N/A	N/A	Any	Spatial	Dynamic	○	●	○

[1] ○ = manual, ◔ = guided manual, ◓ = policy refinement, ● = full automation. [2] Loader-based. [3] Implemented with static analysis, dynamic analysis possible [171].

Ⓢ *Maximizing interface safety.* Choosing boundaries to maximize interface safety: minimizing leakage, maximizing the sanitization potential of data and control flow, and the ability to enforce interface semantics.

Ⓟ *Meeting performance goals.* Choosing boundaries to minimize the performance cost of separation: avoiding to cut critical paths, minimizing data copies, etc.

Ⓒ *Minimizing developer effort.* Minimizing the expertise and effort needed from developers to separate software.

Ⓤ *Maximizing completeness/soundness.* Soundly identifying protection domain privileges to avoid availability issues caused by under-privileged compartments (false positives resulting in crashes, increased latencies, etc.).

Policy definition methods must strike a balance between these goals. Research problems lie in these trade-offs, which we discuss next.

Trade-offs of Automation. Automation accounts for a major part of the research on separation and is structural for all other characteristics discussed in this section. Works seek to improve developer effort, privilege limitation, interface safety, or performance. We characterize four degrees of automation, visualized in Figure 5.3:

Manual methods (○ in Table 5.1) rely on the expert knowledge of developers to separate software. Developers must define a policy at the lowest level: *which* subject is given *what* permissions for *what* object. When determining permissions, manual approaches can be accurate, but are prone to human error, resulting in *false positives* (under-privileged compartments, hurting reliability) and *false negatives* (over-privileged compartments, weakening security properties). Similarly, performance and interface safety widely depend on expert knowledge and human error [299, 227]. Overall, it is nearly impossible to provide quality or correctness guarantees with manual separation because of its reliance on human expertise and engineering effort. Still, much of the literature falls in the manual category (§5.3.3), and manually-separated software can achieve robustness/reliability as we discuss in §5.4.

Guided manual methods (◔) such as RLBox [344] or SOAAP [227] are manual, but provide developers with tools to make the separation less tedious and error-prone. Often featuring a feedback loop [344, 227], they guide users to define and protect boundaries. Guided methods can bring firm guarantees to manually-separated software, e.g., eliminating classes of confused deputies [344] or information leaks [227].

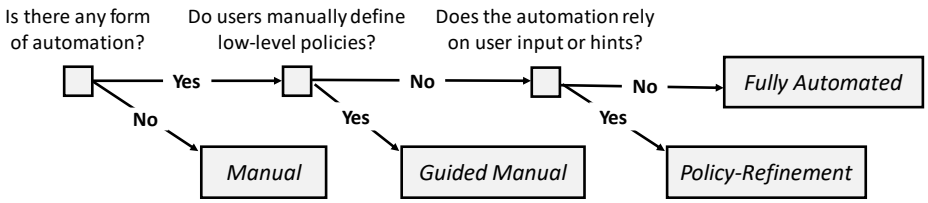


Figure 5.3: Decision tree distinguishing the four levels of automation in PDMs.

Policy-refinement methods (◓) automatically separate provided a high-level policy from the developer. Such policies, written in a *policy language*, provide the PDM with semantic information (e.g., annotations associating objects with a confidentiality level [471]) and/or high-level instructions (pinpoint a library to sandbox [133]). Policy-refinement approaches automatically refine such high-level policies into *concrete, low-level* policies, greatly limiting the amount of expertise required from developers to compartmentalize. Still, policy-refinement methods are not fully automatic: enough skills and/or expertise in the program are required from developers to provide a high-level policy. We expand on policy languages next.

Fully automated methods (●) automatically separate software without any policy input from the user. Instead of relying on semantic information on the software, which is hard to infer automatically [470, 153], fully automated approaches analyze the software for data dependencies, and apply least-privilege on this basis [384, 463]. One drawback of relying on data dependencies only is that, due to the architecture of software, data-dependency relationships may exist between confidential data and untrusted parts, resulting in partitionings with weaker security properties [384]. Thus, fully automated approaches trade off least privilege for developer effort.

Policy Languages. Policy languages allow developers to describe high-level security policies to Guided manual (◔) or Policy-refinement (◓) PDMs. They are central to achieve intermediate degrees of automation, providing PDMs with keys to understand program semantics and trust relationships, which are otherwise hard to extract automatically [470]. We distinguish between two types of policy languages: *annotations*, and *placement rules*. Annotation-based policies provide fine-grain semantic information on subjects and objects, such as describing shared, confidential, or sensitive entities [471, 147, 227, 320, 318, 319, 299] (e.g., object `key` is confidential, function `parse()` is sensitive), past vulnerabilities [227], or performance goals/bottlenecks [227, 319]. Annotation-based policies are often tightly coupled with program code. Conversely, placement rule languages provide coarse-grain, high-level descriptions of component

trust relationships [471, 133, 299] and/or building rules [319, 171]. Placement rule policies are generally clearly separated from source code, and are expressed in many ways: previous work proposed human-readable JSON files [133, 299], or build-system integration [248]. Annotations are more expressive than placement rules because they provide low-level semantics. This enables to separate at a finer grain, but requires more expertise from developers vs. placement rules. The two classes are not mutually exclusive and PDMs may leverage both [319, 299].

Separation Granularities. Many granularities have been explored in the literature (Table 5.1): functions [451], libraries [133], linkage units [114], drivers [248], software packages [213], etc. These choices are guided by design decisions. On the one hand, finer granularities of separation make it possible to better enforce least-privilege, or reach boundaries more favorable to performance. Fine granularities may also be necessary to tackle certain vulnerabilities, e.g., Heartbleed [193]. On the other, for some separation approaches, the threat model itself may be defined at a coarse granularity; larger components such as libraries or packages are valid units of trust in real-world scenarios such as supply-chain attacks [438]. Operating at coarser granularities may also reduce developer effort and expertise requirements: as granularities become finer, it becomes more complex to express policies and reason about them; higher-level boundaries such as libraries are more intuitive separation units than arbitrary internal functions. Finer granularities may also negatively impact interface safety: as boundaries are set at internal, less encapsulated software layers [384], interfaces are more exposed to confused deputies and harder to safeguard [299]. Finally, finer granularities pose technical challenges of state explosion [299], performance [249] and analysis/clustering [384].

Analysis Techniques. Automated PDMs (◔◓●) employ a range of static, dynamic, and hybrid techniques which, via subtle trade-offs, strongly impact final properties.

Static methods. When determining permissions, approaches based on static analysis are sound and guarantee the absence of false positives, but suffer from false negatives due to the fundamental imprecision of this class of techniques [319, 171]. This means that separations are *conservative*: separated software is guaranteed to function correctly, but compartments may be over-privileged. When applied to performance analysis, static approaches can provide relevant metrics [319], but generally not precise, performance estimates [227]. When applied to improve interface safety, static analysis approaches can detect potential issues at scale and systematically [344], but not precise impact metrics [244].

Dynamic methods. When determining permissions, dynamic approaches guarantee the absence of false negatives, but are *incomplete*, i.e., they suffer from false positives due to their fundamental coverage problem [384, 319]. Permissions granted to each domain are guaranteed to be needed, but domains may be left under-privileged so that separated software may not function correctly anymore for all workloads. Dynamic methods can provide precise performance estimates [299, 384] but only on covered workloads. Applied to interface safety, dynamic methods can detect vulnerabilities and their concrete impact, but not systematically [298].

Hybrid methods. Ideally, static and dynamic methods would compose to achieve complete and precise permission-, performance-, and interface analysis. In practice, composing them is challenging, as there is no clear approach to utilize the delta between static and dynamic results. On the privilege detection side, works use this delta as the *suspicious subset* [320], authorizing but reporting uses, which poses usability questions. Others use dynamic results in optimizations [147] or to predict performance accurately [227], with an otherwise static approach, or to measure information flow [319].

Approaches are Mainly Code-Centric (Spatial). Separations can be spatial, temporal, or hybrid, each more or less suited based on software designs (§5.3.1). The type of subjects targeted by a PDM deeply impacts its separation logic. As a result, automated PDMs generally need to specialize. Nearly all automated methods focus on spatial separation (Table 5.1); to the best of our knowledge, existing non-spatial PDMs fall into the guided-/manual categories. This may be due to the overall lesser popularity of non-spatial approaches (further shown in §5.3.3), but also to the greater complexity of temporal separation: while many automated spatial PDMs assume single-threaded programs, temporal separation brings requirements for concurrent separation, raising the complexity of the analysis [318]. Overall, there is a need for more research in automated methods that support non-spatial approaches.

Approaches are Language-/Domain-Specific. Most PDMs specialize on classes of programming languages (Table 5.1), for several reasons. First, some focus on domain-specific problems or threat models: e.g., pointer aliasing in C [318], untrusted packages in modern languages [213]. Second, specializing on language classes allows PDMs to leverage their specificities: their type system to detect API sanitization needs [344]; their memory safety [471, 170] or interpreted nature [438, 213] to simplify boundary detection; or the overall system architecture [247] to make assumptions on boundaries. There is, for instance, a vast body of work (only partially covered in Table 5.1 for

space reasons) specifically targeting 3rd-party Android library code [416, 402, 245], with some [364, 406, 474, 316, 247] specializing entirely on advertisement libraries. These works are well-covered by Acar et al. [110].

5.3.3 Compartmentalization Abstractions (P2)

Definition: A *compartmentalization abstraction* defines and implements primitives to express privilege separation policies in software. Depending on the semantics of these primitives, abstractions may be used to express different types of subjects, trust models, properties, etc.

We first discuss the properties pursued by compartmentalization abstractions. We then describe the core primitives abstractions provide, and how abstractions approach them to reach their goals. This section is supported by Table 5.2.

Goals. When applied to abstractions, §5.2's four high-level properties translate to the following:

Ⓢ *Enable for new/stronger security properties:* covering the range of properties from §5.3.1, but also fault-tolerance, smaller TCBs, finer granularities, etc.

Ⓢ *Maximize interface safety:* making it easy to achieve interface safety by construction: expose/enforce checking primitives, define interface restrictions, etc.

ⓈⓅ *Make the most of each mechanism:* expose primitives that make optimal usage of mechanisms (domain count, permissions enforced, domain-switch cost, etc.).

ⓈⓅ *Leverage domain-specific knowledge:* classes of software have specificities of their own: structure, type system, usage patterns, etc. Abstractions can leverage them to better balance security and performance.

Ⓒ *Easy integration/retrofitting:* provide abstractions that easily integrate in existing program structures, system designs, privilege levels (user, kernel, hypervisor).

Ⓤ *Work with generic mechanisms:* provide abstractions that can be implemented on top of any separation mechanism to maximize mainstream usability.

These goals are in fundamental tension: making the most of each mechanism *and* supporting generic mechanisms; leveraging domain-specific knowledge *and* serving generic usages; etc. Abstractions must trade off, as we discuss next.

Table 5.2: Taxonomy of Compartmentalization Abstractions. *Targets*: User, Kernel, Hypervisor. *Properties*: Confidentiality, Integrity, Availability, Recompartmentalization. *Semantics*: *Asynchronous*, *Synchronous*, SHared memory, MESsage passing.

Class	*Target* U/K/HV	*Abstraction*	*Subject Selection* *Spatial / Temporal*	*Semantics* CALL	*Semantics* ASSIGN	*Abstraction Granularity*	*Properties* C	I	A	R	*Interface Safety*	*Design Bound to Mechanism*
Mutual Distrust	U	Virtines [451]	Spatial	*S*	MES	Function	●	●	○	○	○	EPT
		ACES [171]	Spatial	*S*	SHD	Function	●	●	○	○	○	∅[5]
		SeCage [320]	Spatial	*S*	SHD	Function	●	●	○	○	○	∅[5]
		Hodor [236]	Spatial	*S*	SHD	Library	●	●	○	○	○	∅
		CAPACITY [194]	Spatial	*S*	SHD	*Any*	●	●	○	○	○[4]	ARM PAC+MTE
		Jif [471]	Spatial	*Both*	MES	*Any*	●	●	○	○	●	∅
		Arbiter [449]	Temporal	*S*	SHD	Function[1]	●	●	○	○	○	∅[5]
		SMV [243]	Temporal	*S*	SHD	Function[1]	●	●	○	○	○	∅[5]
		Salus [415]	Temporal	*S*	SHD	Function[1]	●	●	○	○	○	∅[5]
		LwCs [315]	*Any*	*S*	SHD	Function[1]	●	●	○	○	○	Page Table[2]
		Processes [181]	*Any*	*Any*	*Any*	*Any*	●	●	○	○	○	Page Table
		SOAAP [227]	*Any*	*S*	SHD	*Any*	●	●	○	○	○[4]	∅
		libMPK [359]	*Any*	*S*	SHD	*Any*	●	●	○	○	○	Protection Keys
	U+K	CheriOS [202]	Spatial	*Any*	*Any*	U/K-component	●	●	○	○	○	CHERI
		μkernel Servers [199], [...]	Spatial	*Any*	MES	U/K-component	●	●	○	○	○	∅[5]
		MPDs [360, 361]	Spatial	*S*	SHD	U/K-component	●	●	○	●	○	∅[5]
		RedLeaf [346]	Spatial	*S*	SHD	U/K-component	●	●	●	○	○	Safe Languages
		CubicleOS [395]	Spatial	*S*	SHD	μLibrary	●	●	○	○	○	Protection Keys
		FlexOS [299]	Spatial	*S*	SHD	μLibrary	●	●	○	○	○	∅
		xMP [371]	Spatial	*S*	SHD	*Any*	●	●	○	○	○	∅
		Monza [57]	*Any*	*A*	SHD	Function[1]	○	●	○	○	○	∅[5]
	K	VirtuOS [348]	Spatial	*Both*	SHD	K-component	●	●	●	○	○	EPT
		HAKC [333]	Spatial	*S*	SHD	Function	●	●	○	○	○	ARM PAC+MTE
		LibrettOS [349]	Spatial	*S*	SHD	K-component	●	●	●	●	○	∅[5]
Sandbox	U	Cali [133]	Spatial	*S*	SHD	Library	●	●	○	○	○	∅[5]
		CompARTist [247]	Spatial	*S*	MES	Library	●	●	○	○	○	∅[5]
		Enclosure [213]	Spatial	*S*	SHD	Package	●	●	○	○	○	∅
		Google SAPI [35]	Spatial	*S*	MES	Function	●	●	●	○	○	∅[5]
		RLBox/μswitch [344, 365]	*Any*	*S*	SHD	Function	●	●	○	○	●	∅
		Wedge [141]	*Any*	*S*	SHD	Function[1]	●	●	○	○	○	∅[5]
	U+K	CompartOS [114]	Spatial	*S*	SHD	Linkage Unit	●	●	●	○	○	CHERI
	K	LVDs/KSplit [347, 248]	Spatial	*S*	MES	K-component	●	●	○	○	○	∅[5]
		XFI/LXFI [201, 327]	*Any*	*S*	SHD	K-component	●	●	○	○	●	SFI
	HV	Nexen [408]	Temporal	*S*	SHD	Per-VM domain	●	●	○	○	○	Page Table[3]
Safebox	U	Shreds [166]	Spatial	*S*	SHD	*Any*	●	●	○	○	○[4]	∅[5]
Dual World	U	Privman [272]	Spatial	*S*	MES	Function	●	●	○	○	○	Page Table[2]
		Privtrans [147]	Spatial	*S*	MES	Function	●	●	○	○	○	Page Table[2]
		Swift [170]	Spatial	*Both*	MES	*Any*	●	●	○	○	●	∅
		Glamdring [313]	Spatial	*S*	MES	Function	●	●	○	○	●	∅[5]
		PtrSplit/PM [318, 319]	Spatial	*S*	MES	Function	●	●	○	○	○[4]	∅[5]
		DataShield [153]	*Any*	*S*	SHD	*Any*	●	●	○	○	○[4]	Bounds Checking
		ERIM [434]	*Any*	*S*	SHD	*Any*	●	●	○	○	○	Protection Keys
	K	Nested Kernel [183]	Spatial	*S*	SHD	Function	●	●	○	○	○	Page Table

[1] Inherited from thread-like semantics, [2] from process-like semantics, [3] from the Nested Kernel, [4] The PDM does (to a certain extent).
[5] The abstraction could plug onto any intra-AS mechanism, though the paper or documentation claims reliance on a particular one.

Primitives. Compartmentalization abstractions instantiate the notion of *compartment*, defining the type of subjects compartments may contain, properties that they may enforce, and the trust models that the compartmentalized system may implement. With this, abstractions must provide five primitives: *`CREATE` and `DESTROY`* a compartment (defining semantics of compartment lifetime management, default permissions of new compartments, etc.); *`CALL` and `RETURN`* from a compartment (defining semantics of cross-compartment control flow); and *`ASSIGN` privileges* (granting/revoking permissions across compartments, resource ownership). Abstractions achieve trade-offs by *controlling the semantics of these primitives.*

Not all five primitives must be exposed to users; when they are, we refer to them as having explicit semantics. Inversely, primitives can be implicit, i.e., handled automatically under the hood. Implicit semantics are common in abstractions that are coupled with the PDM (e.g., automatic `CREATE`/`DESTROY`, transparent `CALL`s, automatic `ASSIGN`). Abstractions may also provide additional primitives: e.g., to support additional properties such as availability. We now detail the core primitives. For space reasons we focus on `CALL`/`RETURN` and `ASSIGN`.

`CALL`/`RETURN`. Cross-compartment control flow can be approached *synchronously*, or *asynchronously*. In the synchronous case, `CALL` and `RETURN` are semantically similar to a local function call and thus transparent to separated programs. In the asynchronous case, `CALL` and `RETURN` abandon function-call semantics: the execution of the caller domain continues after the call, and the return is processed by the caller similarly to a separate message [292] (e.g., as part of an event loop). Asynchronous semantics are less popular in Table 5.2. This is likely due to the fact that asynchronous semantics are more disruptive compatibility-wise: applications must be "structurally" aware of the separation, and redesigning for asynchronous behaviors is non-trivial [454]. Still, asynchronous semantics can be beneficial to performance, as their non-blocking nature can mask boundary-crossing delays [411]. For certain target properties such as availability, `CALL` and `RETURN` ultimately need distantiating from function call semantics, as new error types appear: timeout, callee compartment failure, etc. These translate into asynchronous features in call semantics that may otherwise be synchronous [114].

Regardless of their implementation, `CALL` and `RETURN` must be *sanely* specified. Generally, "sanely" means: 1) guaranteeing the validity of control-flow entry-points in compartments (compartments cannot `CALL` or `RETURN` to arbitrary code the context of other compartments, for obvious security reasons); 2) switching call stacks and clearing registers appropriately to avoid unintentional leakages; but also 3) ensuring that the abstraction composes safely with other system interfaces.

ASSIGN. `ASSIGN` semantics are structured by two fundamental approaches to communicating data [125]: *message passing*, and *shared data* (or *message/object* systems [292]). With message passing, domains share data across boundaries via messages over a communication channel (e.g., POSIX sockets or pipes). This means that objects are not only systematically copied, but also marshalled, and as part of this, potentially serialized and checked. This makes message-based solutions rather disruptive compatibility- and performance-wise: they do not map to natural shared-memory semantics found in applications, and require copies. Still, systematic copying and checking greatly benefits security [298]. With shared memory, protection domains both have privileges over shared memory, and communicate via loads/stores. Although copies can still be made systematic by the abstraction [344], it is not the norm: copies are costly, it is thus enticing to avoid them whenever possible. This makes shared memory much less disruptive compatibility- and performance-wise, but potentially deceptive security-wise [298]. Note that we describe exposed abstraction semantics here; under the hood shared-data can be implemented on top of message passing, and vice-versa [125].

Trust Models. Safebox, sandbox, and mutual distrust (§5.3.1) are all represented in Table 5.2. Sandbox and mutual distrust abstractions all support arbitrary scenarios, i.e., the number of sandboxes/mutually distrusting compartments is not limited. Although Shreds [166] enables for arbitrary safebox scenarios, all other safebox abstractions are limited to two compartments (trusted/untrusted, called *Dual World* in Table 5.2). This shows a lesser interest in applying distrust among trusted entities. Though safe/sandbox abstractions can both be implemented on top of arbitrary mutual distrust (and can thus be seen as special cases), they show structural differences: the presence of safe/sandboxes only, or of a fixed number of compartments, considerably simplifies their semantics. Each presents benefits. Mutual distrust abstractions are more expressive and come closer to least privilege. On the other side, they must enforce integrity and other properties on all compartments, whereas one-sided distrust models (safe/sandboxes) only need enforcing it on the trusted side, benefiting performance. Mutual distrust also makes interface hardening generally more challenging and performance-costly [298].

Enforcing More or Fewer Properties. All abstractions in Table 5.2 enforce integrity, a prerequisite for other properties (§5.3.1). Most target confidentiality, and some target availability, runtime re-compartmentalization, or interface safety.

Confidentiality. Since nearly all abstractions provide confidentiality, we focus on the impact of *not* doing so. Not providing confidentiality does not cause structural changes in abstractions [57]. It may benefit performance, as it enables zero-cost read-only data sharing, i.e., fewer copies at compartment boundaries. It also simplifies separation: only write-shared data require explicit sharing. On the downside, not providing confidentiality vastly reduces the abstraction's ability to counter information leakages: the only remaining barrier is the compartments' ability to limit data exfiltration vectors. It may also be detrimental from an interface-safety viewpoint, defeating randomization (thus easing cross-compartment exploits). Lastly, avoiding copies at boundaries may make the system more prone to shared-memory TOCTOU [298]. Overall, with the exception of legitimate integrity-only use-cases (e.g., shadow-stack protection [281]), giving up confidentiality trades security for performance and compatibility.

Availability. Abstractions may go beyond fault isolation and target fault tolerance across compartments. This brings many well-known challenges from the fault-tolerance and distributed-systems fields [437]: avoiding, detecting, and recovering from resource exhaustion, various failures (omission, Byzantine), etc. Particularly relevant to compartmentalization are resource ownership problems [346] (when restarting a domain, can shared resources be released?), and state coherence issues [143] (the state of restarted domains may be incoherent with that of other components), as compartmentalized components, particularly when retrofitting, are typically less encapsulated [298]. Abstractions take a vast variety of approaches, whose exhaustive listing outreaches the scope of this paper: bounded execution in resources [348] or time [227, 114], and generally performance isolation [228] to tackle resource exhaustion; leveraging type systems for resource ownership [346]; proposing manually-designed interface wrappers [419, 346], per-component fault-handlers [114], careful TCB/interface designs to store state outside of domains [349], or recursive restarting of relevant components [348] for state coherence. As the problem is generally hard, all require expert understanding of fault domains, rely on a variable amount of manual engineering, and not all approaches are complete; e.g., Google SAPI [35] automatically re-iterates failed calls/restarts faulty components, but does not consider state coherence problems.

Runtime Re-Compartmentalization. Some abstractions also support changing the policy enforced at *runtime* [360, 361, 349], e.g., to adapt policies to evolving requirements in performance and fault isolation. Beyond technical challenges of achieving transparent, fast re-compartmentalization, we observe that this poses non-trivial security challenges in adversarial contexts: assuming attackers can wait for the weakest fault-isolation profile to be instantiated, or influence workloads to trigger such profiles

(e.g., generate more/different traffic), then the overall security properties are that of the *weakest* profile achievable at any point in time. This may be true even when attackers cannot wait; since component states are preserved across policy changes, any undetected corruption triggered by an attacker during a "strong" profile will eventually reach other components when weaker profiles trigger, similar to delay attacks [466]. This limits applications of re-compartmentalization to non-adversarial scenarios.

Interface Safety. Although a central problem of compartmentalization, most abstractions consider interface safety orthogonal in Table 5.2. Instead, works often transfer the responsibility to PDMs by assuming a safe separation policy, or to downstream developers by assuming hardened components. Still, though the purpose of compartmentalization abstractions is not to help users defining security policies (this is the role of PDMs), they can contribute to interface safety by *making it harder to implement unsafe policies*, thus ensuring that compartment interfaces are free of (certain classes of) confused deputies. They may do so at various levels, e.g., by *enforcing restrictions on interface definitions*, such as restricting interface-crossing types [327, 344] or enforcing points-to ranges in interface-crossing pointers [327, 313, 344, 270]; by *forcing the presence of checks on interface-crossing data*, forcing users to write checks [344], or checking automatically when possible [327, 313, 344]; or by *enforcing restrictions on cross-compartment control-flow*, providing primitives to specify and enforce API call ordering [344], or by enforcing additional properties such as CC-CFI to raise the bar for cross-compartment attacks [395, 299]. These measures are not orthogonal to the core mission of compartmentalization abstractions, as they generally cannot be implemented independently. For instance, without knowledge of compartments mappings, it may be impossible to verify pointers/indexes; to check reference types; to check call ordering; to implement CC-CFI; etc. Hu et al. [244] and Lefeuvre et al. [298] provide more in-depth coverage of the topic of interface safety in compartmentalization.

Abstractions Specialize Towards...

Subjects: Spatial/Temporal/Hybrid. Most abstractions specialize towards a certain class of subjects (Table 5.2). Abstractions focusing on spatial models (§5.3.1) assume a direct mapping between code and compartment. As a result, `CREATE` and `DESTROY` are *implicit*, i.e., developers are not provided with explicit controls to manage compartment lifetime, and the abstraction is *static*, i.e., the number and content of compartments is known at compile time. Conversely, those focusing on temporal models feature *explicit* `CREATE` and `DESTROY`, and are *dynamic*, i.e., the number and content of compartments

may not be known at compile time and depend on the control flow taken at runtime. Abstractions that are hybrid (*Any* in Table 5.2) are similar to temporal abstractions but allow restricting the code available to a compartment; they can be leveraged to implement both temporal and spatial separations.

Granularities. Abstractions may also specialize towards specific domain granularities. These decisions are embodied in `CREATE` and `CALL` semantics which define the granularity at which compartments may be created and entered. This specialization comes with various goals: target properties may imply a granularity (e.g., fault resilience implies coarser grains), threat models may imply a granularity (library sandboxing → library granularity), or abstractions may be coupled together with a PDM that itself restricts granularity.

Mechanisms. Many abstractions are tightly coupled with a specific mechanism (Table 5.2). This is typically done to better leverage mechanism-specific properties such as safe copy-less sharing for capabilities, or strong typing, points-to knowledge, and provable termination for safe languages. This practice has drawbacks: beyond making abstractions unusable without their related mechanism, such coupling creates a mechanism dependency in downstream programs, which curbs efforts to roll out new mechanisms (e.g., `fork()` makes it hard to replace the page table [134]). These limitations incited a recent trend towards mechanism-agnostic abstractions [344, 299].

Modes: User/Kernel/Hypervisor. In userland, semantics of compartmentalization abstractions are heavily influenced by the presence of the user/kernel interface. Compartmentalization abstractions may [243] or may not [434] be exposed as part of the user/kernel boundary for performance, security, or compatibility reasons. As we discuss in the next section, user-mode abstractions must harmoniously compose with kernel interfaces such as processes, threads, signals, or system calls. Different factors shape kernel and hypervisor mode compartmentalization abstractions. Kernel and hypervisor codebases are often designed assuming ambient privilege on hardware and user data, making the TCB (and retrofitting) less obvious than in userland. The need to integrate with low-level events such as interrupts brings even more specific challenges that make it necessary for abstractions to integrate deeper in kernel and hypervisor designs [183, 347, 299], encompassing boot, scheduling, memory management, or interrupt handling.

Composing with Other Abstractions.

Threads. Compartmentalization abstractions must define a threading model for

compartments, typically either *orthogonal*, or *coupled*. In the orthogonal case [327, 434, 299], threads cross protection domains as they `CALL` and `RETURN`. To ensure safety, these operations must guarantee that the underlying thread state is updated to reflect the crossing, and carefully define the behavior of thread local storage. Orthogonal threading models are exclusively used in *spatial* approaches (by definition, see §5.3.1). Conversely, in the coupled case [141, 449, 243], threads are immutably assigned to a protection domain at their creation, and `CALL` spawns a new thread in the desired protection domain. Spatial, temporal, and hybrid approaches may all adopt coupled threading.

Processes. Although processes are themselves a compartmentalization abstraction, they are also used by programs for reasons other than protection [134]. Prior works showed that composing compartmentalization abstractions with processes is error-prone and source of confused deputies [173, 399]. User-mode compartmentalization abstractions must thus take special care defining how they compose with processes. To ensure safety, approaches may intercept and forbid attempts to spawn new processes [399], at the expense of compatibility.

System Calls (& Other System Interfaces). We discussed earlier how compartmentalization abstractions can contribute to safeguarding intra-program compartment interfaces. Yet, interfaces with the *rest of the system* are also major interface safety weak spots [397, 173, 442, 298]. These include, in user-mode, system calls, other kernel abstractions (pseudo-filesystems [93, 90], files, sockets), but also interfaces exposed by other applications on the system [163]. Safeguarding these interfaces is non-trivial: mechanisms come with different protection needs (e.g., protection keys with "PKU pitfalls" [173]); protection needs are ABI-bound and thus vary across OSes, configurations, and architectures; protection must be maintained as these ABIs evolve; the protection effort itself comes with application compatibility problems (e.g., precisely detecting the OS features required by individual application components is hard [185, 152]); and this protection often results in noticeable performance overheads [173]. Compartmentalization abstractions either attempt to solve this problem through careful composition with the kernel [227, 344, 315, 213, 399, 365], or scope it out as a separate problem [318, 434, 236] addressed by separate solutions [372, 452, 82, 72]. Still, there is a growing consensus that user-mode compartmentalization abstractions should be designed hand in hand with kernel and broader system interfaces to maximize interface safety [365, 399, 173].

Table 5.3: Taxonomy of Compartmentalization Mechanisms. Page-Table = PT; Permissions: Read, Write, Execute, Address (*create pointers to*), ● = supported, ◐ = supported by some, ○ = unsupported; Overhead: *free* = ○ < ◔ < ◓ < ◐ < ● = *very costly*.

	Mechanism Class		*Trust Model*	*Permissions* R	W	X	A	*Granularity*	*№ of Domains*	*Conditioned*	*TCB*	*Domain Switch Cost (Versus Non-Separated)*
Hardware	Physical Separation [387]		Mutual	●	●	●	○	Physical Mem.	№ of machines	○	Full	◓ – ● (link latency)
	PT	Access Bits [44], EPT/`vmfunc` [45]	Mutual	●[1]	●[1]	●[1]	○	Page	∞	◐ (in Ring 0)	Full	● (PT switch + [4])
		Supervisor Bit [44, 296, 297]	Single	●[1]	●[1]	●[1]	○	Page	2 (kernel/user)	○	Full	◐ (interrupt + [4])
		Mondrian Mem. Protection (MMP) [459]	Mutual	●[1]	●[1]	●[1]	○	Word	∞	○	Full	◓ (MMP hardware + [4])
		Protection Keys [41, 46, 5, 400, 469]	Mutual	●	●	◐	○	Page	8-1024 [5, 400][3]	●	Full	◔ (special register flip + [4])
	Segmentation-like Hardware [208, 341]		Single	●	●	○	○	Byte - Page [341]	2 (safe/unsafe)	○	Full	◔ ([4])
	TEE	Enclaves [47, 178]	Mutual	●	●	●	○	Page	∞	○	TEE	● (enclave call, incl. [4])
		Confidential VMs [48, 2, 49]	Mutual	●	●	●	○	Page	∞	○	TEE	● (> EPT switch)
		World Separation [6, 1]	Single	●	●	●	○	Page	2 (trusted/rest)	○	TEE	● (world switch, incl. [4])
	Hardware Capabilities [154, 440, 454, 343]		Mutual	●	●	●	●	Byte	∞	○	Full	◓ (special instr. + [4])
	Bounds-Checking Hardware [186, 89, 396, 273, 281]		Mutual	●[1]	●[1]	○	○	Byte	∞	○	Full	◓ (bounds hardware + [4])
	(Other) Tagged Architectures [31, 86, 412, 455, 246, 386, 257, 188]		Mutual	●[1]	●[1]	○	○	Byte - Words[2]	16[3] - ∞ [257]	●	Full	◓ (tagging hardware + [4])
Software	Software Capabilities [237, 158]		Mutual	●	●	●	●	Byte	∞	○	Full	◓ – ● (impl. dep, incl. [4])
	Bounds-Checking Software [413]		Mutual	●	●	◐	○	Byte	∞	○	Full	◓ – ● (impl. dep, incl. [4])
	Safe Langs. [80] / Software Verif. [304, 280]		Single	●	●	●	●	Byte	2 (safe/unsafe)	○	Full	○ (function call)
	Software Fault Isolation [445, 155, 467, 472, 342, 229, 260, 259]		Single	●	●	●	●	Byte	∞	○	Full	◔ ([4])
	Memory Encryption / AES-NI [281]		Mutual	●	●	○	○	128 bits	∞	○	Full	◐ (copy key + encrypt + [4])

[1] Not all combinations of R/W/X supported. [2] Covers many granularities, see [257]. [3] Separate approaches [359, 223, 333] increase it. [4] Register saving/scrubbing, stack switch.

5.3.4 Compartmentalization Mechanisms (P3)

Definition: A *Compartmentalization Mechanism* is a solution to enforce separation, as defined and implemented by PDMs and compartmentalization abstractions, at runtime.

Next, we concretize this definition by discussing the fundamental primitives a compartmentalization mechanism must provide. Then, we show how mechanisms approach these primitives to reach trade-offs, going through Table 5.3.

Mechanism Primitives. In order to achieve minimal compartmentalization, a mechanism must support two primitives: 1) a *protection domain* primitive, and 2) a *communication* primitive. A protection domain enforces subject separation, and must at least guarantee cross-domain integrity (§5.3.1). A communication primitive must be able to transmit bits bidirectionally across compartments, and to enforce compartment control-flow entry points. Communication primitives can be implemented in many ways [292]: message passing, shared memory, specialized control-flow operations (e.g., cross-compartment call). Still, a simple message passing primitive suffices for compartmentalization. `CALL`, `RETURN`, and `ASSIGN` (§5.3.3) can all be implemented on top of it, and `CREATE`/`DESTROY` can be left implicit. If we expand the mechanism with a third primitive to `CREATE` domains, we obtain a general compartmentalization mechanism; `DESTROY`, and other mechanism-specific primitives, may also be supported.

In certain cases, the suitability of a mechanism for compartmentalization may be *conditioned* to certain requisites. For instance, compartment control-flow entry points cannot be enforced in PKU [46] without additional measures [434, 236] to monitor and restrict key-editing instructions (`wrpkru`, `xrstor`, etc.). Other conditioned cases include PT-based protection in kernel mode [183], or bounds-checking [89, 281]. Not all mechanisms discussed next were specifically designed for compartmentalization (e.g., TEEs, bounds-checking). Still, all are either *suitable* or *conditioned*, and have been leveraged for compartmentalization in practice.

Trust Models & TCB. Mechanisms are designed for specific trust models (*TM* in Table 5.3). Here it is sufficient to distinguish between *single* and *mutual* distrust. For instance the PT supervisor bit enforces single distrust, protecting one subject (the kernel) from others (users), whereas mutual distrust allows protection of arbitrary sets of subjects from other arbitrary sets of subjects. This constrains the trust models abstractions can implement on top of these mechanisms (§5.3.3).

Mechanisms also influence the content of the TCB. In the general case (*Full* in Table 5.3), TCBs of compartmentalized systems include (part of) the workload, compiler, loader, system software, firmware, CPU Package, and physical environment. TEEs exclude firmware and physical environment from the TCB [159, 210]. From a compartmentalization view, the TCB can be the only difference between otherwise similar mechanisms, e.g., confidential VMs/EPT.

SW / HW. Mechanisms can be implemented in hardware (ASICs, FPGAs, simulators) or in software. Compartmentalization advances have historically been driven by progress in hardware, which enabled separation granularities and security properties previously unreachable in software. Still, hardware is not a silver bullet: hardware takes time to reach end-devices, can be cost-prohibitive, and often ends up heterogeneous due to the lack of industry standard. In recent years, software-based approaches building on commodity hardware (MMU) have thus been particularly successful in contributing to the spread of compartmentalization practices (e.g., SFI with WebAssembly [229]).

Permissions Enforced. Mechanisms can target different subsets/combinations of four primitive permissions: *read*, *write*, *execute*, and *address* (create pointers to). Subsets are common: most protection key approaches [41, 46, 5, 400] do not protect instruction fetch; PT approaches do not support all combinations of R/W/X [44]; and capabilities [154, 237] also protect addressing, something that few other classes achieve. These specificities are the result of trade-offs. Security benefits from more expressiveness in permissions, allowing for finer least-privilege enforcement. Properties such as addressing also benefit interface safety, tackling certain classes of confused deputies by construction [298]. On the other hand performance is sensitive to implementation constraints which may require to trade off security: PT entries are limited in size, and using more bits to represent more permissions requires separate tables, degrading performance; capabilities often trade off performance (e.g., trough cache pressure [460]). This poses non-trivial problems to abstractions, which must map diverse levels of permission expressiveness to the previously described high-level properties (§5.3.3).

Enforcement Granularity. Mechanisms enforce permissions at varying memory granularities. At the extremes, enforcement may be done at the granularity of the entire physical memory of a machine [387], or at byte granularity [454]. Others (Table 5.3) operate on pages, machine words, 128-bit chunks, etc. Granularity choices too result

from trade-offs between performance, security, and compatibility. As discussed previously, finer granularities are generally desirable from a security standpoint, and mechanisms strive to achieve them. On the other hand, performance and memory footprint often benefits from coarser granularities, due to implementation constraints: supporting finer grains implies storing more permission information, and potentially increases the complexity of permission checks, or of protection domain instantiation.

Number of Domains. In cases, the domain creation primitive restricts the maximum number of domains for practical reasons: protection keys support, depending on the implementation, a handful [5] to over a thousand [400] domains. The domain creation primitive may also not exist at all: physical separation [387], TrustZone [6], and others, rely on a fixed number of physically separated domains. Other mechanisms may not limit the number of domains, but other considerations limit it in practice (e.g., scalability, size of memory). These decisions too are all trade-offs. Performance, like permissions/granularity, is bound to technical constraints: for protection keys, the number of domains can be increased with virtualization at a significant performance cost [359, 223, 469].

Performance. Mechanisms impact performance in many ways: latencies of compartment switch, creation, modification, and destruction; locality and cache effects; domain-crossing sanitization costs (which may be accelerated by the mechanism); scalability (domain switch, creation, modification, or destruction costs growing as a function of domain count or domain size); and other mechanism-specific runtime overheads (memory access cost, generated code size, etc.) [281, 255]. The performance profile of mechanisms varies widely. PKU domain switches, for instance, are unprivileged and thus fast, but domain creation/modification requires a costly trap [46]. With software memory encryption, domain switches are expensive (corresponding to full compartment encryption and decryption), but their creation and modification is merely an unprivileged bookkeeping operation, i.e., cheap.

There is a focus on domain switch latencies in the literature, as they are often a dominant cost factor [280]. Domain switch costs are the consequence of design and architecture decisions leading to security, performance, and compatibility trade-offs: Is the switch a *privileged primitive*, i.e., do cross-domain switches require trampolines/-traps to the TCB with elevated privileges? Should domain switches require trampolines at all, or should they be encoded with separate load/store instructions [208, 341]? Should switches be made faster at the expense of, e.g., compartment creation costs?

How deeply can compatibility (with compilers/kernels/programs) be broken? How generic (granularity, domain count) must the mechanism be? Because of these trade-offs, comparing mechanism domain switch costs may be deceptive: e.g., conditioned mechanisms (§5.3.4) such as PKU [46] are very fast (unprivileged register switch), but insufficient by themselves to guarantee safety, requiring combination with additional software techniques [359, 223] that have costs and trade-offs of their own.

5.4 Deployed Compartmentalized Software

How does the vast state of the art systematized in §5.3 translate in practice? To answer this, we discuss a corpus of 60 mainstream compartmentalized programs (Table 5.4). We constitute the corpus via a systematic search in the Debian 13 archive [26] (55 apps), completed with previous works (§5.3, 3 apps), and our knowledge of the field (2 apps). For the former, we manually triage all Debian packages with >1K installations [27] (1520 apps), whose source feature privilege-separation keywords. For space reasons we group applications in 12 classes, which we classify and describe based on our taxonomy from §5.3. Interested readers can find a a detailed list of keywords and programs in Section 5.10. We present our insights next.

Software compartmentalization is (still) not the norm. As Table 5.4 shows, compartmentalized designs started gaining mainstream awareness with applications such as qmail [137, 230], OpenSSH [373], or Postfix [231] in the mid-2000s. Compartmentalization progressed since then, driven by the challenges of the web, as well as by the OpenBSD and academic communities. Still, today compartmentalized designs remain a minority (< 55 out of 1520 apps), constrained to security-aware vendors (8 / 12 classes in Table 5.4 are authored by academics or security professionals). Non-expert developers, even of popular software, still do not commonly compartmentalize.

With skills and time, retrofitting is realistic. Partly due to the OpenBSD privilege separation effort, cases where separation was retrofitted outnumber those architected with separation in mind (Table 5.4). This shows that retrofitting can be realistic even in service-critical, long-established codebases such as OpenSSH, V8, or Firefox. In cases, the deployment of compartmentalization schemes caused functional regressions, e.g., due to overly restrictive policies [70, 69]. Described in §5.3.2, these issues are a concern when deploying policies obtained manually or dynamically, *which is accepted by practitioners in Table 5.4*. Dunlap [192] covers this in greater details.

Table 5.4: Characteristics of mainstream compartmentalized software. Abbreviations same as in Tables 5.1 to 5.3.

Software Class	*Author Profile*[1]	*SPM* (§5.3.2)	*Abstraction* (§5.3.3)						*Mechanism* (§5.3.4)
			Name	*Trust Model*	*Model*	*Semantics*	*Properties*	*Granularity*	
Google V8 [37]	Sec. Pro.	○[3]	Custom	Sandbox	Spatial	$\mathscr{S}$, SHD	CI	Coarse (High-level components)	*Fixed*: SFI
OpenBSD Privilege-Separated Userland (>40 apps) [65][2]	Sec. Pro.		Custom (Process-Based)	(*mainly*) Sandbox	(*mainly*) Spatial	$\mathscr{S}$, MES	CI(A)		*Fixed*: PT (Due to a dependency to `fork()` semantics)
IRSSI [51]	Academic			Safebox	Spatial	$\mathscr{S}$, MES	CI		
Debian `man` [56]	Other			Sandbox	Spatial	$\mathscr{S}$, MES	CI		
DHCPCD [29]	Other				Temporal	$\mathscr{A}$, MES	CIA		
VSFTPD [98]	Sec. Pro.	○			Temporal	$\mathscr{S}$, MES	CI		
qmail [137, 230], Postfix [231], djbdns [461]	Sec. Pro.				Spatial	$\mathscr{A}$, MES	CIA		
Dovecot [30]	Other				Spatial	$\mathscr{A}$, MES	CIA		
Web Servers [43, 3, 53]	Other				Temporal	$\mathscr{A}$, *both*	CIA		
Browser Site Isolation [163, 447, 378]	Sec. Pro.		Site Isolation		Temporal	$\mathscr{A}$, *both*	CIA		
μkernels [468, 311, 238, 278, 199] [...]	Academic		μkernel	Mutual Distrust	Spatial	*both*, MES	CIA		*Fixed*: PT[5]
Firefox Library Sandboxing [58]	Academic	◔[3,4]	RLBox	Sandbox	Spatial	$\mathscr{S}$, SHD	CIA	Finer (Libraries)	*Flexible*

[1]Sec. Pro. = Security Professional, [2]Including OpenSSH/NTPd/SMTPd, X.org (Xenocara), and others, [3]Separation was retrofitted, [4]RLBox, [5]Alternatives in research [224].

Performance (still) matters. Hardware historically imposed a heavy performance tax to compartmentalized software. This explains why most of Table 5.4 came together with faster hardware in the 2000s. A textbook case, the Windows NT 3.x kernel compartmentalized its graphics stack in the 1990s under the influence of microkernels, but soon reverted this in NT 4.0 reacting to performance concerns [59]. Still today, overheads are a decisive factor when shipping compartmentalization schemes to production [81]. This importance is reflected in research, with a majority of performance-focused works throughout §5.3.

Separation is effective but vulnerable. Concrete impact on bug exploitability has been documented where compartmentalization was pushed to production – most of Table 5.4. In the OpenBSD userland, the OpenSSH and slaacd sandbox compartments successfully mitigated code-execution flaws [20, 68, 15, 16]. Similar observations were made for Nginx workers [11, 18], and web browser site isolation. Still, the protection is not limitless: interface safety vulnerabilities have been reported against OpenSSH [67, 66, 9], Firefox [401, 268], Nginx [10], Chrome and generally site isolation [34]. These observations are more or less direct effects of the ad-hoc nature of deployed compartmentalization approaches. Although some works in §5.3 are concerned with this problem, most scope out interface safety to focus on performance. This constitutes a gap between mainstream needs and research trends.

Policy definition methods are mostly manual. For all of Table 5.4, separation boundaries are manually identified and maintained organically over time, a process costly in expertise and efforts [73, 7]. As discussed in §5.3.2, the engineering cost of manual separation limits achievable separation granularities (for nearly all 60 apps, separation is coarse with at most 2-3 domains), and makes separation generally less approachable by the the mainstream. Firefox library sandboxing, stemming from a research project, is the only case of a non-fully manual PDM [344].

Diverse abstractions & Focus on interfaces. Abstractions in Table 5.4 show a clear tendency towards sandboxing of untrusted code (compared to other models from §5.3.1). For the rest, abstractions feature an heterogeneous mix of spatial and temporal designs, synchronous and asynchronous semantics, and message-passing- and shared-memory-based communication. Overall great attention is dedicated towards interfaces. For instance OpenSSH leverages a custom protocol with fully serialized and checked objects [373], RLBox leverages type data to systematically check and copy objects, and

Nginx leverages a very thin interface with nearly no communication from the untrusted to the trusted world. The importance of interfaces is characterized by the dominance of message-based approaches, which ease the checking of interface-crossing data and control flows.

Importance of availability. Most designs in Table 5.4 target a degree of availability. This comes in contrast with research, which generally considers availability out of scope (§5.3.3). This may make it difficult to deploy many of the previously described approaches under real-world expectations.

Mechanisms are PT-centric. Nearly all designs from Table 5.4 are strongly dependent on the page table mechanism, an effect of their building atop `fork()` semantics. Regrettably, this dependency is hard to break [134], making it hard to reap the benefits of modern mechanisms (§5.3.4), as we discuss in the next section. The exception, Firefox library sandboxing, a product of modern research [344], supports flexible mechanisms with implementations on the page table and SFI.

5.5 Discussion: Outstanding Challenges

Challenge: creating and maintaining safe compartmentalization policies is still too hard. The skills required to design safe policies are very specific: ensuring interface safety, reasoning about the performance of separated software, maintaining separations over time; all constituting an art often mastered by trial and error. Compartmentalization cannot go mainstream expecting non-expert developers to acquire this art. As we show, this makes many approaches described in §5.3.3 rather unsuitable for that purpose. In fact, even when developers posses the required skills, compartmentalization policies are still overly hard to get right. For instance, most of the works described in §5.3.3 leave the job of securing internal and external interfaces, a complex and particularly error-prone task, entirely to the developer. As we observe in §5.4, some software projects have the skills, time, and budget to so, but this is not the case of the vast majority of the software ecosystem.

This calls, we argue, for concerted efforts in two directions. First, more work is needed *on approaches that do not require a policy from application developers*. This can be approached with a focus on third-party dependencies that have the skills and community, and where costs get amortized. Shared library and software package APIs, for instance, should be designed to be transparently distrusted from the ground on,

following the example of, e.g., V8 (§5.4). This can also be approached through more works on automated, generic compartmentalization, trading off security for deployability (§5.3.2). Second, more work is needed on *supporting the policy development process, for developers who have the skills to do so.* There is a need for PDMs that understand application and boundary semantics; for more tooling to integrate compartmentalization into long-term maintenance workflows to ensure safe separation over time; for more automated interface safety checking methods; and for more fuzzing efforts specifically targeted at the needs of compartmentalization.

Challenge: compartmentalization performance overheads are still too obscure. Compartmentalizing represents a high engineering effort which developers are unlikely to undertake without understanding how it will impact performance. Yet, it remains hard to estimate the performance cost of compartmentalization prior to implementation. The problem is emphasized by the relative obscurity of general separation overheads: research vastly focused on domain-switching latencies, but other less studied costs such as those induced by mechanisms (§5.3.4), the runtime costs of interface protection, of system call shielding, or of allocation hardening when heaps are shared, are also known to impact performance. We call for more efforts towards characterizing holistic performance costs of compartmentalization, and towards techniques and tooling to support developers in estimating, diagnosing, and optimizing compartmentalization performance costs early on.

Challenge: abstractions are still not future-proof. Most abstractions used in the mainstream are strongly dependent on processes and `fork()` semantics (§5.4). This is a problem because these semantics 1) do not compose, making it hard to use them in conjunction with new compartmentalization abstractions, and 2) hinder the ability to leverage new mechanisms, which are released at a fast pace (§5.3.4). Yet most new abstractions still do not compose, and are developed specialized towards particular mechanisms (§5.3.3). Will we repeat the mistake of `fork()` [134], requiring each codebase to implement several compartmentalization approaches? This calls for concerted efforts towards generic abstractions that map to the ecosystem of existing and future mechanisms, as well as for reflections on how abstractions should safely compose.

Challenge: threat models are insufficiently challenged. Nearly all compartmentalization works discussed in this study make assumptions about interfaces (sanely

checked, enforce semantics, thus free of high-level logic bugs), compilers (no correctness-security gap [190, 464]), shared components (libc, threading libraries, memory allocators, are bug-free when shared), the kernel (no confused deputies), or the hardware (the requirements of conditioned mechanisms are properly satisfied, no transient execution flaws or side-channels). These assumptions do not hold in practice. Interfaces are porous and abstractions must take structural measures to ensure safety by definition [397, 173, 298]. Compilers [436, 260, 259], shared components/kernels [173, 442], hardware [274, 354] all showed prone to separation-threatening flaws. This calls for more offensive research exploring gaps in compartmentalization threat models and characterizing their impact, and defensive works on holistic threat models and systematic counter-measures.

5.6 Related Works

Shu et al. [409] surveys general isolation, and Acar et al. [110] systematizes general Android security research, including application compartmentalization. Both feature a more general scope than this SoK (cf. §5.2) and thus do not cover software compartmentalization as systematically. Both also pre-date many recent works, as most of Tables 5.1 and 5.2 were published after 2016.

Other works cover vulnerability classes mitigated by compartmentalization like memory safety [421], side-channels [156], and supply-chain attacks [289], as well as mechanisms suitable for compartmentalization [157, 246, 257, 358]. These works are orthogonal to this paper. Other works [163, 244, 270, 298] classify interface safety issues and mitigations. These complement this SoK, which draws a bigger picture of compartmentalization challenges. Sammler et al. [393] models sandboxing to prove safety properties. These efforts motivate this SoK, which confirms the validity as well as the limitations of their model.

5.7 Conclusion

Despite its benefits and decades of research and industry works, compartmentalization remains a niche software engineering practice. Through a systematic study of 192 software compartmentalization works and 60 deployed approaches, this paper shed a light on the strengths and limitations of current compartmentalization knowledge. We stress that popularizing software compartmentalization will require progress towards eliminating (and, when relevant, supporting) the definition of compartmentalization

policies; towards better framing separation costs early on; to designing abstractions that will stand the test of time and progress; and to better challenging our threat models, particularly in light of interface safety issues.

Acknowledgements

We thank the anonymous reviewers for their insights. We are also grateful to Shravan Narayan for their insightful feedback. This work was partly funded by a studentship from NEC Labs Europe, a Microsoft Research PhD Fellowship, the UK's EPSRC grants EP/V012134/1 (UniFaaS), EP/V000225/1 (SCorCH), the EPSRC/Innovate UK grant EP/X015610/1 (FlexCap), and the NSF CNS #2008867, #2146537, and ONR N00014-22-1-2057 grants.

5.8 Appendix: A Glossary of Compartmentalization

This appendix groups and expands the compartmentalization terminology used throughout the paper as well as in the literature. We hope that these definitions will be useful for young researchers starting in the field of compartmentalization. This content is strictly provided as an independent addition to the paper and is in no way necessary to understand it.

We base our proposed compartmentalization terminology on slightly modified definitions of *subject*, *object*, *permissions* and *privilege* from Miller [339] and Saltzer and Schroeder [392], given next:

- A subject is a unit of computation that *may be given distinct access rights*. Examples of subjects: a thread of execution, a region of code, an assembly instruction.
- An object is the smallest unit *to which* distinct direct access rights may be provided. Examples of objects: bytes or pages of virtual memory, instances of OS abstractions (e.g., files, sockets).
- A permission is an action a subject can perform on an object it can *directly* access. In the case of a byte of virtual memory, a permission can be the ability to read/write it. For a file, it can be the ability to open or close it.
- A privilege is an action a subject can cause on objects it can directly *or indirectly* access. For instance, a subject with write *permissions* to `/proc/self/mem` has *privileges* over the entire address space. We formally refer to the privileges of a subject

s to an object o as $priv(s, o)$.

- Enforcing least privilege is the action of granting each subject only the privileges it requires to operate.

Based on this, we define compartmentalization notions:

- An object o is called shared object between two subjects s_1 and s_2 when both s_1 and s_2 have privileges over o, i.e., $priv(s_1, o) \neq \emptyset$ and $priv(s_2, o) \neq \emptyset$.
- A protection domain is a maximal set of subjects sharing identical permissions. Examples of protection domains: POSIX (or Linux) processes are generally protection domains. Traditional execution threads are *not* protection domains, as they cannot be associated with distinct permissions (they have the permissions of their respective process, and thus do not qualify as *maximal* set).
- A compartmentalization of a program P is the set of (1) a policy to separate P into two or more protection domains (which, in this context, are called compartments), and (2) the enforcement of this policy at runtime.

  This definition refers to software compartmentalization as partitioning *within* individual programs, and thus does not include separating programs or groups of programs from each other. This captures the modern use of software compartmentalization, and as we show in this paper, a vast and coherent body of work.
- Privilege separation [373] has been used as a synonym to our definition of software compartmentalization. However, it is also used [409] with a more generic meaning as defined by Saltzer [392] (*separation of privileges*), referring to the practice of designing systems as sets of partitioned components on which least privilege is applied. This definition goes beyond program separation to include containers, application groups, user/kernel separation, etc.
- The complete isolation [392] (or *total confinement* [291]) of parts of a program is the special case of compartmentalization without shared objects. Formally: compartment P_x is isolated $\Leftrightarrow$ for all subjects $s \in P_x, s' \in P \setminus P_x$, and for all objects $o : priv(s, o) \neq \emptyset \Rightarrow priv(s', o) = \emptyset$. BPF packet filtering [332] is an example of (near-)complete isolation.

  Note that *isolation* in general is an ambiguous term. Some works [409] use it to designate general privilege separation including containers, application groups,

user/kernel separation, etc. Yet others [299, 248] use it as synonym of software compartmentalization. Hence, we avoid this term throughout the paper.

- The compartmentalization space [227] (also called *compartmentalization continuum* [384]) is the set of all possible compartmentalizations of a program. Compartmentalizations can differ in the choice of subjects and objects, in what permissions they are granted, and how these permissions are enforced.

- In a compartmentalized program, a sandbox is a compartment whose privileges are minimized to protect the rest of the system.

  The term *sandbox* is also used outside of compartmentalization when referring to reducing the privileges of whole applications [192], e.g., with seccomp [82, 92, 101], or language-based techniques such as applets [217]. This usage is not compartmentalization as we scope it.

- A Compartment-Interface Vulnerability [298, 168] (CIV) is an interface safety vulnerability at a compartment interface. CIVs arise due to inadequate validation of control flows and data flows at compartment boundaries. They may affect internal interfaces (intra-program), as well as external interfaces (with other system components, e.g., the kernel, or other applications).

  CIVs are more commonly referred to as *confused deputies* [384, 385, 269], although this usage is somewhat ambiguous [110]: classical confused deputies [234] refer to a privileged component C_p (the confused deputy) performing a malicious action on a victim component C_v on behalf of an unprivileged malicious component C_m, due to insufficient validation of C_m's capabilities by C_p. Many CIVs follow a pattern where $C_p = C_v$ [298], which is not covered by the standard definition of a confused deputy [234]. Still, since *confused deputy* is a more common term, we adopted it throughout this paper.

  Previous works also referred to certain specific classes of CIVs as *Dereferences Under Influence* [244] (DUI) [298]. Interface vulnerabilities described in the TEE community (Iago [161], COIN [270], both specialized instances of the confused deputy problem at the system call interface) are also relevant to compartmentalization and may be viewed as part of the CIV-spectrum.

5.9 Appendix: Taxonomy Addendum (§5.3)

This section expands on the methodology adopted in §5.3. This information is provided as an independent addition and is not necessary to understand the paper.

Systematic Search (Precise) Inclusion Criteria. We include a paper if it concerns software compartmentalization, as defined in §5.2. Venues systematically filtered are ranked A* by CORE 2023[5] in security, systems, and programming languages, namely: S&P, USENIX Security, CCS, NDSS, ASPLOS, OSDI, SOSP, PLDI. The manual triage of titles and abstracts is performed in an ablative, conservative manner: when processing titles, papers which are clearly unrelated to compartmentalization are removed from the list. If the title does not allow making an unambiguous decision, the paper is moved to the next stage (triage of abstracts). Abstracts were sufficient to make an unambiguous decision in the vast majority of cases. We consider a paper for Tables 5.1 to 5.3 if it addresses at least one of P1, P2, or P3 (cf. §5.3). This is determined by analyzing the content of the paper.

Other Venues. We recognize that researchers also published influential works before 2003, in other venues (e.g., CORE A-ranked), and/or types of venues (journals, technical reports, theses), in neighboring communities (e.g., embedded systems, software engineering, networking, architecture, mobile computing, formal methods), as well as in the industry. We aim to cover these influential works. Still, extending the systematic search to all these sources is untenable, and may ultimately lower the quality of the study. We address this with our recursive reference search, and by factoring in works from our knowledge of the field. To perform the recursive reference search, we extract the list of all papers cited by the 89 relevant works identified in the systematic literature search. We then perform the same ablative and conservative procedure on this list as for the systematic search (title, then abstracts, see previous paragraph).

As a side note, though the general privilege separation literature before 2003 is rich, we find that works that qualify as software compartmentalization are not numerous.

Effort. Although the total number of papers systematically reviewed is high (9083 papers, not including the reference search), the effort to process it title-wise remained reasonable. Overall, provided appropriate tooling, it takes little more than a few seconds for an expert to conservatively whitelist or reject a paper by looking at the title.

[5] http://portal.core.edu.au/conf-ranks/?search=&by=all&source=CORE2023

Similarly, processing abstracts (of 923 potentially related papers) is a relatively quick task for an expert. Overall, the effort of carefully reading, categorizing, systematizing, and reflecting on the remaining 192 papers represents the vast majority of the efforts and complexity of writing such a study.

5.10 Appendix: Source Search Addendum (§5.4)

This section expands on the methodology adopted in §5.4. This information is provided as an independent addition and is not necessary to understand the paper.

Systematic Source Search. We automatically filter the source code of 1520 Debian 13 packages with >1K installations as per Debian Popularity Contest (Popcon) [27], according to the following keywords: *sandbox, privilege, isolat*, separat*, compartment, partition, domain, capabilit** (and derived words; partition*(ed|ing)*, separat*(ed|ion)*, etc.). This yields 361 packages, which we manually triage to assert whether or not they implement compartmentalization. This search yields 18 packages. The large number of false positives is composed of software that declare a sandboxing profile (whole application sandboxing [192, 82]), drop privileges (whole application least privilege), implement internal user access control policies that do not qualify as compartmentalization (e.g., databases), or use isolation or separation keywords to refer to object-oriented development practices (e.g., "isolate a component in a class", "protected" methods in C++). We look up the vendor website of each of the 18 packages for other software from the same vendor that may also adopt a compartmentalized design (not found in our initial search due to being less popular or not packaged for Debian), resulting in 37 more compartmentalized software packages, most coming from the OpenBSD userland, for a total of 55 packages. We complete this list with previous works (§5.3, 3 apps), and our knowledge of the field (2 apps), to reach a corpus of 60 compartmentalized programs.

Choice of the Debian Archive. We choose the Debian archive because it 1) covers software from all origins (e.g., independent developers, academics, commercial, as well as software from other communities such as OpenBSD), 2) is very large (59K+ packages [26]), and 3) comes with Popcon [27], a package popularity metric which maps well to our "mainstream" criteria necessary to make the search practicable.

We recognize that a systematic search in the Debian archive does not cover compartmentalized software in the mobile ecosystem (e.g., Android or iOS). Beyond the

difficulty to integrate it in the paper's space, a systematic search of mobile applications for compartmentalization patterns is generally difficult due to the unavailability of sources in popular application stores. We therefore leave it for future works.

Detailed Program List. We provide the full list of programs, which, for space reasons, was grouped in classes in Table 5.4 (that includes a shortened list, particularly for the OpenBSD distribution). In the following, programs found through our systematic search of Debian archive are labeled [S], those found through the related works with [R], and those added from our knowledge of the field with [K]. For the OpenBSD userland, we give program names as they appear in OpenBSD.

- *OpenBSD Privilege-Separated Userland*: openssh[S], bgpd[S] (openbgpd), dhclient[S], dhcpd[S], dvmrpd[S], eigrpd[S], file[S], httpd[S], iked[S], ldapd[S], ldpd[S], mountd[S], npppd[S], ntpd[S] (openntpd), ospfd[S], ospf6d[S], pflogd[S], radiusd[S], relayd[S], ripd[S], script[S], smtpd[S] (opensmtpd), syslogd[S], tcpdump[S], tmux[S], xconsole[S], xdm[S], Xserver[S] (Xenocara), ypldap[S], pkg_add[S], xlock[S], snmpd[S], dhcrelay[S], rbootd[S], pppoe[S], mopd[S], afsd[S], rdate[S] (openrdate), sndiod[S], isakmpd[S], named[S], acme-client[S].
- *Web Servers*: Apache HTTPd[S], Nginx[S], Lighttpd[S]
- *Browser Site Isolation*: Chrome[S], Firefox[S], Epiphany[S] (all similar, with technical nuances in each implementation)
- *Separated mail transfer agents (and inspired approaches)*: qmail[K], Postfix[S], djbdns[K]
- *μkernels*: MINIX[R], L3/L4 family[R], and many others.
- *Single-application classes*: IRSSI[S], Debian man[S], DHCPCD[S], VSFTPD[S], Firefox library sandboxing[R], Google V8[S] (libnode), Dovecot[S].

Note that some works labeled [S] have also been found through the literature and/or our own knowledge, e.g., openssh, browser site isolation, Firefox library sandboxing. In this list we follow the order of precedence [S], [R], [K].

Chapter 6

Conclusions and Future Research

To outrace attackers, we should design applications assuming their different parts can be compromised. *Software compartmentalisation* is one approach to achieve this. By architecting applications as groups of isolated and distrusting compartments, as opposed to a single privileged monolith as is the industry standard, we can proactively limit the impact of software vulnerabilities. Over decades of research and practice, software compartmentalisation has shown capable of mitigating memory safety issues, supply-chain attacks, and many other classes of compromises. These benefits are providential amid a time of heightened concern for the security of states, businesses, and individuals.

Yet, software compartmentalisation is still not a widespread practice. As showed in Section 5.4, little more than 55 out of 1520 popular applications from the Debian archive adopt a compartmentalised design. Even when it comes to isolating cryptographic secrets, a rather obvious case for compartmentalisation long pushed by many works [258, 225, 395, 227, 133, 315, 434, 147, 166, 141, 415, 226, 296, 313], approaches proposed in research are not adopted in practice by popular cryptography libraries. We are missing out on the security benefits compartmentalisation can bring, at a time where systems security has become more important than ever.

This thesis contributed three elements towards addressing this problem: 1) showing that the performance cost of compartmentalisation and its security benefits can be optimised through full-system specialisation; 2) furthering our understanding of the new classes of security vulnerabilities that arise in compartmentalised systems and how we should go about to solve them; and 3) bringing a global overview on compartmentalisation approaches, highlighting gaps, mainstream needs, and future challenges.

In Chapter 3, we proposed FlexOS, a modular operating system which enables users

to easily implement highly-specialised compartmentalisation policies. We showed that the flexible architecture of FlexOS opens for a vast design space, which makes it possible to reduce overheads by specialising the protection profile of the OS from the ground-up to reach the most appropriate security/performance trade-offs. Further, we showed that automated methods such as our partial safety ordering approach can be used to explore the design space automatically, making these benefits accessible to non-expert users.

In Chapter 4, we presented a study of interface vulnerabilities in compartmentalised software. To enable this study, we proposed ConfFuzz, a fuzzer designed specifically to detect Compartment-Interface Vulnerabilities (CIVs) at possible compartment boundaries in applications. Applied to 39 real-world compartmentalisation scenarios, we constituted a data set of 629 potential CIVs, which we dissected to extract insights on their prevalence, their causes, impact, and the complexity to address them. We stressed the critical importance of CIVs in achieving strong security properties with compartmentalisation, demonstrating an attack to extract isolated keys in OpenSSL and uncovering a decade-old vulnerability [19] in sudo. Throughout this chapter we showed, among others, that not all interfaces are similarly affected, that API size is uncorrelated with interface weakness, and that addressing interface vulnerabilities goes beyond writing simple checks, concluding with guidelines to design stronger compartment interfaces.

Lastly, in Chapter 5, we presented a systematisation of knowledge study in the field of software compartmentalisation. We proposed a theoretical framework for compartmentalisation, characterised existing approaches into a taxonomy that highlights solved problems and gaps in our knowledge, and systematises mainstream needs and how research approaches address (or do not address) them. We concluded the chapter with insights on future compartmentalisation challenges: among others, we stressed that popularising this practice will require progress towards eliminating the need for developers to manually define compartmentalisation policies (or, when relevant, helping them doing so); towards better framing separation costs early on; to designing abstractions that will stand the test of time and progress; and to better challenging our threat models, particularly in light of interface safety issues.

6.1 Limitations and Future Research

To wrap up, we will discuss limitations of the results presented in this dissertation, how these limitations call for more research, and how this research relates to other works we conducted in parallel to this thesis. This complements §5.5, which provided a higher-level perspective on the field's next steps towards mainstream compartmentalisation.

Chapter 3: Large-scale design-space exploration. Section 3.5 presented our *partial safety ordering* method to semi-automatically explore FlexOS design spaces. This method makes a few simplifications. First, although it is effective for the configuration spaces we explored with FlexOS, this method will remain bound to state explosion when large numbers of compartments (or hardening mechanisms) are considered. Another limitation is the method's consideration of performance as a unique and final value – a simplification given that performance varies across workloads, and that several performance metrics may make sense for a same deployment (e.g., throughput, tail latency, memory usage). Overall, these limitations call for more research in design-space exploration methods that can cope with the complexity of the larger-scale spaces we are advocating for in the long term. Together with my co-authors, we are exploring these topics as part of our continuing efforts on the Wayfinder [261] project, where we apply machine-learning techniques to explore OS configuration spaces.

Chapter 4: Avenues in CIV fuzzing. Chapter 4 presented ConfFuzz, a fuzzer designed specifically to detect CIVs in unmodified applications. As a fuzzing contribution, ConfFuzz presents a number of limitations. First, ConfFuzz does not cover the full spectrum of CIVs described in Section 4.3. Data leakages (§4.3.1) are not covered due to the lack of appropriate detectors, and temporal violations (§4.3.3) are only partially covered. Second, ConfFuzz relies on relatively simple, per-type semi-random data alteration strategies. Although these strategies suffice for the scope of Chapter 4, they leave significant room for improvement in increasing the coverage of CIV fuzzers. These limitations call for more work in the field of fuzzing to target the specificities of CIVs and compartmentalised applications, covering more types of vulnerabilities, faster.

Chapter 4: *Complete* CIV analysis. Because it is a fuzzing technique, ConfFuzz is fundamentally incomplete: there is no guarantee that ConfFuzz or any other fuzzer will find all CIVs present at a given compartment interface. This is an acceptable limitation for the needs of Chapter 4. Still, this lack of completeness makes ConfFuzz (and fuzzers in general) unsuitable for other purposes, such as comparing interface strength. For example, problems of the form "is interface I_a stronger or weaker than interface I_b w.r.t. CIVs" are commonplace when partitioning software: there are generally many ways to partition a program at a similar performance cost, which differ in their vulnerability to CIVs. Future work should explore complete CIV analysis methods to cover these use-cases. Together with my co-authors, we are exploring these topics as part of the CIVSCOPE [168] project, where we apply static analysis techniques to systematically detect CIVs and automatically evaluate and compare the vulnerability of interfaces.

Chapters 4 and 5: Addressing Interface Vulnerabilities. Chapters 4 and 5 furthered our understanding of CIVs as one of the next major problems to solve in software compartmentalisation. Overall, both chapters call for more research towards designing mechanisms, abstractions, and partitioning methods with CIVs in mind; as well as tools and methods to refactor existing interfaces or design them to be safe in the first place.

One such direction to explore is to expand and generalise the guidelines we provided in Section 4.6 into a full-fledged set of "compartmentalisation design patterns", akin to traditional software design patterns [209]. Looking long-term, such a contribution would be invaluable in teaching researchers and practitioners how to design software for distrust; an art which, to this day, is still widely mastered by trial and error.[1] Together with my co-authors, we are conducting first steps in this direction by confronting the guidelines from Section 4.6 to real-world interfaces, as part of an effort to define safe by construction virtual I/O host-guest device boundaries [301] (paper attached in Appendix B).

Chapter 5: Other approaches to systematisation. Chapter 5 aimed to further our understanding of the next most important challenges in software compartmentalisation. We performed a large-scale survey of existing approaches, and systematised them. Based on the resulting overview, we identified limitations and knowledge gaps, which we prioritised according to our expertise. One limitation of this approach is that our understanding of the field *as researchers*, and particularly of the constraints that rule the deployment of compartmentalisation in real-world environments, may differ from that of industry practitioners. To address this limitation, future works may take a "practitioner-centric" perspective on the question by surveying the developers themselves. This study should provide another perspective on the constraints that rule the deployment of software compartmentalisation, and on the next steps, as perceived by the practitioners themselves, needed to loosen these constraints. Such a contribution would be complementary to this thesis, since the perspective of practitioners remains itself subject to biases, and should be counterbalanced with a systematic view of existing approaches and of the literature, as provided in Chapter 5, to obtain a full picture.

[1] Note that the historical motivation for software design patterns precisely was to *provide a means to reuse the design knowledge gained by experienced practitioners* [209], which echoes our experience with compartment interface design.

6.2 Conclusion

In this dissertation, we contributed three elements towards bringing the benefits of software compartmentalisation to the mainstream. In a first part, we showed that compartmentalisation overheads can be significantly reduced through full-system specialisation. This contribution showed the potential of specialisation to meet ambitious performance targets, and motivated for more research in design-space exploration to leverage this potential. We expect that specialised compartmentalised systems will find applications together with the popularisation of specialised cloud appliances [285, 108]. In a second part, we furthered the field's understanding of the new classes of security vulnerabilities that arise in compartmentalised systems, and how we should go about to solve them. Beyond shedding light on CIVs, an important and often ignored class of vulnerabilities, this work motivated for new research directions in further understanding, detecting, and mitigating these vulnerabilities. Finally, we brought a global overview on compartmentalisation approaches, highlighting gaps, mainstream needs, and future challenges. This systematisation effort should provide a reliable basis for entire lines of research towards bringing software compartmentalisation to the mainstream. Looking forward, the need for compartmentalisation can only increase. On the long run, we hope that this thesis will contribute to achieving compartmentalisation approaches that are safer, faster, and more usable.

Appendix A

Loupe: Driving the Development of OS Compatibility Layers

This chapter presents Loupe, a systematic method to guide and optimize the developement of new research OSes. Loupe leverages dynamic analysis to determine the OS feature set that must be implemented by a prototype OS to support applications at different levels. Using Loupe, we perform a study of 100+ applications and several OSes under development to extract many new insights on OS development practices.

These contributions address a "meta-problem" to this thesis. As we showed throughout this thesis, building new operating systems is fundamental to bringing significant advances to the state of the art in application security and performance. Yet, building OSes that can run real-world applications comes at the cost of a daunting amount of engineering. To simplify and streamline this development effort, we envisioned to develop a method that would provide a systematic and pragmatic approach to writing OSes with application compatibility layers. The result of this effort is Loupe, which we used to accelerate the development of Unikraft [285] and FlexOS [299]. The contributions presented in this Appendix are derived from Lefeuvre et al. 2024 [302].

Contributions of the Author

I designed and implemented Loupe, the analysis method presented in this paper. Using Loupe, I gathered part of the dynamic analysis data presented in the paper. I analysed the vast majority of the data presented, and wrote the majority of the paper. Gaulthier Gain developed the static analysis tool used as baseline in Appendix A.5, and used it to gather static analysis data. Vlad-Andrei Bădoiu developed the integration of Loupe

with Debhelper, documented in Appendix A.3.3. Gaulthier Gain, Vlad-Andrei Bădoiu, Daniel Dinca, and Vlad-Radu Schiller helped me gather dynamic analysis results, running Loupe on part of the 100+ applications considered in the study. Costin Raiciu provided feedback on the paper. Felipe Huici provided feedback and edits on the paper. Pierre Olivier provided feedback and key edits on various part of the paper. Pierre Olivier also proposed the support plan method presented in A.4.

Abstract

Supporting mainstream applications is fundamental for a new OS to have impact. It is generally achieved by developing a layer of compatibility allowing applications developed for a mainstream OS such as Linux to run unmodified on the new OS. Building such a layer, as we show, results in large engineering inefficiencies due to the lack of efficient methods to precisely measure the OS features required by a set of applications.

We propose Loupe, a novel method based on dynamic analysis that determines the OS features that need to be implemented in a prototype OS to bring support for a target set of applications and workloads. Loupe guides and boosts OS developers as they build compatibility layers, prioritizing which features to implement in order to quickly support many applications as early as possible. We apply our methodology to 100+ applications and several OSes currently under development, demonstrating high engineering effort savings vs. existing approaches: for example, for half of the 62 applications supported by the OSv kernel, we show that using Loupe, would have required implementing only 37 system calls vs. 92 for the non-systematic process followed by OSv developers.

We study our measurements and extract novel key insights. Overall, we show that the burden of building compatibility layers is significantly less than what previous works suggest: in some cases, only as few as 20% of system calls reported by static analysis, and 50% of those reported by naive dynamic analysis need an implementation for an application to successfully run standard benchmarks.

A.1 Introduction

An operating system is only as useful as the applications it can run. Thus, developers of new OSes seeking to gather early performance numbers, to attract open source contributors, early investors, or to transition to real-world use [242, 368] need to provide support for existing applications. Manually porting software [71, 79] is only viable in the short term [352, 353], hence developers of new and existing OSes must provide support for unmodified software by building compatibility layers [119, 430, 432, 352, 353, 285, 36, 52, 39, 276, 102, 54, 76, 428, 99, 74, 375, 105] that present applications with interfaces similar to that of popular OSes such as POSIX or the Linux kernel ABI.

Building a compatibility layer represents a non-negligible engineering effort [368, 242, 119, 265, 352, 353, 285, 286] and involves 1) identifying the OS features (system calls, pseudo-files) required for a target application and 2) implementing these features. This process is iteratively repeated for each application to support. In this paper we focus on streamlining 1), the latter being generally OS-specific [119].

We observe that, despite their cost, compatibility layers are often developed in an ad-hoc fashion [119]: there is no systematic approach to determine and prioritize what OS features to develop and when, which applications to support, or to what extent used system calls should be implemented to achieve a desired degree of support. This results in a significant amount of unnecessary engineering.

Past attempts at streamlining that process leverage static analysis [431] and suffer from its drawbacks, heavily overestimating the set of OS features required to support an application. For instance, while binary-level static analysis identifies that >100 system calls are required to conservatively support the entire superset of operations, configurations, and error handling code in Redis (much of which can be quite rarely used in practice, or simply irrelevant for an early prototype), we find that only 42 are actually needed to reliably pass its entire test suite, and just 20 to run `redis-benchmark`.

Hence, OS designers often fall back to naive dynamic analysis, e.g., using `strace`. These tools fail to take into account common practices used in early OS development to save engineering effort: feature stubbing (returning `-ENOSYS` [32] upon invocation, without implementing the feature), faking feature success (returning a success code without implementing the feature), and partial implementation of complex features [242, 352]. Indeed, in early development, the goal is not to support every feature but rather core functionalities of target applications [119]. For example, we find that more than half of the system calls invoked by Redis running the `redis-benchmark` can be stubbed or faked, and do not need to be implemented to support that application and workload.

We propose a systematic methodology based on dynamic analysis, centered around a novel tool called *Loupe*. Loupe measures, for an application and a given input workload (e.g., a benchmark, test suite), which OS features really need to be implemented and which ones can be faked, stubbed, or partially implemented. Loupe also computes, given an OS under construction and a set of applications and workloads, an optimized development plan to support as many applications as possible with as little engineering effort as possible.

Dynamic analysis comes with its own challenges, in particular the difficulty to scale to numerous applications. This is tackled by designing Loupe to require as little effort as possible to integrate a new application, letting us present results for more than 100 applications in our evaluation. Another challenge is how to detect OS features that can be stubbed, faked, or partially implemented. This is addressed by leveraging Linux's `seccomp` [82] and `ptrace` [75] tracing and interposition facilities to measure what OS features' implementation can be avoided with these techniques.

We run Loupe on 100+ popular applications, and present examples of optimized Linux compatibility layer development plans obtained with Loupe for three OSes under construction [36, 52, 285] with various levels of existing support for the Linux system call ABI. We further measure the engineering effort savings obtained by using Loupe to drive the development of compatibility layers. Taking half the applications supported by OSv [276], Loupe reports that only 37 system calls are required to run them, vs. 92 for our estimation of the non-systematic process followed by OSv developers, and 142 for a process driven by `strace`-based dynamic analysis.

We study Loupe's Linux API usage measurements for our set of applications. This analysis brings many new insights. We demonstrate that the minimal effort needed to provide compatibility is significantly lower than that determined by previous works using static analysis [431]. Our study shows that as much as 40-60% of system calls found in application code do not need implementation to successfully run meaningful workloads, including full test suites. We also find that many applications are resilient to stubbing, faking, and partial implementation of OS features. We investigate the reasons behind it, and the impact of such practices on application performance and resource usage. Finally, we study how the C library influences OS feature requirements.

In all, this paper makes the following contributions:

- A novel methodology to measure the minimum set of OS features that need implementation for a compatibility layer to support a set of applications and workloads, with the aim of minimizing development effort.

- Loupe, a tool able to derive, for a given OS and target applications, an optimized OS feature support plan to run as many apps as possible, as early as possible.
- A demonstration of engineering effort savings obtained with Loupe, with examples of optimized feature implementation plans for 11 OSes under development.
- An analysis, using Loupe, of the OS features required by a set of applications showing the lack of precision of past approaches and investigating common development practices in compatibility layer development.

Loupe is actively used in Unikraft [285], an open-source commercial OS, and has attracted the attention of several others. Overall, this study brings a message of hope: contrary to what past work seems to suggest, a good degree of compatibility with existing applications can be achieved without immense engineering, provided we follow a focused and methodical approach.

A.2 Motivation and Approach

Building Compatibility Layers for New OSes. Compatibility layers can be found in mature OSes for interoperability reasons [428, 99, 74, 54, 105], but also in a plethora of new/prototype/research OSes [119, 430, 432, 352, 353, 285, 36, 52, 39, 276, 102, 76, 375]. Providing support for existing applications in these OSes is generally crucial [352, 353, 285, 303] to gather early performance numbers, to attract open source contributors, early investors, or transition to real-world use. Manually porting software [71, 79] is not sustainable in the long run, nor does it scale to a large amount of applications [352, 353]. Hence, the developers of many new OSes resort to implementing compatibility layers. Even considering OS models that choose to drop application compatibility for other gains (e.g., performance), it is not uncommon to see Linux versions of these models appear a few years after the seminal paper, with claims of stronger compatibility, e.g., Popcorn Linux [130] for the multikernel [135] or Graphene, Lupine and UKL [430, 286, 376] for the unikernel [323, 329].

Building a compatibility layer is seen as a non-negligible engineering effort [368, 242, 119, 265, 352, 353, 285, 286, 431]. We investigated the compatibility layers present in several open-source OS projects [119, 430, 432, 352, 353, 285, 36, 52, 39, 276, 102, 76]. Based on this study, and on our multiple years of experience providing Linux/POSIX

compatibility in research OSes, we observe that compatibility layers are built in an ad-hoc, non-systematic ("organic") way: developers select an application to support, determine the OS features it requires, and implement them [119]. That process is repeated for each target application. Because so many projects undergo the task of building compatibility layers [119, 430, 432, 352, 353, 285, 36, 52, 39, 276, 102, 76, 428, 99, 74, 54], there is a need for tools to streamline that process. The corresponding effort consists in 1) identifying OS features required by target applications and 2) implementing these features. The latter task is known to be very specific to the new OS considered [119], and can hardly be streamlined. We show in this paper that the former task, identifying and prioritizing what OS features to implement, can be systematized and optimized. Next, we motivate our method by explaining how past and current approaches are suboptimal.

Limitations of Static Analysis. Existing approaches measuring the usage of OS features by applications often rely on static analysis [431, 124, 352, 353, 212, 185, 152, 444, 357, 149]. Static analysis is comprehensive: the set of features identified for an application includes all the ones that *may* be invoked at runtime, under any possible workload, operation, or configuration, and traversing any possible error path. Alas, static analysis is also conservative and yields many false positives: it overestimates OS features that will actually be invoked at runtime.

Static analysis can be performed on application sources or binaries. Binary analysis [431, 124, 352, 353, 185, 152] scales well to a large number of applications because it targets a common format (e.g., ELF binaries). However, it suffers from a lack of precision due to the difficulty of extracting information from a binary [185]. Such issues may be alleviated with source-level analysis [212, 444], which is not a panacea either: being language-specific, source-level analysis does not scale well to many applications written in different languages.

Tsai et al. [431] measure, using static binary analysis, the system call usage of the entire set of applications from an Ubuntu distribution. The study concludes that to support 100% of the distribution's packages, 272 system calls need to be implemented. That number goes down to 81 system calls for the 10% most popular applications. These results suggest that a large implementation effort would be required for an OS aiming at supporting even a few applications. As we demonstrate in the evaluation, both source- and binary-level approaches significantly overestimate the OS features required by an application to run popular workloads. This is due to dead or unexecuted code, and the difficulty or impossibility to statically determine runtime-level information (e.g., memory content such as function pointers). Though all of these system calls would need to

```c
long sys_sigaltstack(const stack_t *ss, stack_t *oss) {
  return -ENOSYS; // stubbed: not supported
}

long sys_mprotect(size_t addr, size_t len, uint64_t prot) {
  return 0; // faked: pretending success
}
```

Listing 10: System call stubbing and faking examples from the HermiTux unikernel [352], where `sigaltstack` is stubbed, and `mprotect` is faked.

be implemented in a production-grade general-purpose OS, these numbers remain an upper bound of limited usefulness for OS designers in earlier development stages.

Limitations of Naive Dynamic Analysis. Dynamic analysis too has well-known drawbacks. Its precision depends on the coverage of the input workload run during the analysis: if it is too low, some required OS features may not be identified. It is also harder to fully automate, as there is a variable amount of manual effort required for each application to analyze (e.g., selecting an input workload). In this paper we refer to using a tool such as `strace` [87] to trace OS features invoked by an application, as *naive* dynamic analysis. The main drawback of naive dynamic analysis is its failure to consider two techniques commonly used in early OS development [242]:

- Feature *stubbing*: not implementing the feature and returning an error code (`-ENOSYS`: "Not Implemented" [32]) to the application when it invokes the feature.
- *Faking* feature success: not implementing the feature and returning a success code (typically system-call specific) to the application upon invocation.

These two techniques are illustrated in Listing 10, extracted from the source code of the HermiTux unikernel [352], where the `sigaltstack` system call is stubbed, and the `mprotect` system call is faked.

Many applications are resilient to the failure of OS features [242, 382] and will run correctly when stubbing and faking. In this study, we show that many invoked OS features can avoid being implemented through these practices in the development stages of an OS. This highlights the importance of faking and stubbing as an engineering practice: without it, showcasing a particular application use-case for a new OS concept would take significantly longer, or even be unattainable for a small-scale research project. Despite of this, naive dynamic analysis does not typically consider stubbing

and faking. Naive dynamic analysis traces all features and sub-features invoked by an application, independently of the fact that they can be stubbed/faked or not for a given workload. Thus, OS designers typically rely on trial and error to determine which features they need to implement first, and which ones they can fake or stub.

When to Stub or Fake and When not To? The reliance on stubbing and faking as a development practice in transitional OS development stages introduces a pivotal question: *when to stub and fake, and when not to?* Two sources of concern drive this question:

- *Impacting stability.* Though guaranteed stability of full applications is not a primary goal in the early development stages of an OS, faking and stubbing must not impact the stability of relevant application features. Failing to do so would negate the benefits of faking and stubbing by creating an additional debugging cost.
- *Impacting performance metrics.* Early OS prototypes must be comparable to full-fledged mainstream OSes; this is especially true for research OSes. Impacting performance metrics by faking or stubbing would defeat the purpose of the OS prototype by making it impossible to fairly evaluate its performance advantage or cost. For instance, stubbing or faking an expensive and relevant security feature may provide an unfair advantage to an early OS prototype vs. a full-fledged OS that implements it.

Non-systematic, trial-and-error-based approaches are especially prone to fall into stability and performance pitfalls. Although important, these concerns have been little discussed by works which rely or relied on faking and stubbing.

Breaking the Status Quo with Loupe. We aim to propose a systematic and adaptive method to determine which OS features to implement first. Our goal is to help OS designers transition from *no support* towards *full support* to run as many applications as early as possible.

Overall, dynamic analysis is better suited to the problem we aim to solve, being able to evaluate the concrete impact of both stubbing and faking, and providing fine-grain, per-workload results. The coverage issue of dynamic analysis is a nonproblem in our context: in early development phases of an OS, the goal is not to support every feature but rather core functionalities of target applications [119], which are easily exercised by standard benchmarks and test suites. Dynamic analysis is precisely suited because it is adaptive: as support progresses, workloads can be extended to cover more and

more application features. We are left with two challenges: the difficulty to 1) scale to numerous applications, and to 2) detect OS features that can be stubbed, faked, or partially implemented without impacting stability or performance for relevant application features. As we detail in §A.3, we solve the former by designing Loupe to require as little effort as possible to integrate a new application (at most writing a Dockerfile and test script), letting us present results for 100+ applications in §A.5. We address the latter challenge by leveraging Linux's `seccomp` [82] and `ptrace` [75] facilities to measure what OS features can be stubbed, faked, or partially implemented for a given application workload. To maintain stability and performance metrics for the evaluated workload, Loupe replicates the analysis several times in containerized environments, and offers a framework for identifying performance regressions on various generic and application-specific metrics. We further discuss stability in Appendices A.5.2 and A.6, and performance impact in Appendix A.5.3.

A.3 Accurate Run-time Analysis of OS Feature Usage

To accurately quantify what OS features are needed to gradually support a set of popular Linux applications, and to measure to what extent static and naive dynamic analysis overestimate these requirements, we built *Loupe*. Loupe is a dynamic analysis tool that hooks into each OS feature used by applications at runtime, analyzing the application's behavior as it simulates different degrees of compatibility. Unlike existing naive dynamic analysis tools (e.g., `strace`), Loupe is built as a framework specifically meant for OS feature support analysis. It supports identifying what system calls and pseudo-files are used by a given application, and determining which can be faked, stubbed, or partially implemented. Loupe focuses on reliable and reproducible results, and supports easy integration into existing build systems and complex test suite systems. Finally, Loupe can process measurement data for a set of applications and output targeted OS feature support plans. We implemented a prototype of Loupe in 2.5K LoC of Python, and 500 LoC of C (used for `seccomp` and `ptrace` hooks). In this section, we summarize the functioning and architecture of Loupe (§A.3.1), detail our approach to evaluate the success and performance of application runs (§A.3.2), and conclude with details on various aspects of Loupe (§A.3.3).

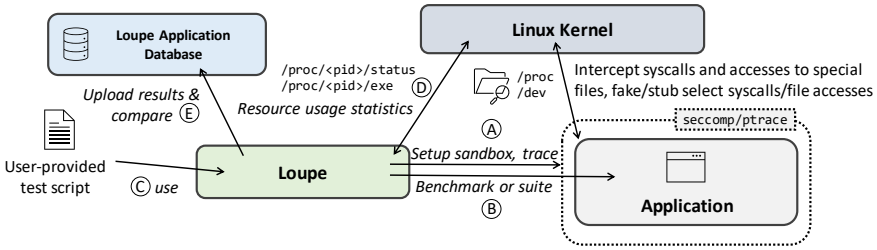


Figure A.1: Loupe architecture diagram.

A.3.1 Loupe Overview

Using Loupe to measure the OS feature usage of a new application is straightforward. Users provide Loupe with 1) the application binary whose OS feature usage needs to be measured, and 2) a per-application *test script*, responsible for providing external input to the application, and measuring the performance and success of each run. Loupe operates on binaries, so there are no language or compiler restrictions on either the application or the test script. Provided this, Loupe evaluates the OS feature usage of the application feature by feature. Each used system call and pseudo-file is tested for one of two modes in separate runs: 1) *stubbing* the system call, i.e., do not run the system call and return `-ENOSYS` or 2) *faking* it, i.e., return a success code without running the system call. Once all OS features have been tested, a final run confirms that the analysis performed on a per-feature basis holds when all features are considered. In the event of a failure, users can use Loupe to alter subsets of system calls to find the culprits, a process which could be automated in future works.

We now detail the behavior of Loupe *for each run*, as visualized in Figure A.1. Loupe first simultaneously sets up tracing and sandboxing (Ⓐ on Figure A.1) and starts the application (Ⓑ) using the `seccomp` [82] and `ptrace` [75] Linux tracing and interposition facilities. Once the application has been started, Loupe uses the test script to feed the application with inputs (e.g., generating client requests for a server application) and gather performance numbers (Ⓒ), all the while recording data on resource usage via `/proc` (Ⓓ). Using the hooks set up in Ⓑ, Loupe intercepts each system call invoked by the application, and tests it for one of the two previously described modes. At the end of the run, Loupe determines the success of the application using the return code of the test script (more in §A.3.2). Accesses to pseudo-files are hooked and disabled, stubbed, or faked similarly by catching system calls from the `open` family (see §A.3.3).

In order to maximize the reliability and reproducibility of the results, each analysis is performed multiple times in containerized replicas, and the result of the analysis

is conservatively updated to take all results into account. The number of replicas (*3* by default) and whether they run in parallel (*no* by default) can be configured to suit different applications, accuracy needs, and available hardware.

Finally, OS developers can specify the system calls supported by their OS in CSV form, and Loupe will recommend which OS features to implement, stub, or fake, to support a set of applications selected among those measured by the tool. Loupe will prioritize the list of features to indicate which should be implemented first in order to support as many applications, as early as possible. Loupe's measurements can optionally be shared in an online database (Ⓔ).

A.3.2 Evaluating Success and Performance

Loupe builds on the premise that users are able to describe a workload that they want to support for a given application. Loupe then tells the user which precise set of system calls they have to support (and how) to be able to run that workload reliably, i.e., over multiple runs without observable functional and non-functional issues.

Describing Workloads. Workloads describe the feature set that must be supported in a given application. Loupe users express workloads in *test scripts*, responsible for supplying external input, if required by the application, and detecting the success of a run.[1] Test scripts may materialize any type of workload: simple health checks (e.g., for a web server: can the application process a simple HTTP query?), benchmarks, test suites, or even fuzzing. If specific error cases or application features must be supported, then the test script must also exercise them as part of the run. In this paper we explore health checks, benchmarks, and test suites. Each workload may be relevant at different stages in the development life cycle of a new OS. Workloads correspond to different levels of guarantee of application stability; they can be evolved as support progresses, until complete compatibility can be provided to ensure stable application behavior in all circumstances.

Defining "Success". A run is considered successful when the application terminates and the test script exit code indicates success. Crashes, or unresponsiveness are considered as generic failure signs. The notion of generic failure can be extended to unusual

[1] Some programs do not require input and determine success by themselves or via a wrapper script (e.g., test suites). If so, the test script is *practically included* in the application and need not be passed separately. Loupe supports this. Since this is similar to the general case, we do not further discuss it here.

```
#!/bin/bash
# [...] omitted helpers (including is_failed and grep_req_per_sec)

b=$(wrk http://localhost:8080 -d10s | grep_req_per_sec)
if [[ $(is_failed $b $?) ]]; then exit 1;
else echo $b; fi
```

Listing 11: Example of a test script for Nginx benchmarked with `wrk`.

resource usage, or even unusual filesystem or network usage, which Loupe can observe without understanding application semantics. Generally however, the notion of success or failure is application-specific and inseparable from the workload itself: e.g., outputs on the standard output/error channels or logs that do not correspond to normal application behavior, or altered performance (e.g., throughput, latency, packet loss rate). Application-specific success criteria must be evaluated by the test script.

An example of a test script for Nginx benchmarked with `wrk` is shown in Listing 11. Here, `is_failed()` is responsible for detecting failures, left out of the listing for space reasons. When performing a simple health check, the function verifies that the throughput is non-zero.

We implemented detection of unusual resource usage and performance in our prototype. Loupe records application resource usage (maximum resident size and open file descriptors) via `/proc` and compares results over multiple runs when stubbing or faking. Similarly, when performing a performance benchmark, test scripts return the relevant performance number (which can be any application-specific performance metric), and Loupe ensures that the performance does not incur a statistically significant variation from the full-fledged baseline. Together, resource usage and performance checks can provide insights into the impact of stubbing or faking features, and particularly increase the confidence on the correctness (or incorrectness) of faking and stubbing. We further discuss performance and resource usage in §A.5.3.

A.3.3 Loupe in Detail

We now discuss various aspects of Loupe that are relevant in this paper: supporting vectored system calls and pseudo-files, making Loupe easy to use in many applications, how long Loupe analyzes take, and sharing analysis results.

Vectored System Calls. Identifying OS features at the granularity of an entire system call is sometimes too coarse, considering vectored system calls (e.g., `ioctl`, `fnctl`)

and system calls with several functionalities that may be partially implemented in a compatibility layer (e.g., `mmap`, or `madvise`). In such cases, Loupe can also disable, stub, and fake system calls based on *individual system call parameters*, allowing users to easily explore partial implementations at a fine granularity. The output is a list of system calls along with their used sub-features, and whether they can be faked or stubbed.

Beyond System Calls: Pseudo-Files. Part of the Linux API is exposed through pseudo-files such as `/dev/random`. Loupe is able to detect usage of such special files by pattern matching the arguments of certain system calls (e.g., `open`, `openat`) against paths (e.g., `/dev`, `/proc`). Loupe can also fake or stub system calls accessing these files, enabling users to track which special files require an implementation for applications to run.

Testing Framework Integration. Dynamic analysis tools can be difficult to integrate in application testing frameworks. Test suites, for instance, may start the application multiple times, from complex scripts, from different call points [40, 78]. Calling a naive analysis tool such as `strace` requires manual changes, along with additional logic to gather and merge results obtained from the multiple runs triggered by the test suite. Calling the tool on the test suite itself (e.g., `strace make test`) is not effective either, as the test suite may call external tools whose OS feature usage is not part of the application's. For instance, the Ruby test suite makes extensive calls to `git` to set up test environments; the OS feature requirements of `git` should not be included into the application's. We tackle this problem with a whitelist system: when run on a wrapper (e.g., a test suite), users can specify which binaries are that of the application and should be considered in the analysis. Loupe then tracks all children processes, checking the binary path upon `exec`, to ignore any system call originating from a binary that does not correspond to the specified one(s). This allows, for instance, unmodified analysis of test suites run via `make test`; Loupe simply executes the Makefile and only considers system calls executed by the appropriate binary.

Debhelper Integration. To further simplify running Loupe on many applications, we integrated Loupe into the Debhelper [25] Debian package build system. Loupe can build Debian packages and run on the package's `dh_auto_test` [28] rule which, if provided by the package, executes the target application's test suite. Combined with the previous technique, which Loupe can leverage by listing the package's binaries, we can significantly reduce the cost of testing applications. Running Loupe on the Lighttpd, Memcached, and webfsd test suites, for instance, is fully automated this way.

Loupe Run Time. The runtime of a full Loupe analysis is $(2 + (2 * t * s)) * \lceil \frac{r}{p} \rceil$ with: t the application workload runtime, s the number of distinct system calls (and pseudo-files, if enabled in the analysis) executed by the application under the specific workload, r the number of replicas, and p the number of replicas executed in parallel. 2+ corresponds to the initial run to discover executed system calls, and to the final run to confirm the analysis. 2∗ corresponds to the "stubbing" and to the "faking" run for each system call. The overall runtime is therefore dominated by the length and complexity of the application workload; it varies from about 4 minutes for a fast Nginx health check, to 50 minutes for the Lighttpd test suite, and 1-1.5 days for the SQLite test suite (by far the largest we encountered, running *millions* of tests [40]). These run times are reasonable: porting cost for a single application often reach multiple weeks or months in early OS development stages [285] and, as we expand next, this is a one-time cost.

Sharing Loupe Results. Thanks to the techniques described previously, Loupe test scripts are easy to write; 2-30 minutes on average according to the expertise of the user, most of it spent on understanding how to run and test the application. The main barrier to running Loupe on a large number of applications is runtime. Nevertheless, as we described previously, the results are final for a fixed build of the software, its workload, dependencies, kernel, and test script. To leverage this, we have set up a shared online database that can be populated and looked up by any individual running Loupe or interested in its results. Loupe can automatically submit results to the database along with metadata (Ⓔ in Figure A.1). We envision that in the long run, this database will contain results for a wide range of applications, helping OS and application developers to study OS features usage patterns, build compatibility layers, and more, without even the runtime cost mentioned previously.

A.4 Loupe: OS Feature Support Guide

For space reasons, we set aside pseudo-files and focus on system call support, as it represents the majority of the engineering effort to build compatibility layers [352, 285, 52].

A.4.1 Examples of Support Plans

We ran Loupe on a total of 116 applications with various workloads including standard benchmarks (e.g., `wrk` for web applications, `redis-benchmark`). We choose a selection of representative applications from OpenBenchmarking.org [64], as well as various other

Table A.1: Step-by-step support plans for 3 OSes.

Step	Implement	Stub	Fake	Support for…
Unikraft (commit `7d6707f`, supports 174 syscalls)				
0	-	-	-	(12 apps)
1	290	273, 218, 230	-	+ Memcached
2	218	-	-	+ H2O
3	283, 27	186	-	+ MongoDB
Fuchsia (commit `5d20758`, supports 152 syscalls)				
0	-	-	-	(10 apps)
1	33	273, 302, 105	-	+ Lighttpd
2	302	230	-	+ Memcached
3	-	99, 222, 223	-	+ HAProxy
4	105	40	128, 99, 27	+ Nginx
5	128, 99, 27	-	-	+ MongoDB
Kerla (commit `73a1873`, supports 58 syscalls)				
0	-	-	-	(4 apps)
1	56, 257, 54	(17 syscalls)	47	+ Httpd
2	10	-	302	+ Weborf
3	8, 21, 87	-	25	+ SQLite
4	232, 233, 302	(9 syscalls)	288, 213	+ HAProxy
5	17, 213, 262	95	-	+ Redis
6	291	105, 106, 116, 293	-	+ Lighttpd
7	288, 290	32	102	+ H2O
8	46	230	-	+ Memcached
9	105, 18, 53, 106	40	92, 130, 107, 273, 116, 157	+ Nginx
10	104, 107, 108, 102	-	-	+ Webfsd
11	128, 99, 229, 27, 73, 202, 283	131	137	+ MongoDB

sources [71, 79, 94][2]. Leveraging these measurements, Loupe guides the process of developing a compatibility layer by giving a prioritized list of system calls to implement/stub/fake. Specifically, given (1) the state of a partially Linux-compatible OS in terms of system calls supported (a simple text file with one line per supported system call) and (2) a set of target applications to support, Loupe can output an incremental support plan listing the order in which missing system calls should be implemented/faked/stubbed in order to enable compatibility with a maximum of applications as early as possible.

We enabled Loupe to generate support plans for all 116 applications we measured, for 11 OSes under development: Unikraft [285], Google Fuchsia [36] and Zephyr [102], Kerla [52], HermiTux [352], Google gVisor [39], Graphene/Gramine [430, 38], FreeBSD Linuxulator [54], Browsix [369], OSv [276], and Linux nolibc [423]. To illustrate this functionality, we present here a subset of these results (for space reasons): we consider recent versions of 3 OSes: Unikraft, Fuchsia and Kerla, and a target set of 15 popular

[2]Our artifact includes a list of all applications and support plans for 11 OSes: https://github.com/unikraft/loupedb/blob/staging/ASPLOS24-supp.pdf

cloud applications. The support plans are presented in Table A.1. The number of steps to reach support for all 15 apps is directly linked to the maturity of the OS: Unikraft for example has initial support for 12 applications and requires only 3 steps to reach full support, while Kerla, with initial support for only 4 applications, requires 11 steps. Loupe's incremental support plans optimize the development of compatibility layers by breaking down the effort into small steps (>80% of which requiring to implement 1-3 system calls), unlocking support for an application after each step. The support plans in Table A.1 target a small set of applications for space reasons. Full support plans for each of the 11 OSes we target, for all 116 applications in our database, are larger: 35 steps for Fuchsia, 32 for Unikraft, and 79 for Kerla.

A.4.2 Engineering Effort Savings

To estimate the engineering effort savings that an OS project would enjoy while building a compatibility layer with Loupe rather than in an ad-hoc, organic fashion, we designed the following experiment: we select a large set of 62 applications supported by a popular experimental OS, OSv [276], from the OSv-Apps repository [71]. We then estimate the order in which these applications were organically supported by the OS. For that we use git metadata to track the creation date of the folder corresponding to each app in the repository. We then derive from the order in which applications were supported, the organic order in which system calls had to be implemented by OSv developers. Because stubbing/faking OS features are well-known practices [242], and because there are traces of their usage in OSv's codebase [104], we assume that OSv developers used stubbing and faking as much as possible. We can then derive, in chronological order, the number of system calls that were implemented by OSv developers, and the evolution of the number of supported applications. We also compute these numbers for a hypothetical optimized compatibility layer development process that would be guided by Loupe's support plan, which would also take stubbing/faking into account, as well as a naive approach that would implement every system call traced by dynamic analysis, without stubbing/faking.

These results are presented on Figure A.2. As one can observe, Loupe would have heavily optimized the process of implementing OSv's support for the target application set, leading to more applications supported earlier and with less engineering effort vs. our estimation of the organic process undertaken by OSv's developers. For example, to support half (31) of the applications, with Loupe only 37 system calls need to be implemented, vs. 92 for the organic process. The naive method relying on dynamic

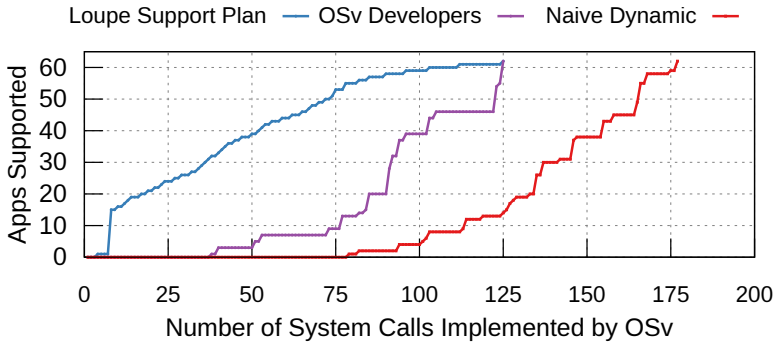


Figure A.2: Evolution of the number of applications and system calls supported by OSv assuming 1) a support plan generated with Loupe, 2) organic development based on git history, 3) measurement with naive dynamic analysis without stubbing/faking. Higher values indicate more applications supported for the same effort.

analysis without stubbing/faking requires even more engineering effort: to reach 31 applications, 142 system calls would need to be implemented.

Our method to estimate engineering efforts makes a few simplifications. The real order in which applications were supported by OSv is likely not exactly that of folder creation in the OSv-Apps repository. We repeated the study using the date of the *last commit* in each application's folder to determine the order; results were similar. The effort to implement system calls is also variable according to which system call is targeted: the x-axis in Figure A.2 is non-uniform since not all system calls have the same implementation cost. However, we believe these results provide a sufficiently solid estimation of the engineering effort reduction that Loupe can bring to demonstrate its usefulness.

A.5 Analyzing the Linux API with Loupe

Here we study the Linux API usage results obtained using Loupe for the 116 applications considered in our study. We aim to answer the following research questions:

- How important is the accuracy gap between Loupe's method vs naive dynamic analysis (`strace`) and static analysis?
- When building a Linux compatibility layer, which system calls must be implemented, and which ones can be commonly faked or stubbed? What is the absolute minimum set of system calls that must be implemented for a test suite to correctly run?

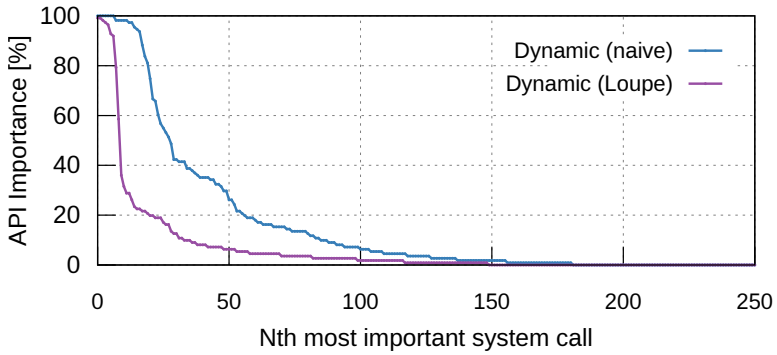


Figure A.3: API importance for dynamic analysis with Loupe and a naive approach (i.e., without stubbing and faking). A lower curve means less system calls are required to support applications.

- What are the most important system calls, i.e., the ones whose implementation is required by most applications?
- Why can some system calls be faked or stubbed? Does it impact performance or resource usage metrics?
- How much do the system call requirements of applications and standard libraries evolve over time?

For space reasons, we concentrate on system calls and set aside results regarding special files and vectored system calls.

A.5.1 Analysis Method: Static vs. Dynamic

Loupe vs. Naive Dynamic Analysis. We computed the *API importance* of each system call as reported by Loupe and by naive dynamic analysis. API importance [431] represents the probability that in our 116 applications data set, a system call is required by at least one application in that set. A system call is defined as required for an application if it is traced with dynamic analysis, and if it is traced and can neither be stubbed nor faked with Loupe.

Figure A.3 visualizes our results. They show that naive dynamic analysis severely overestimates the amount of system calls required to support applications. Loupe reports a total of 148 system calls requiring implementation to support 100% of our 116 applications, vs. 180 system calls for a naive analysis. The 25 most commonly required system calls are present in more than 80% of the applications with Loupe, and in less than 50% with naive dynamic analysis.

Loupe vs. Static Analysis. We faced scalability issues when trying to apply binary- and source-level static analysis tools to our data set of 116 applications. There exists no source-level tool able to identify system calls for all the relevant programming languages. We also attempted to run several binary-level tools and experienced a high level of failures (close to 50%) skewing the results. Hence, we fall back on selecting a subset of applications from our data set for comparison between static analysis and Loupe.

We select 7 popular cloud applications that support standard benchmarks and ship with comprehensive test suites: Redis, Nginx, Memcached, SQLite, HAProxy, Lighttpd, and Weborf. To gather results for static analysis we use the source- and binary-level tools made available by Unikraft [97, 96]. Figure A.4 details the amount of system calls identified in each application by each method. Both static analysis techniques severely overestimate the number of system calls actually needed to run the benchmarks and test suites. The minimum number of system calls identified by Loupe as required for these applications varies around 20 for benchmarks, and 20-40 for test suites. Both static binary and source analysis methods report numbers that are generally between 5x and 2x higher. For example, on Redis, binary-level static analysis identifies 103 system calls vs. 68 dynamically traced ones from the test suite, and Loupe further indicates that more than a third of these can be stubbed/faked. This observation can be generalized to all other applications. Overall these results show that the effort to provide comprehensive support of core features and even full test suites is much lower than suggested by previous work based on static analysis [431].

Figure A.5 details which system calls are detected by the various analysis techniques when applied to the 7 applications running benchmarks. Once again the overestimation of static and naive dynamic analysis is clear, compared to the results obtained with Loupe. Regarding static analysis, operating on the binary only yields more system calls compared to targeting the sources. Concerning dynamic analysis, a non-negligible amount of system calls can be stubbed/faked, confirming the benefits of Loupe vs. naive dynamic analysis. We investigate faking/stubbing more in details next.

> **Insight:** Static and naive dynamic analysis both highly overestimate the engineering effort needed to build a compatibility layer for a target set of applications.

A.5.2 Resilience to Stubbing and Faking

As visualized in Figure A.4, we find that, on average, the proportion of invoked system calls that can be stubbed or faked is 46% for test suites (ranging from 31% for Nginx to

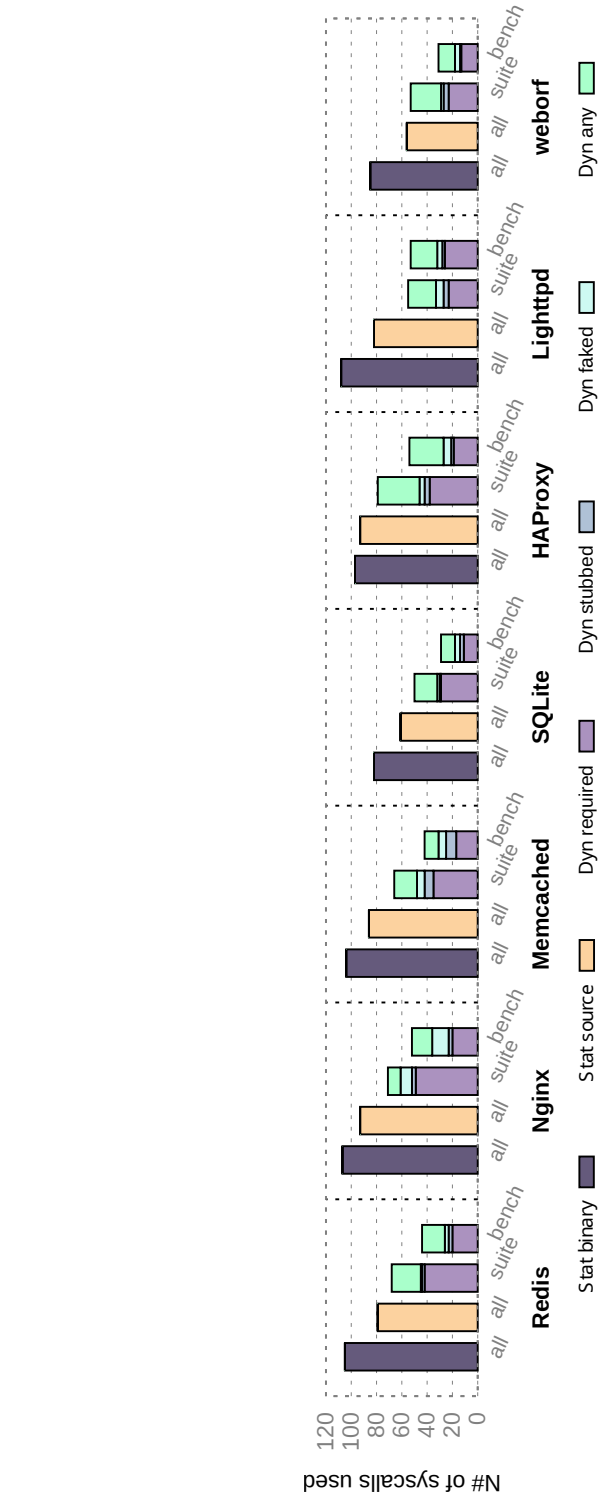
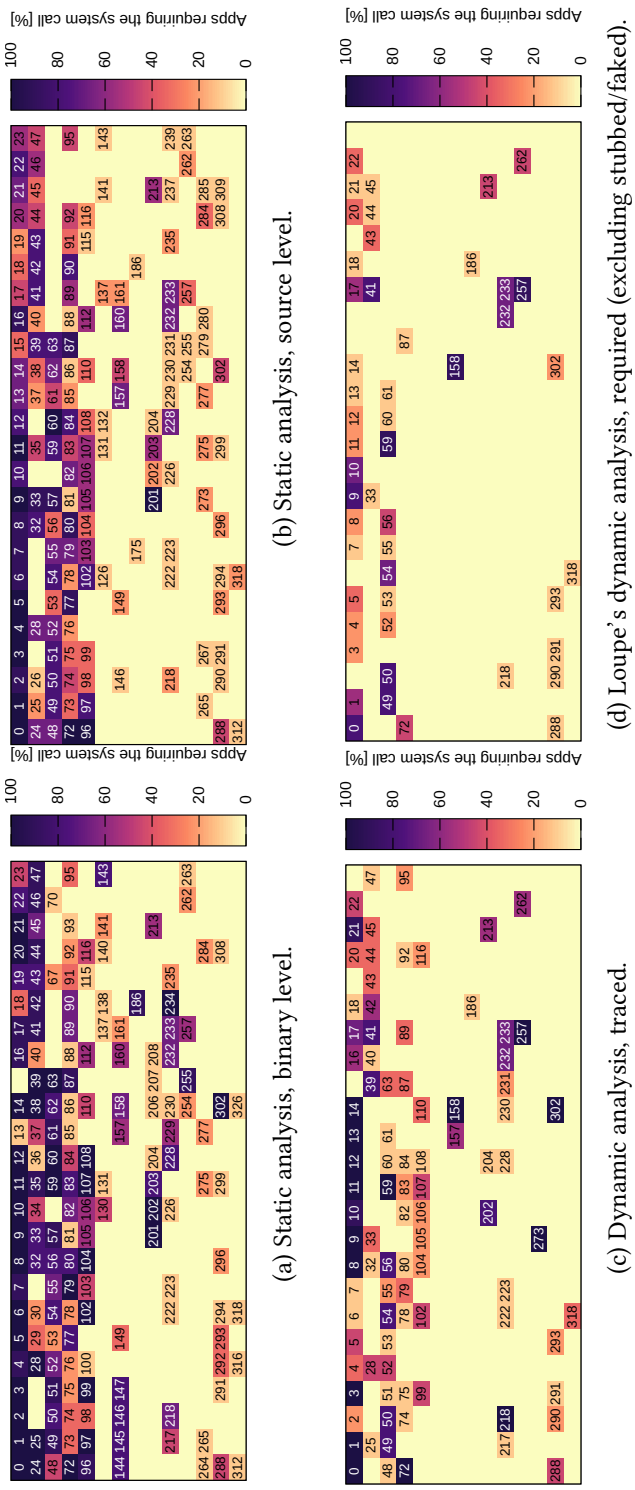


Figure A.4: Number of system calls statically identified and dynamically traced by Loupe for applications running standard benchmarks (*bench*) and test-suites (*suite*). Traced system calls are broken down into those that can be stubbed, faked, either faked or stubbed (*any*), and those that can neither be faked nor stubbed (*required*).



(a) Static analysis, binary level.

(b) Static analysis, source level.

(c) Dynamic analysis, traced.

(d) Loupe's dynamic analysis, required (excluding stubbed/faked).

Figure A.5: System calls identified by static binary, static source, naive dynamic *traced* (all system calls detected), and Loupe's dynamic *required* (those that cannot be stubbed/faked). Each box represents a Linux system call and its number.

58% for Lighttpd), and 60% for benchmarks (from 51% for Lighttpd to 65% for HAProxy). This shows that the effort required to provide strong support of core features (i.e., those covered by test suites) for these popular applications is certainly lower than suggested by previous work, and is even lower when considering support for benchmarks only (needed for evaluation in research papers). The difference between Figure A.5c and A.5d clarifies this, highlighting which system calls can commonly be stubbed and faked. We observe broadly two categories:

- **Low range system calls (system call ID ~< *150*)**, representing the majority of system calls detected by all analysis methods. This is unsurprising as these system calls correspond to core services that have been present in the Linux feature set for a long time, such as basic network system calls (`bind`, `accept`, etc.).
- **Higher range system calls (ID ~> *150*)**, where a small set of popular system calls are invoked corresponding to more modern but prominent functionality concerning multithreading (`futex` – 202, etc.), scalable I/O (`epoll` family – 213, 232, 233), as well as new variants of core system calls (`openat` – 257, `prlimit64` – 302, etc.)

Though system calls from both categories can be stubbed or faked, system calls with higher numbers are better candidates: out of the lower half of used system calls (46 system calls with number < *63*), 13 system calls can always be stubbed vs. 30 for the upper half (46 system calls with number > *63*). This is because these map to more recent, generally less critical functionalities; we expand on this next.

> **Insight:** Though applications may invoke many system calls, many of them can be stubbed or faked to run popular workloads.

Why are Programs Resilient to Stubbing and Faking? Applications are able to detect and react to the failure of a system call. Often, system call failures are non-critical and programs can take action to circumvent them. These actions are the enabling factor of system call stubbing. They include, among others (cf. Figure A.5d):

- **Ignoring the issue.** Not all failures are consequential, and programs can simply decide to not take further action. For instance, Redis ignores when `sysinfo` (99) fails to return the maximum memory size and when `ioctl` (16) fails to return the resident size, as this information is only used for output to the debugging logs.

```
if (getrlimit(RLIMIT_NOFILE, &limit) == -1) {
  serverLog(LL_WARNING, "Unable to obtain the current NOFILE limit,"
  "assuming 1024 & setting the max clients config accordingly.");
  server.maxclients = 1024 - CONFIG_MIN_RESERVED_FDS;
}
```

(a) Stubbing-resilient Situation (Redis).

```
if (prctl(PR_SET_KEEPCAPS, 1, 0, 0, 0) == -1) {
  ngx_log_error(NGX_LOG_EMERG, cycle->log,
  ngx_errno, "prctl(PR_SET_KEEPCAPS, 1) failed");
  exit(2); /* fatal */
}
```

(b) Faking-resilient Situation (Nginx).

Figure A.6: Real-world code snippets where it is effective to stub (top) and fake (bottom) system call implementations.

- **Using other system calls**. The system call API is redundant in features: a same means can often be achieved through different system calls. For instance using `mmap` (9) instead of `brk` (12) – a pattern from the glibc early allocator, or reallocating mappings with `mmap` (9) when `mremap` (25) fails, as we observe in SQLite.
- **Falling back to safe default values**. Applications query the OS for various values to tune their behavior (max stack size and file descriptor count, processor affinity and scheduling importance, etc.). When this fails, a safe default can often be adopted. Figure A.6a shows an example with `getrlimit` (97) and `prlimit64` (302) in Redis. Another example is using `ioctl` (16) to query the terminal width: when this fails, Redis assumes a safe value of 80 characters.
- **Disabling program functionalities**. Programs may also decide to simply disable the functionality that makes use of the system call; in certain cases, this may not even have observable consequences. For example, many applications only make use of `connect` (42) through the glibc for the NSCD cache socket [63]. When `connect` fails, name caching is simply disabled.

In other cases, programs may interpret the failure conservatively and decide to abort, making stubbing impossible. Still, in a subset of these cases, programs are overly conservative and the failure of the system call is in reality non-critical: if so, faking

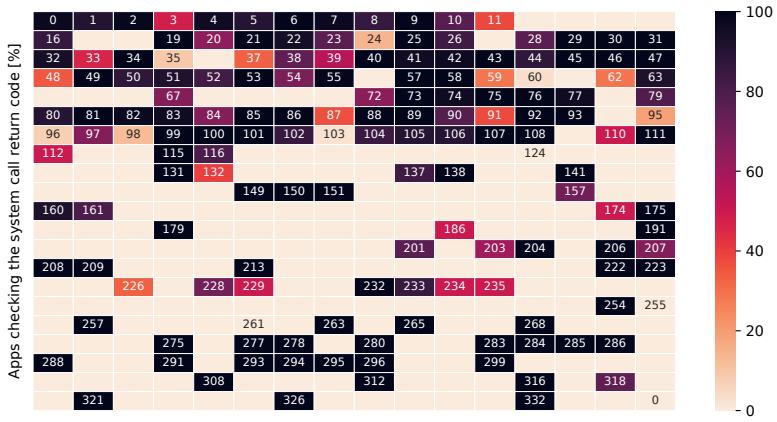


Figure A.7: Applications checking system calls return values. Each box represents a system call. The darker the box, the higher the percentage of applications checking the system call return value.

a successful return value for the system call, without *actually* doing the work of the system call in question, will work. Figure A.6b presents a concrete example in Nginx, where `prctl` (157) fails to force the retaining of capabilities upon UID transition; in the context of an OS that does not have a user/kernel separation, such as a unikernel, capabilities make little sense and so it is fine to fake success: faking the system call here will have strictly no impact on the correct execution of the software. Similar examples are `get/setgroups` (115-116), or `setsid` (112) which have, once again, no meaning in the context of a unikernel. Still, faking OS features may also result in breaking program functionalities, e.g., `pipe2` (293) in Redis (see §A.5.3). If the functionality is not part of the target set of application features, faking may remain a reasonable approach to achieve a first level of compatibility.

Inversely, certain system calls can (almost) never be stubbed nor faked without breaking core program functionalities. Though generalization is difficult, these system calls typically represent fundamental OS features: executing programs with `execve` (59), opening and writing to connections with `bind` (49), `listen` (50), `socket` (41), and `writev` (20), allocating memory with `mmap` (9). We also find vectored system calls such as `fcntl` (72), motivating our discussion in §A.5.4.

System Call Return Value Checks. In addition to identifying system calls issued by applications, we performed a manual inspection of these applications' source code in order to gather ground truth about which system calls had their return values checked. Is there a link between the presence or absence of checks, and the ability to stub or

fake? Note that we are interested here in user-written code, so we look at whether C standard library system calls wrappers – not system calls themselves – have their return value checked.

We choose manual inspection; building an automated static analysis method for this task is non-trivial and rather out of the scope of this paper: some programs directly check the return value, others store it in a variable which is later checked, directly or through auxiliary functions, while yet others rely on macros to do the checking. We semi-automated the process by building scripts scanning sources for system call wrapper invocations and displaying their corresponding location in source files; we then manually checked this output to determine if the return code was checked or not.

Figure A.7 shows, for each system call wrapper, the number of programs that check its return value. The majority have their return value checked. Studying the small set of system calls for which no application has checks, we identify system calls that always succeed, e.g., `alarm` (37), `getppid` (110), but also several that can actually fail: `getrusage` (98), `utime` (132), `inotify_rm_watch` (255) and `futimesat` (261). For those invoked and traced by Loupe, we observe that all can be stubbed/faked for this set of applications. Nevertheless, it would be incorrect to conclude that the ability to stub and fake is induced by the absence of checks: inversely, numerous system calls that are always checked can themselves often be stubbed/faked, such as `ioctl` (16), `uname` (63), or `geteuid` (107). There is also a set of system calls for which only some applications feature checks. These include system calls that are generally assumed to always succeed (even if they can fail) such as `clock_gettime` (228), or freeing resources: e.g., `close` (3), or `unlink` (87). Generally, these can be stubbed/faked only in some applications. Overall, we conclude that the ability to stub or fake is not a factor of the presence (or absence) of checks, but rather of the semantics of individual system calls and applications.

A.5.3 Impact on Performance and Resource Usage

An important concern when stubbing and faking system calls is whether doing so would have an effect on performance or resource usage. Both detrimental and *positive* effects are undesirable, as unintended improvements on these metrics may skew comparisons with a full-fledged baseline. To study the question, we use Loupe's ability to record performance and resource usage metrics while performing its analysis.

Table A.2: Performance and resource usage (file descriptors: FD, memory usage) impact of stubbing and faking for Nginx, Redis, and iPerf3 (*Applications*). Only systems calls with impact outside of the error margin (>3%) in either category are displayed. "-" means *no impact*; *+X%* means *X% faster* or *more* resource usage; *-X%* means *X% slower* or *less* resource usage.

Application	System Call	Perf. Impact	FD Usage	Mem. Usage	Explanations of Stubbing/Faking Impact	Breaks...
Nginx	`write`	+15%	-	-	Access logs are not written anymore, increasing performance.	Access Logging
	`brk`	-	-	+17%	Triggers a fallback to `mmap` in the glibc early allocator.	∅
	`clone`	-	-	+10%	Results in master process executing the worker loop.	Core functioning
	`sigsuspend`	-38%	-	-	Results in master process polling (busy-waiting) for events.	∅
Redis	`close`	-	x8	-	FDs are not closed anymore.	∅[1]
	`munmap`	-	-	+19%	Regions are not disposed anymore.	∅[2]
	`brk`	-	-	+2%	Triggers a fallback to `mmap` in the glibc early allocator.	∅
	`sigprocmask`	-	-	-15%	Prevents creation of jemalloc background threads, resulting in memory being freed synchronously and/or at an earlier point.	∅
	`futex`	-66%	+94%	-	Inconsistent synchronization results in incorrect behavior.	Core functioning
	`pipe2`	-	-25%	-	Pipes are not created anymore, resulting in less FDs.	Persistence
iPerf3	`brk`	-	-	+11%	Triggers a fallback to `mmap` in the glibc early allocator.	∅

[1]Within the maximum number of FD limits, core functioning is altered beyond this point. [2]Within the limits of available memory.

As described in Appendix A.3.2, Loupe gathers performance metrics through user-defined scripts, and resource usage information (peak file descriptor and memory usage) through `/proc`. For the sake of conciseness, we provide detailed results for a subset of three representative, performance-focused applications: Nginx (web server), Redis (key-value store), and iPerf3 (TCP benchmark framework). Nginx is benchmarked with `wrk` [100] (HTTP requests/s), Redis with `redis-benchmark` [77] (SET requests/s), and iPerf3 with an official iPerf client [50] (TCP throughput). All numbers are provided as averages of 10 runs. Our results are visible in Table A.2.

Impact on Performance. For the majority of system calls, the variation in performance when stubbing or faking is within the error margin. For the applications considered here, 3/45 system calls trigger a performance change when faked or stubbed. For Nginx, stubbing/faking `write` increases performance as it prevents writing to access logs [61] (something that test scripts do not check – access logs are usually disabled in performance-focused settings as they are written to once per request). It does not, however, prevent payloads from being written to, as this is done via `writev` (which, when stubbed or faked, prevents Nginx to answer requests correctly, and is detected by the test script). Still for Nginx, stubbing or faking `rt_sigsuspend` hurts performance, as it turns the master process' notification-based behavior into busy-waiting. None of these alters the well-functioning of Nginx's core features as tested by the Loupe test script. Conversely, in the Redis case, faking `futex` results in synchronization issues, manifesting as a performance degradation. This alters the core functioning of Redis, clearly indicating that faking `futex` is not a correct path to follow for compatibility, which matches intuitive expectations. As for iPerf3, no system call results in performance degradation when faked or stubbed.

When such variations occur, Loupe notifies the user that further investigation is needed to understand the implications (e.g., on stability or scientific soundness) of stubbing or faking a particular OS feature for a given application. This further emphasizes the need for a tool such as Loupe to avoid pitfalls which may cause debugging costs down the line, or skew comparisons with a full-fledged baseline.

Impact on Resource Usage. Similarly to performance, we find that faking or stubbing most system calls does not result in statistically significant variations in resource usage. For the three applications considered, 4/45 system calls result in memory usage variations, and 3/45 in file descriptor usage variations, with one (`brk`) being caused by the libc and thus common among all three applications.

In the general case, as discussed earlier, system calls that allocate resources cannot be stubbed or faked: this is the case for memory allocation services such as `mmap` (9), but also for those that allocate file descriptors such as `openat` (257) (see Figure A.5d). In particular cases, the claim is more nuanced; alternatives such as `open` (2) do not need to be implemented (e.g., because `openat` is used instead, see Section A.5.6). Similarly, `brk` *can* be stubbed or faked in a significant number of cases: for instance, the program exclusively uses `mmap`, and the only usage of `brk` is in the glibc initialization sequence, which is itself capable of falling back to `mmap` if `brk` does not function (at the cost of a slight memory usage increase, see Table A.2). Another case is `pipe2`, which creates pipes at the process' demand. Stubbing or faking it results in pipes not being created, which in turn results in an observable reduction in file descriptor count. In the case of Redis, this breaks the persistence feature (which is often disabled in performance-focused experiments), but not the key-value store's core functionalities.

The situation is different for APIs that free resources. In general, `munmap` and `close` can be stubbed or faked without functional impact, though resource usage will increase. For Redis, faking or stubbing `munmap` and `close` leads to a 20% increase in memory usage, and an 8x increase in open file descriptors under a `redis-benchmark` workload (cf. Table A.2). Still, although these features can be stubbed or faked without sacrificing stability (as long as resources suffice), we note that the incentives to do so are lower than for other API elements; if the algorithm was developed to allocate resources, it should not be a problem to develop one that frees them.

Lastly, similarly to performance, variations in resource usage turn out to be good indicators of instability caused by stubbing or faking. In the case of Nginx, faking `clone` results in the master process executing the worker event loop, which itself manifests as an increase in memory usage (likely because resources are left dangling). Although functional in practice, it is not a reliable path to take for compatibility and meaningful performance comparison. In the case of Redis, faking `futex` results in inconsistent synchronization, which itself translates into an increased number of allocated file descriptors (see Table A.2).

Beyond system calls that (de-)allocate resources, and those that indicate underlying instability, we identify two more classes of system calls which may impact resource usage (or performance):

- **Optimizing system calls:** by giving semantic indications to the kernel regarding e.g., memory management policies, system calls such as `madvise` [55] should influence performance and resource usage. This behavior is not visible when

faking or stubbing in Table A.2: kernel hints are used rather sporadically in applications, and for those that use them (e.g., Redis), the kernel did not perform actions that impacted our metrics. Impact may be observable in other settings, for instance in multi-process scenarios.

- **System Limit Setters/Getters:** by getting/setting system defaults (e.g., max stack size, number of FDs), getter/setter system calls such as `prlimit64` (or part of `ioctl`) may also result in resource usage or performance variations. For instance, with system defaults different from the ones in Table A.2, stubbing `prlimit64` in Redis results in 30% lower memory usage under a `redis-benchmark` workload because the libc (stack size) and Redis (FD limits) default to values conservatively lower than the system limits.

Impact on Stubbing and Faking Policy. Overall, we stress the importance of evaluating the impact of stubbing and faking on performance metrics as part of the process of deciding what to support and how. Though most system calls do not impact performance metrics, some do: when the underlying reason is instability, the OS feature should never be faked; otherwise, whether or not to stub or fake should be an *explicit factor* of the experimental setup and expectations on the OS prototype. It is critical that the (positive or negative) impact of stubbing and faking must not be mistaken for that of the system's design. Overall, we encourage authors of future systems research works to explicitly list features that they stub or fake for reproducibility and future analysis.

> **Insight:** Stubbing or Faking does not impact performance and resource usage in the general case. Still, there are edge cases which may or may not indicate correctness issues. Impact on either metric must call for special care when stubbing or faking.

A.5.4 Partial Implementation of System Calls

In the previous sections, we considered system calls as monolithic API elements. This consideration shows its limits when investigating vectored system calls (e.g., `ioctl`) or complex system calls such as `mmap` (usable for memory allocation and file mapping, two very different purposes). To clarify this point, we use Loupe to determine the precise set of sub-system call features applications require.

Our insights are twofold. First, applications execute surprisingly few features from complex or vectored system calls. For example, almost all applications require `arch_prctl`

(158) (see Figure A.5d). However, they are far from requiring a full implementation: in fact, in all applications that we considered, this system call was exclusively called by the libc, which requires one single feature (`ARCH_SET_FS`, out of 6 in total) related to thread local storage setup. The situation is similar for `prlimit64` (302), required by many applications: out of 16 features, only 3 are used, `RLIMIT_CORE`, `_NOFILE`, and `_STACK`, the latter one being used almost exclusively as part of the libc initialization. This is also the case for `ioctl` (16): with a benchmark load, Redis, weborf, and h2o use one single feature (`TCGETS`), Nginx two (`FIONBIO` and `FIOASYNC`), and Lighttpd none. All of them can be stubbed.

Second, when looking at required features of system calls, we find that certain system calls such as `fcntl` typically exhibit a mix of required and fakeable/stubable features, and the required set is typically common among applications. For instance, `F_SETFL` is required to put file descriptors in non-blocking mode in all applications except Nginx, a critical operation for most codebases. On the other hand, `F_SETFD` is widely executed across applications but can always be stubbed as it is used to enable *close-on-exec* on file descriptors, a non-critical operation. In these cases, taking a look at the required system calls at the granularity of a system call would make the situation appear worse than it is in practice.

> **Insight:** Several complex system calls do not require a full implementation to support a large number of applications.

A.5.5 Stability of System Call Usage Over Time

Once an OS prototype supports an application, how likely is it that, as the program evolves over time, additional or different system calls will be required, breaking the initial support? Here we study the stability of system call usage by applications and libcs.

Evolution: C Standard Library. We first study the libc, from which most system calls invocations generally originate. We compiled Nginx v0.3.19 against an old version of glibc (2.3.2, from 2003) and a modern one (2.31, from 2020). Since we were unable to run Nginx 0.3.19 with glibc 2.3.2 in 64-bit mode, we compiled and run this configuration in 32-bit. This is likely due to these versions featuring unstable AMD64 support (the first AMD64 CPUs were released in 2003 [266]). The results in Table A.3 show that the number of used system calls is more or less unchanged, 48 vs 51. Moreover, we see that most of the change in system call usage is caused by the deprecation of old system calls. Still, there is some evolution in the types of system calls invoked, which we classify into

Table A.3: Nginx 0.3.19 system call usage with different glibc versions. System calls that vary because of the architecture (32/64-bit) are in *italics*; other variations are in **bold**.

GLIBC 2.3.2 / 32-bit (*48 system calls*)	GLIBC 2.31 / 64-bit (*51 system calls*)
_llseek, accept, access, bind, brk, clone, close, connect, epoll_create, *fcntl64*, epoll_ctl, epoll_wait, execve, exit_group, dup2, *fstat64*, *geteuid32*, mkdir, **mmap2**, *setuid32*, **old_mmap**, *setgroups32*, **uname**, **open**, prctl, *pread*, **pwrite**, read, rt_sigaction, rt_sigprocmask, rt_sigsuspend, **set_thread_area**, *setgid32*, setsid, setsockopt, **recv**, socket, socketpair, *stat64*, bind, munmap, umask, getpid, getrlimit, ioctl, write, writev, gettimeofday, listen	read, write, close, *stat*, *fstat*, lstat, *lseek*, brk, rt_sigaction, **mmap**, ioctl, rt_sigprocmask *pread64*, setsockopt, writev, access, sendfile, socket, munmap, accept, connect, epoll_wait, **mprotect**, **recvfrom**, listen, socketpair, *pwrite64*, *prlimit64*, epoll_create, clone,execve, *fcntl*, mkdir, umask, *setuid*, *setgid*, *geteuid*, setsid, rt_sigsuspend, dup2, *setgroups*, **_sysctl**, prctl, **arch_prctl**, getpid, **set_tid_address**, exit_group, epoll_ctl, **openat**, **set_robust_list**

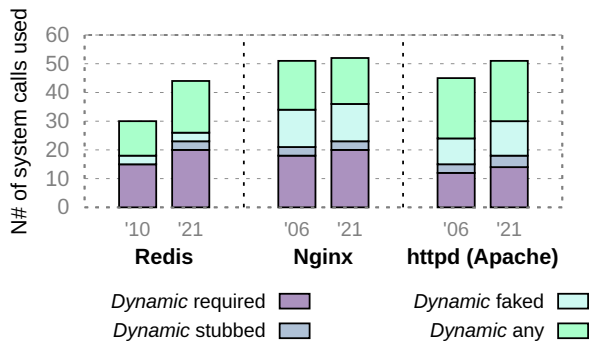


Figure A.8: System call usage and capacity to be stubbed/faked for recent (2021) and older (2005-2010) releases of Redis, Nginx, and Apache HTTPd.

two categories. First, the recent libc uses a different version of some system calls due to a change of architecture (e.g., it uses `pread64` instead of `pread`). Second, the recent libc uses additional system calls, e.g., `arch_prctl` to set up thread-local storage. Setting aside the issue of supporting a new architecture (orthogonal to compatibility), we assume that it is the second category that would require supporting effort for updating a given compatibility layer as applications evolve. However, we consider this effort to be low: we only count 8 new system calls in 17 years for this libc/application combination.

Evolution: Application. We are now interested to see how the system call usage of an application changes over the years. For this experiment, we used a modern glibc/-compiler. We explore the difference in system call usage of Nginx, Apache and Redis through the years and list the results in Figure A.8. We observe that, although the number of Linux system calls has increased, all applications are using roughly the same amount of system calls; the number of system calls that can be stubbed or faked also

remains almost unchanged.

In all, we find the usage of system calls by applications and libcs to be fairly stable over time. This is further encouragement to OS prototype developers: once you provide support for an application, you are likely to be able to keep it with minimal work for a long while.

> **Insight:** Application and libc system call usage patterns tend to be stable over time: support is a one-time effort.

A.5.6 C Library Impact on System Call Usage

Typical applications perform the majority of their system call invocations through the C standard library (libc). Bypassing the libc using direct system call invocation happens only for functionalities rarely called by user code (e.g., `futex`) or newer system calls for which libcs do not provide a wrapper: we counted around 51 system calls (58 including removed/unimplemented system calls) that do not have a wrapper as of glibc 2.33. In this case, applications wishing to invoke them use the `syscall` function. Setting aside these special cases, we find that the libc implementation greatly influences the system call API usage of applications. This is due to two main factors: (1) the libc initialization sequence and (2) the choice of system call alternatives (e.g., `openat` vs. `open`).

Libc Initialization Sequence. The initialization sequence is the libc code executed from the program entry point until the user's `main` function is invoked. The system calls invoked by that code will be by construction present in any binary linked against that libc and constitute the minimum set of system calls an OS should implement to support this libc. To study initialization sequences, we recorded the system call usage of a trivial application printing "Hello, world!" across two libcs, glibc (version 2.28) and musl (version 1.2.2), for a dynamically- and a statically-linked executable. Results in Table A.4 show that the number and types of system calls executed vary: glibc's initialization sequence invokes for dynamically compiled binaries 2.5x more system calls vs. musl, and 1.8x more for statically compiled programs. The system calls invoked also change: glibc is not a strict superset of musl and out of 18 system calls in total, only 6 are common to both libcs for dynamic, 3 for static (and 3 overall).

System Call Alternatives. Some discrepancies are due to the libcs choosing different system call alternatives to perform the same task. For example, glibc uses `write` for `printf`, vs. `writev` for musl. Similarly, musl uses `ioctl` to check that the TTY is writable,

Table A.4: System call API usage of a hello world application across glibc (2.28) and musl (1.2.2). Apart from `exit_group` and `write`/`writev`, this set corresponds to the libc initialization sequence. Differing system calls are in bold.

glibc	musl
28 system calls (dynamic binary)	*11 system calls (dynamic binary)*
execve (1x), brk (3x), arch_prctl (1x), exit_group (1x), **access** (1x), **openat** (2x), **fstat** (3x), mmap (7x), **close** (2x), **read** (1x), mprotect (4x), **munmap** (1x), **write** (1x)	execve (1x), brk (2x), arch_prctl (1x), exit_group (1x), **writev** (1x), mmap (1x), mprotect (2x), **ioctl** (1x), **set_tid_address** (1x)
11 system calls (static binary)	*6 system calls (static binary)*
execve (1x), arch_prctl (1x), exit_group (1x), **brk** (4x), **fstat** (1x), **write** (1x), **uname** (1x), **readlink** (1x)	execve (1x), arch_prctl (1x), exit_group (1x), **writev** (1x), **ioctl** (1x), **set_tid_address** (1x)

while glibc uses `fstat`. Finally, glibc uses `openat`, `read`, `mmap`, and `mprotect` to map the libc into the address space, an operation that musl achieves by embedding the libc into the linker itself, avoiding these system calls entirely. Other differences are caused by libc-specific initialization and debugging features. For example, even in single-threaded programs, musl will call `set_tid_address` during the initialization of thread-local storage, something that glibc does not. Glibc, on the other hand, uses `uname` to ensure that the kernel is recent enough, `readlink` to expand `$ORIGIN` with statically compiled binaries, and `access` for a debugging feature; none of these used by musl's initialization sequence.

> **Insight:** The *choice of libc* and *linking type* strongly influences system call usage: as much as 4.5x fewer system calls between dynamic glibc and static musl.

A.6 Discussion: Pitfalls & Future Works

As discussed throughout this work, there are pitfalls to developing OS compatibility layers with dynamic analysis, stubbing, faking, and partial support techniques.

Impact on Stability. Dynamic analysis, stubbing, faking, and partial support techniques, bring the concern of stability: *do we trade off correctness to reduce porting time?* Loupe assumes that users are able to evaluate the functionality of application features they aim to support by specifying a set of tests (§A.3.2). The tool ensures that this set of tests can be passed reliably, over multiple runs, when applying stubbing, faking, and partial support techniques. Loupe can also ensure that performance, resource usage, and any other metric, remains stable (§A.5.3). Under this assumption, stability issues

outside users' target feature range are not in the problem scope of Loupe, or our study. Still, perfect correctness cannot be guaranteed, and compatibility bugs may hide in incomplete or buggy tests, varying test environments, etc. We believe that these are reasonable trade-offs to be made in transitional development stages of a new OS.

Impact on Evaluation Metrics. Assuming stability, another concern remains: *do we trade off (or simply influence) performance, resource usage, or any other metric for porting time?* This is most relevant as early OS prototypes must be able to compare, in a sound manner, properties with full-fledged baseline OSes. We show that, although the majority of system calls do not influence performance metrics when stubbed, faked, or partially supported, there *are* pitfalls: even when reliably passing tests, these techniques can result in visible performance or resource usage variations (§A.5.3). Loupe improves on the state of the art, which does not consider this problem, by evaluating these costs systematically and early, to provide strong evidence that achieved support does not impact chosen metrics. Still, it remains impossible to formally guarantee that these metrics will be unaffected in all cases. We believe that this too constitutes a reasonable trade-off in development stages.

Overall, dynamic analysis, stubbing, faking, and partial support should not be seen as end-goals for production-ready compatibility, but as a transitional, "necessary evil" in development phases. The takeaway of this paper should not be that most of the system call API is irrelevant, or that static analysis is impertinent in engineering compatibility layers; each corresponds to distinct life cycle phases in the development of new OS. As we show, static analysis is not appropriate in earlier stages, however its output should decisively be a target in later stages of development, and full support should eventually come to achieve high levels of correctness assurance.

Looking forward, we plan to improve Loupe with support for other analysis metrics, such as identifying standard application-specific logs and error message formats, or network and file system usage statistics, to better detect silent faults and effects of stubbing, faking, and partial support techniques. We believe that there remain many interesting research opportunities in application analysis for compatibility that should be explored in future works. Future research avenues include exploring speeding up the analysis by transferring knowledge across applications, and generally using machine learning techniques to identify patterns over the data set, at scale, and generating application-specific workloads.

A.7 Related Work

OS Compatibility Layers. Many research and prototype OSes have implemented compatibility layers to transparently support legacy software. An early example [119] presents a compatibility layer for Linux applications implemented in the K42 [284] OS. Similarly to our work, the authors note that to be widely adopted, an OS must provide good support for existing applications, and that emulating the Linux API is the best way to achieve this goal without requiring modification of target applications. In another study [242], researchers propose a POSIX compatibility layer for the Embassies [241] system. This work presents the construction of the compatibility layer, which is realized in a fully ad-hoc way. As we demonstrate, this process can be highly optimized with Loupe. Still, the authors make some observations similar to ours, in particular the fact that some system calls are "failure-oblivious" (i.e., they can be stubbed) and others are "neutered" (they can be faked). Other works proposed compatibility layers for new monolithic [54, 52, 102], LibOS [430, 432, 285, 276, 352, 286, 39] or micro-kernels [36, 76], web browsers [369], for running applications within the Linux kernel [423], as well as various OS interoperability layers for existing kernels [428, 99, 74, 54, 105]. To the best of our knowledge, all these compatibility layers have been developed in an organic way.

Libc-Based Compatibility Layers. Some works [60, 276] approach compatibility at the libc level, instead of the system call API. Though most system calls are performed through the libc, prior works have shown that interfacing at the libc level leads to weaker degrees of compatibility [353] because many programs do issue system calls outside the libc (500+ ELF Debian 10 executables fall into that category [353]). Thus, we focus on compatibility at the system call level.

Linux & POSIX APIs Studies. Past work studied the usage of the Linux [431] and POSIX [124] APIs by applications. Tsai et al. [431] use binary static analysis to measure the system calls and pseudo-files required by a large set of binaries from the Ubuntu 15.04 archive. Even for the most minimal Ubuntu installation, the study reports that 224 system calls, 208 `ioctl`/`prctl`/`fcntl` codes and 100+ pseudo-files require support. Our results demonstrate that static binary analysis is overly pessimistic. Using dynamic analysis, Loupe shows that the amount of OS features required to run standard benchmarks or even full test suites is actually much lower.

Another study [124] leverages both static and dynamic analysis to measure applications' POSIX API usage. Unlike this work, the authors' goal is not to determine and

optimize compatibility efforts, but to study the evolution of the POSIX interface and identify emerging/missing abstractions. Though the study provides valuable insights for building a compatibility layer at the POSIX (i.e., libc) level [276, 286], past studies showed that the Linux API (mainly system calls) provided a higher degree of compatibility [353]: the authors themselves [124] note that many applications (e.g., Go apps) circumvent POSIX to use OS specific APIs.

A.8 Conclusion

We propose Loupe, an efficient method to determine and prioritize OS features new compatibility layers should implement to provide support for as many applications as possible, as early as possible. Applying Loupe to 100+ applications, we provide examples of support plans, demonstrate high engineering effort savings, and study our measurements in depth. A significant number of system calls identified as needed by previous works are actually not required for those applications to run. These results bring a message of hope to the level of compatibility a new OS must provide in order to support mainstream applications, and should provide encouragement to ongoing and future research OS development efforts. Loupe and our results are available online under an open-source license at `https://github.com/unikraft/loupe` and `https://github.com/unikraft/loupedb`.[3]

Acknowledgments

We thank the anonymous reviewers and our shepherd, Donald E. Porter, for their insights. This work was funded by a studentship from NEC Labs Europe, a Microsoft Research PhD Fellowship, UK's EPSRC grants EP/V012134/1 (UniFaaS), EP/V000225/1 (SCorCH), and the EPSRC/Innovate UK grant EP/X015610/1 (FlexCap), as well as EU H2020 grants 825377 (UNICORE), 871793 (ACCORDION) and 758815 (CORNET). UPB authors were supported by VMWare gift funding.

[3] See Appendix C.

Appendix B

Towards (Really) Safe and Fast Confidential I/O

This chapter takes the format of a position paper where we present a discussion of the problem of safe Input/Output (I/O) interfaces in the context of confidential cloud computing [391]. We review its implications and challenges, and devise research paths to achieve confidential I/O interfaces that are both safer and faster.

These contributions sketch some of the future works we envision to explore following this thesis. Put in the context of this thesis, we envision to apply the learnings of Chapter 4, and particularly the patterns drawn in Section 4.6, to real-world interfaces as part of an effort to define "safe by construction" virtual I/O host-guest device boundaries. This Appendix chapter is derived from Lefeuvre et al. 2023 [301].

Contributions of the Author

I gathered and analysed the data presented in this publication, and wrote the majority of the paper. David Chisnall contributed ideas that influenced this paper. Marios Kogias and Pierre Olivier provided feedback and contributed edits, particularly in B.1 and B.3.4.

Abstract

Confidential cloud computing enables cloud tenants to distrust their service provider. Achieving confidential computing solutions that provide concrete security guarantees requires not only strong mechanisms, but also carefully designed software interfaces. In this paper, we make the observation that confidential I/O interfaces, caught in the tug-of-war between performance and security, fail to address both at a time when confronted to interface vulnerabilities and observability by the untrusted host. We discuss the problem of safe I/O interfaces in confidential computing, its implications and challenges, and devise research paths to achieve confidential I/O interfaces that are both safe and fast.

B.1 Introduction

Confidential cloud computing [391] enables tenants to distrust their service provider. By encrypting data and manipulating it within a Trusted Execution Environment (TEE), confidential computing technologies can guarantee that even strong attackers (e.g., insider threats) controlling the hypervisor/cloud infrastructure remain unable to access tenant data. As a solution to the problem of data privacy [380] in public clouds, there is a very strong demand for confidential computing across the industry and regulatory bodies [203].

Achieving confidential computing solutions that provide concrete security guarantees requires 1) robust isolation, encryption and attestation mechanisms for enforcing the integrity and confidentiality of data throughout its lifetime and 2) carefully designed software interfaces between trusted and untrusted components. Both are critical: there is no confidentiality without strong mechanisms [374, 309, 457], but by themselves they are insufficient to protect a TEE in the presence of unsafe software interfaces [436, 161, 270, 298].

There are two dominant confidential computing paradigms: enclaves [177], that offer abstractions to run parts of an application inside the TEE; and confidential VMs [2, 48], that turn the popular VM abstraction confidential. These paradigms guarantee data confidentiality and integrity during compute. There are independent solutions covering similar guarantees for data at rest [426, 427] and in transit [381, 123, 219]. Yet, *the transitions between those states*, happening through I/O interfaces, remain challenging and potential attack vectors.

We make the following key observation: unlike TEE mechanisms, the topic of confidential I/O interface safety has been underexplored and even explicitly ignored in prior works, e.g., rkt-io [424] and ShieldBox [429] use unhardened (or partially hardened) DPDK drivers for network communication. This is concerning: as we show in this paper, the design of safe and efficient I/O interfaces is particularly difficult and error-prone. Unfortunately, previous works aimed at TEE [436] and compartment [298, 244, 344, 227] interfaces is helpful in characterizing the problem but too broad or generic to provide solutions for efficient and safe I/O interfaces. This highlights a need for research in that domain.

This problem is particularly challenging because the design space is vast. Independently of the underlying mechanism (enclaves/confidential VMs), there are two main differences compared to prior work on interface safety [298, 244, 227, 436]. First, at the host end of the TEE there can be an application-specific interface providing I/O

services, as implemented by different confidential computing frameworks [122, 390, 407, 136, 159, 160, 214]; a paravirtualized device; or a directly attached hardware device. Second, I/O depends on deep protocol stacks that allow for different separation of concerns between trusted/untrusted components, leading to different confidentiality guarantees, performance, and porting effort.

The status quo of confidential I/O interface design is not ideal. Intra-enclave library OSes/TEE frameworks [122, 159, 390, 407] limit the complexity of the interface with the untrusted host by internally managing as many system features as possible, but suffer from an increased TEE TCB size [113]. Further, I/O still needs to go through the host which leads to non-negligible performance losses [159, 407]. To address that issue, other systems offer direct hardware access to the enclave [429, 127, 424]. Not only does this solution not scale to large numbers of TEEs, it is also problematic in terms of security [239]: as we highlight in this paper, driver interfaces have neither been designed nor implemented for mutual distrust, and retrofitting distrust into them is hard. The issue is similar for confidential VMs, where traditional device drivers [210] lack the hardening necessary for confidential workloads [239]. In the case of confidential VMs, the problem is even more significant as they are widely publicized as lift-and-shift solutions reusing traditional driver stacks [210].

Coming up with a satisfying solution is hard because of the fundamental tug-of-war between security and performance: security benefits from simpler interfaces, copies, and checks, whereas performance benefits from lower-level, highly configurable interfaces with zero copies/checks.

In this paper we argue that securing confidential I/O interfaces requires to 1) carefully identify where the host/TEE trust boundary should be located in the I/O stack, and to 2) achieve a host/TEE interface that is safe by construction (as opposed to relying on ad-hoc checks). We discuss alternative boundaries across the I/O stack (§B.2.4), based on which we propose a ternary trust model between the confidential application, the I/O stack, and the device (§B.3.1). To achieve interfaces that are safe by construction, we advocate for alternative interfaces: analyzing the hardening effort in popular paravirtualized drivers (§B.2.5), we hint that hardening existing interfaces may not be the way to go for achieving safe and fast confidential I/O. Focusing on networking, we conclude providing design guidelines to achieve safe confidential I/O and reach different performance, compatibility, and confidentiality trade-offs (§B.3.2).

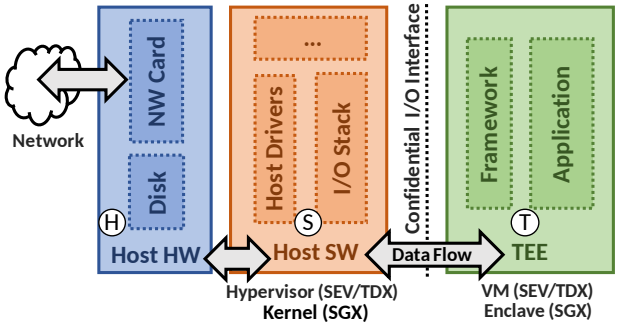


Figure B.1: Key components of confidential computing architectures and confidential I/O boundary.

B.2 Challenges of Confidential I/O Interfaces

B.2.1 Trust Model

Confidential Workload The confidential workload (Ⓣ in Figure B.1) is composed of an application (or part of it), and of a confidential computing framework. The nature of the framework depends on the type of TEE: in the case of enclaves, the framework wraps interactions with the untrusted kernel Ⓢ and other parts of the application, while in the case of confidential VMs, the framework is the trusted OS that runs on top of the untrusted hypervisor Ⓢ. All elements of Ⓣ are trusted in the general case, although, as we describe in §B.3, more fine-grained trust relationships can be achieved with compartmentalization. *We aim to protect Ⓣ's confidentiality and integrity.* We consider Denial of Service (DoS) out of scope, as TEE mechanisms themselves do not support it. We consider architectural side-channels [146, 221, 232, 435, 465] and mechanism-related vulnerabilities [457, 374, 309] out of scope, as mitigations are orthogonal to our aims.

Host Software The host software Ⓢ has a mutual distrust relationship with Ⓣ; i.e., Ⓢ does not trust Ⓣ and Ⓣ does not trust Ⓢ. In the general case, Ⓢ includes an I/O stack and drivers to interface with the host hardware Ⓗ. In the case of enclaves, Ⓢ is an operating system, while in the case of confidential VMs it is a hypervisor. The boundary between Ⓣ and Ⓢ includes the confidential I/O interface.

Host Hardware The host hardware Ⓗ is distrusted by Ⓣ in the general case. It includes networking hardware, disks, etc. In the case of direct device assignment, as we discuss in §B.3.4, part of Ⓗ may be trusted (or partly trusted) by Ⓣ.

B.2.2 Ideal Confidential I/O Properties

Ideal: Interface Safety by Construction. Confidential I/O designs must guarantee the confidentiality and integrity of the confidential workload by construction. This means designing with two main vulnerability vectors in mind:

- *Interface vulnerabilities*: the design of the I/O boundary must avoid or at least minimize vulnerabilities triggered through interface misuses [436, 161, 270, 298], which lead to information leakage, data corruption, as well as temporal violations [298] at the exposed protocol layers.
- *Observability by the host*: the design of the I/O boundary must minimize the amount of non-architectural side-channels exposed to the host (e.g., I/O metadata, ordering and types of I/O calls), as these allow the host to infer information about the TEE [113].

We develop on these two vectors in the following sections. Unlike traditional I/O interfaces, the host is not trusted, making it even more challenging to achieve high performance and strong security properties.

Ideal: High Performance. Confidential I/O designs must be capable of handling high throughput and low latencies. With networking this means saturating a link, i.e., tens of Gbit/s [429, 191]. For storage, this also means throughputs of 10+ Gbit/s [215].

Ideal: Independence from TEE Implementation. I/O designs should be able to plug with the different scenarios highlighted by §B.2.1. I/O services can be provided by an OS kernel as well as by a hypervisor, leveraging a range of technologies (e.g., enclaves, confidential VMs).

Relaxed: Backwards Compatibility. Confidential workloads exhibit generally relaxed backwards compatibility requirements. The cloud infrastructure can be adapted to cater for safer confidential workloads: major cloud players are members of the Confidential Computing Consortium [172]. This makes it reasonable to change the hypervisor, or even push towards hardware changes. For application- and OS- code, compatibility is generally desirable [159, 313] but reasonable to trade-off for security and/or performance, ideally in an iterative manner. There is general awareness that getting the most out of TEEs cannot be achieved without changes across the stack [436, 391], and the community has shown willingness to perform these changes [390].

B.2.3 Confidential I/O: Divide and Rule

Given the previously-mentioned requirements, we propose to divide the problem in two parts: 1) at what abstraction level to place the I/O trust boundary between Ⓣ and Ⓢ, and 2) how to design the I/O interface at the selected level.

P1: Where to Position the I/O Trust Boundary? The trust boundary between the host and the TEE can be placed at various levels in the I/O stack. This influences the nature of the data flowing through the interface. With networking, for instance, data can be raw packets from the network, a TLS-encrypted[1] TCP flow, raw UDP packets, etc. For storage, it can be filesystem operations, block I/Os, etc. Placing the trust boundary at different levels will expose different trade-offs influencing both interface vulnerabilities and observability: important metrics impacted will be the size of the TCB in the confidential domain (e.g., does it contain or not a large TCP/IP stack or a filesystem [113]), as well as the complexity of the driver (e.g., if we get rid of packet layers, we can shave off sources of complexity like negotiation of MTU, who handles the checksum, MAC address, etc.). Having control over the overall design of the interface, *we have the ability to choose at what protocol level we want to position the trust boundary.*

P2: How to Design the I/O Interface Itself? The design of the I/O interface itself defines how data is exchanged over the interface. Many I/O interface designs have been proposed over the years; MMIO-exposed registers, ring buffers, indirect descriptors, various approaches for zero-copy I/O, etc. For example, different approaches may or may not share pointers and/or indexes in safe manners, e.g., via different numbers of indirection layers, resulting in more or less resilience to interface vulnerabilities [298]. Beyond this, different approaches show different control/data path complexity, resulting in varying degrees of statefulness and complexity of error paths, all again impacting interface resilience [298].

We now go through these two problems in §B.2.4 (P1) and B.2.5 (P2), highlighting limitations of previous works and motivating the research paths we propose in §B.3.

B.2.4 P1: I/O Trust Boundary Location

A range of approaches have been explored regarding P1. Enclave approaches that perform networking via the system call interface [159, 390, 122, 407] operate at OSI layer 5

[1]Note: in this Chapter we refer to TLS as Transport Layer Security [381] (not Thread Local Storage).

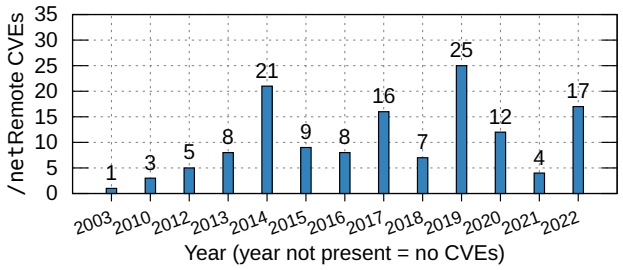


Figure B.2: Remotely-exploitable CVEs in the Linux `/net` subsystem over the years.

(TCP/UDP flows), as exposed by the OS's socket abstraction. However, almost all high-performance approaches [429, 424, 367, 210] work at layer 2, exchanging raw Ethernet packets, processed by the TEE's own I/O stack. Some works [191] also explore tunneled approaches, encapsulating layer 2 packets into a TLS tunnel from a safe network, to hide metadata from the confidential unit's untrusted host and network.

There are valid reasons for most works to focus on layer 2 (L2). First, performance-sensitive approaches build on device pass-through (enclaves), or high-performance paravirtualized devices (confidential VMs), which both require raw Ethernet packets. Second, operating at higher layers means that more observability [113] is exposed to the host, such as the timings of `accept`, or the socket configuration options. At layer 2, the amount of observability is equivalent to what one could obtain by simply observing the network.

On the other hand, there are also arguments to place the boundary higher up at layer 5 (L5). Even though the network subsystem (TCP/IP stack, drivers) has been hardened against network threats over the years, it is ever-growing (about +20% LoC per major version), and remains widely affected by remotely-exploitable vulnerabilities (Figure B.2). Placing it within the confidential workload's TCB is concerning and a clear violation of the principle of least privilege. Assuming a secure interface design (P2), if layer 2 is chosen for P1, it is on the I/O stack that attackers will focus, and its compromise will jeopardize the efforts deployed on securing the I/O boundary. This also provides strong arguments towards a confidential I/O boundary at layer 5, assuming suitable performance can be achieved.

Both layers 2 and 5 are suitable to position the I/O boundary, as the interfaces can be hardened. This is not the case of layer 6, above TLS: although similar argumentation on vulnerability/size could be made for TLS, assuming an untrusted TLS component negates the confidentiality of data and opens for numerous interface vulnerabilities, as the ordering and integrity of payloads is not guaranteed anymore.

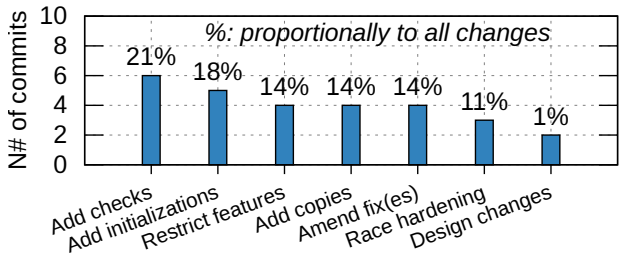


Figure B.3: Distribution of hardening commits to the Linux paravirtualized networking `netvsc` driver.

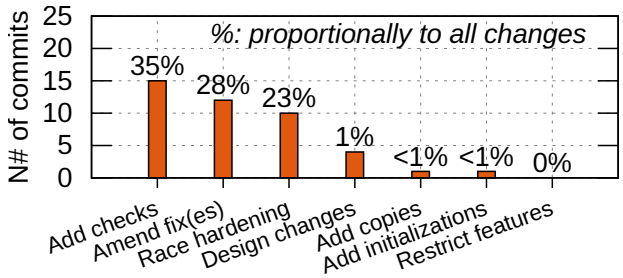


Figure B.4: Distribution of hardening commits to the Linux `virtio` paravirtualized driver family.

We focused here on networking for the sake of brevity, but most observations also apply to storage that similarly has high-level (e.g., filesystem operations) and low-level (e.g., block I/Os) components. Overall, there is no confidential I/O stack approach that combines low TCB (e.g., layer 5 boundary), low observability (e.g., layer 2 boundary), and high performance (rather bound to layer 2); we sketch an approach that relies on a dual-boundary system in §B.3.

B.2.5 P2: Designing a Secure I/O Interface

P2 clearly sparks two alternative resolution approaches: hardening existing I/O interfaces, or defining new ones.

Regarding hardening existing interfaces, we observe ongoing efforts on hardening paravirtualized device driver implementations in Linux [448, 363]. These efforts aim to retrofit mutual distrust into paravirtual I/O device drivers, while remaining compliant to the standards (which were *not* designed with distrust in mind). As we discuss here, this approach is fundamentally limited by the need for backwards compatibility, and by the broad scoping of the standards, ultimately impacting security and performance.

In order to understand the implications and benefits of hardening existing drivers,

and identify characteristics that should be focused on in a from-scratch approach, we systematically record all ongoing hardening efforts performed on the VirtIO [389] and NetVSC [439] Linux paravirtual device drivers. We classify each (merged) commit according to the type of change performed. We record 7 types of changes: adding checks, adding initialization to memory, adding copies, protecting against races, restricting features, performing design changes, or amending previous hardening commits. Figures B.3 and B.4 show the distribution of commits by category.[2]

We make four key observations. First, hardening is extremely error-prone. In the VirtIO case for example, over 40 commits, 12 either revert or amend previous hardening changes, some of them never to be re-applied. Second, much of this complexity is coming from the need to support legacy compatibility features and implementation behaviors. In several cases this need for backwards compatibility prevails, causing the revert of hardening features. Third, the complexity also stems from the large scope of the paravirtual standards. For instance, VirtIO strives to accommodate both paravirtual scenarios and hardware implementations all the same. This results in increased complexity on the control (or configuration) path, with extensive, stateful configuration protocols that open for non-trivial timing and ordering vulnerabilities [298]. The VirtIO standard for example supports at least two alternative *virtqueue* (data transport mechanism) formats, each featuring unique hardening needs. Finally, performance tends to suffer from the hardening more than needed: this is due to disabling security features that bring performance benefits (e.g., in NetVSC), or piggybacking copies on the protocol when the latter wasn't thought with them in mind (SWIOTLB [103] in Linux, applied to VirtIO and NetVSC, which copies systematically even in cases where double fetch is impossible). This increased cost is hard to avoid when retrofitting distrust within the frame of a standard that does not mandate it.

Since they are structural and bound to the designs and motivations of standards, these observations also apply to unhardened driver implementations such as DPDK or SPDK, both popular among enclaves [429, 367, 424, 127]. Other works also explore from-scratch approaches [191], but none of them with a focus on interface vulnerabilities; as a result, even these are prone to similar observations. Overall, no existing stack tackles interface vulnerabilities by design while maintaining performance. There is a need for more research to come up with principled I/O interfaces that, by conception, are resilient against interface threats, which we sketch in §B.3.

[2]The raw data for Figures B.2 to B.4, along with relevant scripts, is available open-source at `https://github.com/hlef/cio-hotos23-data`. See Appendix C.

B.3 Making Confidential I/O Right

We propose a novel confidential I/O design that addresses the limitations identified in the previous section. Different I/O types may require different boundary design decisions, thus for space reasons this section focuses on networking. We discuss a generalization of our principles in §B.3.3.

B.3.1 P1: Plugging at the Right Interface Level

We identify two candidate layers to position the confidential I/O boundary: L2 (raw Ethernet packet) and L5 (TLS-encrypted TCP flow). As discussed previously, both can reasonably be hardened or designed with safety in mind, but neither is ideal: L2 implies a large TCB (I/O stack) in the confidential unit, and L5 results in richer observability by the host, among others.

To address that issue, we propose a *dual-boundary* approach that yields the best of both worlds: we explore an API design combining a strong boundary at L2, limiting observability, and a lightweight one at L5, excluding the I/O stack from the core TCB. This results in a ternary/nested trust model: the set of the I/O stack and the rest of the confidential unit (including the confidential application) does not trust the outside world (host), *yet the I/O stack is not trusted by the rest of the confidential unit either.* The result is an approach that combines 1) low observability, since a powerful attacker on the host does not have access to more information than it would by monitoring the network; with 2) a significantly lower TCB, raising the bar to compromise the confidential workload: compromising the I/O stack, whether through protocol or interface vulnerabilities, only results in increased observability. The host must now mount multi-stage attacks to compromise the confidential application. We represent this intermediate area in Figure B.5, alongside other solutions.

Many design approaches can be taken to enforce the dual boundary. The I/O stack and the rest of the confidential unit could run in two separate TEEs (e.g., two enclaves); however such an additional heavyweight protection domain switch on the I/O path would unnecessarily hurt latency by introducing a dual distrust boundary at L5 where only *single* distrust is needed, as the I/O stack trusts the rest of the confidential unit. Thus, a compartment-based approach [454, 344, 299] relying on low-latency memory isolation techniques within the TEE [407, 394, 225] is more appropriate. In this case, the L2 boundary is a strong host-TEE boundary, and the L5 boundary is a lightweight intra-TEE compartment boundary. Performance is preserved, and security is enhanced compared to an approach without protection at L5.

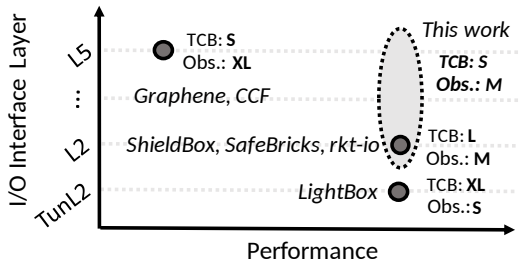


Figure B.5: Leverage control over P1 to reduce TCB and observability (Obs.) without hurting performance.

B.3.2 P2: Achieving Strong I/O Boundaries

B.3.2.1 Hardening L2

In order to achieve a host-TEE boundary at L2 that is by conception resilient to interface vulnerabilities, we propose a design based on five main principles:

- *Stateless Interface*: there are no dependency relationships across and within data/-control plane operations. This considerably simplifies safety reasoning. This means, among others, eliminating error paths whenever possible: in a networked confidential workload for instance, failure to `bind()` likely should be fatal.
- *Copy as a first-class citizen.* Copies are part of the protocol: they are performed early, but only when necessary, and avoided when possible (e.g., when double fetches are absent by design). This improves over legacy approaches where the copy is typically piggybacked everywhere [103].
- *No notifications*: these introduce concurrency that is difficult to protect (Figure B.4), and do not contribute to performance under polling scenarios. When polling is not an option, this must be relaxed; notification handlers are then designed stateless, idempotent, and thread-safe.
- *Zero (re-)negotiation*: parameters like MAC address, MTU size, or who calculates checksums are known at device startup and can be fixed during deployment. This contributes to a minimal control plane. This does not fundamentally preclude live migration, as devices can be hot-swapped; nevertheless, migration without downtime [441] remains difficult as it introduces statefulness in the protocol that is difficult to secure.

- *Safe ring buffer & shared data area*: by construction, pointers to the ring buffer and shared-data area must be protected via careful pointer/index masking [169]: the size of shared memory areas and buffers along with their alignment is designed to make these operations efficient/possible (e.g., alignment at a power of two).

Further, we explore avenues to maintain high-performance.

- *Explore data positioning*. Several designs should be explored: inlining data on the ring buffer together with descriptors, separately in shared memory with mask-protected pointers, or mask-protected indirect descriptors. If buffers are stored separately, buffer freeing should be safe by conception, e.g., via control messages, or with a host-TEE shared memory allocator designed for distrust [84].

- *Explore revocation.* On the receiving side, we explore avoiding copies safely by un-sharing pages with the host on the fly; *we explore when this becomes faster than copies, and how to integrate it harmoniously in interface semantics.*

B.3.2.2 Hardening L5

We design L5 to be resilient by design to interface vulnerabilities based on the following principles:

- *Avoid the need to verify pointers.* Leverage the single distrust to enforce a *trusted component allocates* policy [298]: the application allocates directly in the I/O domain when sending, and provides the buffer when receiving.

- A mandatory TLS layer guarantees data integrity and confidentiality, notably against attempts to break TCP guarantees (e.g., replay attacks, out of order packets, etc.).

Here too we explore a range of performance design choices:

- *How can we achieve zero-copy send on the confidential side?* Here too, leverage the single distrust relationship between the I/O stack and the confidential workload.

- On the receiving side we may need a copy at L5 if we do not trust the I/O stack. Here too we propose to *explore revocation*, as discussed for L2, to eliminate that copy.

B.3.3 Discussion: Beyond Networking

The two-boundary solution to P1 (§B.3.1) should map well to other I/O boundaries that also have observability problems, e.g., storage [113]. Here, the first boundary would be at a low-level interface, e.g., disk driver or block layer, and the second one at a higher level such as file operations. Our approach to P2 (§B.3.2) at both layers should also transfer well to other types of I/O. However, it is likely that it will not always be possible to rewrite drivers at the lowest level (e.g, GPU). In this case, rethinking the answer to P1 should be considered: positioning the boundary just after the driver by compartmentalizing it may be a valid solution as well. Compartmentalizing is realistic: it has been advocated to secure drivers for many years [420, 306, 145, 348] and recently it is getting increasingly automated [248].

B.3.4 Discussion: Direct Device Assignment

Even though Direct Device Assignment (DDA) suffers from the same problems, i.e., the device itself can be malicious, and the communication channel between the TEE and the device is exposed to a malicious host, the hardware community has taken a different path in dealing with the problem: extending PCIe, used to interconnect I/O devices, with support for secure communication and device attestation. Given that the TEE can attest the device, it can trust it/add it to its TCB. Also, given that the communication channel between the TEE and the attested device is encrypted/integrity-protected, there is no need to harden drivers.

Specifically, the TEE Device Interface Security Protocol describes an architecture for confidential communication between TEEs and PCI-attached devices. Here, PCIe exchanges are encrypted using IDE (Integrity & Data Encryption) and there is an attestation protocol between the TEE and device according to SPDM [83] (Security Protocol and Data Model). Different TEE implementations need to implement those PCIe specs, e.g., Intel's TDX [85], through a combination of software and hardware mechanisms.

Despite the performance benefits of directly attached devices, there are limitations in the granularity of partitioning them; DDA is not a silver-bullet, especially with high oversubscription, which paravirtual devices can tackle. Security-wise, DDA is not perfect either: even trusted/attested devices can be compromised (particularly as their complexity is increasing [205]), and adding them to the trusted TCB is a trade-off by itself. SR-IOV-specific attacks have also been described in the past [132, 410, 477, 220, 473], although they are generally limited to availability threats [410, 477, 220] and covert channels [132].

B.4 Conclusion

We highlighted the challenging and often ignored problem of I/O interface safety for TEEs. Studying the hardening effort in widely used systems by the open source community, we show that hardening existing interfaces can only lead to a dead end given the mismatch between the threat model in confidential computing and the threat model in traditional virtualization. Instead, we propose domain-specific interfaces that are easier to harden, can offer both confidential and high-performance I/O, and can be implemented through a ternary trust model between the confidential application, the I/O stack, and the paravirtual device.

Acknowledgments

We thank the anonymous reviewers for their insights. We are also grateful to Jason Wang and Istvan Haller for their insightful feedback. This work was partly funded by a studentship from NEC Labs Europe, a Microsoft Research PhD Fellowship, the UK's EPSRC grants EP/V012134/1 (UniFaaS), EP/V000225/1 (SCorCH), and the EPSRC/Innovate UK grant EP/X015610/1 (FlexCap).

Appendix C

Reproducibility of this Thesis

Great attention was dedicated to making all research artefacts of this thesis available, functional, and reproducible. Next, we link each chapter to their relevant artefacts.

Chapter 3's artefact comprises of the source code of FlexOS, along with all scripts necessary to reproduce the chapter's measurements and plots. It was awarded the *available*, *functional*, and *reproduced* badges, and a *Distinguished Artefact Award* by the ASPLOS 2022 Artefact Evaluation Committee. The artefact is available on:

- GitHub: `https://github.com/project-flexos/asplos22-ae`
- Zenodo: `https://zenodo.org/records/5902507`

Chapter 4's artefact comprises of the source code of the ConfFuzz, along with the data set discussed in the chapter, and all scripts necessary to reproduce the chapter's measurements and plots. The artefact did not go through artefact evaluation, as NDSS 2023 did not have an artefact evaluation phase. The artefact is available on:

- GitHub: `https://github.com/conffuzz/conffuzz`
- Zenodo: `https://zenodo.org/records/7509209`

Chapter 5 does not have an artefact, being of an exclusively theoretical nature.

Appendix A's artefact comprises of the source code of Loupe, along with the data set discussed in the chapter, and all scripts necessary to reproduce the chapter's measurements and plots. It was awarded the *available*, *functional*, and *reproduced* badges by the ASPLOS 2024 Artefact Evaluation Committee. The artefact is available on:

- GitHub: `https://github.com/unikraft/loupe`
- Zenodo: `https://zenodo.org/records/8386116`

Appendix B's artefact comprises of the data discussed in the chapter, and all scripts necessary to reproduce the chapter's plots. The artefact did not go through artefact evaluation, as HotOS 2023 did not have an artefact evaluation phase. The artefact is available on:

- GitHub: `https://github.com/hlef/cio-hotos23-data`

Bibliography

[1] AMD® PRO security: AMD secure processor. https://www.amd.com/en/technologies/pro-security. Viewed: 08/02/23. [Cited on page 141.]

[2] AMD® secure encrypted virtualization (SEV). https://developer.amd.com/sev/. Viewed: 08/02/23. [Cited on pages 141 and 201.]

[3] Apache MPM worker. https://httpd.apache.org/docs/2.4/mod/worker.html. Viewed: 18/03/23. [Cited on page 146.]

[4] Applying ConfFuzz to Unikraft APIs. https://github.com/conffuzz/conffuzz-ndss-data/blob/main/docs/unikraft.md. Viewed: 07/11/23. [Cited on page 80.]

[5] ARM® Cortex-R4/R4F MPU. https://developer.arm.com/documentation/ddi0363/g/Memory-Protection-Unit/About-the-MPU. Viewed: 08/02/23. [Cited on pages 36, 141, 143, and 144.]

[6] ARM® TrustZone for Cortex-A. https://www.arm.com/technologies/trustzone-for-cortex-a. Viewed: 08/02/23. [Cited on pages 141 and 144.]

[7] BSDCan 2018: slaacd(8). https://www.openbsd.org/papers/florian_slaacd_bsdcan2018.pdf. Viewed: 24/03/23. [Cited on page 147.]

[8] Cscope: developer's tool for browsing source code. http://cscope.sourceforge.net/. Viewed: 22/12/21. [Cited on page 57.]

[9] CVE-2006-5794: Sandbox escape via authentication bypass in OpenSSH. https://nvd.nist.gov/vuln/detail/CVE-2006-5794. Viewed: 25/09/23. [Cited on page 147.]

[10] CVE-2016-1247: worker escape vulnerability in nginx. https://nvd.nist.gov/vuln/detail/CVE-2016-1247. Viewed: 24/10/23. [Cited on page 147.]

[11] CVE-2021-23017: mitigated by workers in nginx. https://nvd.nist.gov/vuln/detail/CVE-2021-23017. Viewed: 24/10/23. [Cited on pages 18 and 147.]

[12] CVE-2021-44228: RCE vulnerability in Log4j. https://nvd.nist.gov/vuln/detail/CVE-2021-44228. Viewed: 24/10/23. [Cited on page 120.]

[13] CVE-2022-0847 (Dirty Pipe): RCE vulnerability in Linux. https://nvd.nist.gov/vuln/detail/CVE-2022-0847. Viewed: 24/10/23. [Cited on page 120.]

[14] CVE-2022-22675: RCE vulnerability in Apple macOS/iOS. https://nvd.nist.gov/vuln/detail/CVE-2022-22675. Viewed: 24/10/23. [Cited on page 120.]

[15] CVE-2022-27881: mitigated by sandboxing in slaacd. https://nvd.nist.gov/vuln/detail/CVE-2022-27881. Viewed: 24/10/23. [Cited on pages 18 and 147.]

[16] CVE-2022-27882: mitigated by sandboxing in slaacd. https://nvd.nist.gov/vuln/detail/CVE-2022-27882. Viewed: 24/10/23. [Cited on pages 18 and 147.]

[17] CVE-2022-2852: Use-after-free in Google Chrome. https://nvd.nist.gov/vuln/detail/CVE-2022-2852. Viewed: 24/10/23. [Cited on page 120.]

[18] CVE-2022-41743: mitigated by workers in nginx. https://nvd.nist.gov/vuln/detail/CVE-2022-41743. Viewed: 24/10/23. [Cited on pages 18 and 147.]

[19] CVE-2022-43995: OOB read vulnerability in sudo. https://nvd.nist.gov/vuln/detail/CVE-2022-43995. Viewed: 24/10/23. [Cited on pages 110 and 158.]

[20] CVE-2023-25136: Double-free vulnerability in OpenSSH. https://seclists.org/oss-sec/2023/q1/92. Viewed: 24/03/23. [Cited on pages 18 and 147.]

[21] CVE-2023-28879: RCE vulnerability in Ghostscript. https://nvd.nist.gov/vuln/detail/CVE-2023-43115. Viewed: 24/10/23. [Cited on page 27.]

[22] CVE-2023-28879: RCE vulnerability in Ghostscript. https://nvd.nist.gov/vuln/detail/CVE-2023-28879. Viewed: 24/10/23. [Cited on page 27.]

[23] CVE-2023-42753: RCE vulnerability in Linux. https://nvd.nist.gov/vuln/detail/CVE-2023-42753. Viewed: 24/10/23. [Cited on page 120.]

[24] CVE-2023-5217: Heap buffer overflow in Google Chrome. https://nvd.nist.gov/vuln/detail/CVE-2023-5217. Viewed: 24/10/23. [Cited on page 120.]

[25] debhelper(7) – the debhelper tool suite. https://www.man7.org/linux/man-pages/man7/debhelper.7.html. Viewed: 08/17/23. [Cited on page 175.]

[26] The Debian package archive. https://packages.debian.org. Viewed: 24/03/23. [Cited on pages 145 and 155.]

[27] Debian popularity contest: map the usage of Debian packages. https://popcon.debian.org/. Viewed: 24/03/23. [Cited on pages 145 and 155.]

[28] dh_auto_test(1) – automatically runs a package's test suites. https://manpages.debian.org/testing/debhelper/dh_auto_test.1.en.html. Viewed: 08/17/23. [Cited on page 175.]

[29] DHCPCD: DHCP and DHCPv6 client. https://roy.marples.name/projects/dhcpcd/. Viewed: 24/03/23. [Cited on page 146.]

[30] Dovecot processes. https://doc.dovecot.org/developer_manual/design/processes/#dovecot-processes. Viewed: 25/09/23. [Cited on page 146.]

[31] Enhanced security through MTE. https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/enhanced-security-through-mte. Viewed: 06/04/2023. [Cited on pages 36 and 141.]

[32] errno(3) – number of last error (including a description of `-ENOSYS`). https://www.man7.org/linux/man-pages/man3/errno.3.html. Viewed: 08/01/23. [Cited on pages 165 and 169.]

[33] Ghostscript: an interpreter for the PostScript language and PDF files. https://www.ghostscript.com/. Viewed: 24/10/23. [Cited on page 27.]

[34] Google Chrome: Site isolation bypass via NavigationPreloadRequest. https://bugs.chromium.org/p/project-zero/issues/detail?id=2239. Viewed: 24/09/23. [Cited on page 147.]

[35] Google: Code sandboxing: SAPI. https://developers.google.com/code-sandboxing/sandboxed-api. Viewed: 22/02/23. [Cited on pages 30, 35, 128, 134, and 137.]

[36] Google Fuchsia website. https://fuchsia.dev/. Viewed: 08/01/23. [Cited on pages 165, 166, 167, 168, 177, and 197.]

[37] Google V8 untrusted code mitigations. https://v8.dev/docs/untrusted-code-mitigations. Viewed: 25/09/23. [Cited on page 146.]

[38] Gramine: a library OS for unmodified applications. https://gramineproject.io. Viewed: 08/10/23. [Cited on page 177.]

[39] gVisor (Google) GitHub repository. https://github.com/google/gvisor. Viewed: 05/03/2018. [Cited on pages 165, 167, 168, 177, and 197.]

[40] How SQLite is tested. https://www.sqlite.org/testing.html. Viewed: 08/17/23. [Cited on pages 175 and 176.]

[41] IBM® z/OS concepts: What is storage protection? https://www.ibm.com/docs/en/zos-basic-skills?topic=storage-what-is-protection. Viewed: 08/02/23. [Cited on pages 36, 141, and 143.]

[42] ImageMagick: an open-source software suite for editing and manipulating digital images. https://imagemagick.org/. Viewed: 24/10/23. [Cited on page 27.]

[43] Inside NGINX: How we designed for performance & scale. https://nginx.com/blog/inside-nginx-how-we-designed-for-performance-scale/. Viewed: 18/03/23. [Cited on page 146.]

[44] Intel® 64 and IA-32 arch. SDM. Vol. 3A, Sec. 4.6.1.: ACCESS RIGHTS: Determination of Access Rights. Version: December 2022. [Cited on pages 36, 141, and 143.]

[45] Intel® 64 and IA-32 arch. SDM. Vol. 3C, Sec. 26.5.6: FEATURES SPECIFIC TO VMX NON-ROOT OPERATION: VM Functions. Version: December 2022. [Cited on pages 36 and 141.]

[46] Intel® 64 and IA-32 arch. SDM. Vol. 3A, Sec. 4.6.2.: ACCESS RIGHTS: Protection Keys. Version: December 2022. [Cited on pages 36, 60, 141, 142, 143, 144, and 145.]

[47] Intel® software guard extensions. https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions.html. Viewed: 08/02/23. [Cited on page 141.]

[48] Intel® trust domain extensions. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html. Viewed: 08/02/23. [Cited on pages 141 and 201.]

[49] Introducing Arm confidential compute architecture V1. https://developer.arm.com/documentation/den0125/0100/Arm-CCA-extensions. Viewed: 08/02/23. [Cited on page 141.]

[50] iPerf - the ultimate speed test tool for TCP, UDP and SCTP. https://iperf.fr/iperf-doc.php. Viewed: 08/01/23. [Cited on page 189.]

[51] Irssi chat client. https://irssi.org/. Viewed: 24/03/23. [Cited on page 146.]

[52] Kerla GitHub repository: A new operating system kernel with Linux binary compatibility written in Rust. https://github.com/nuta/kerla. Viewed: 08/01/23. [Cited on pages 165, 166, 167, 168, 176, 177, and 197.]

[53] Lighttpd Angel configuration. https://doc.lighttpd.net/lighttpd2/core_config_angel.html. Viewed: 18/03/23. [Cited on page 146.]

[54] Linuxulator (Linux emulation): running unmodified Linux binaries under FreeBSD. https://wiki.freebsd.org/Linuxulator. Viewed: 08/01/23. [Cited on pages 165, 167, 168, 177, and 197.]

[55] madvise(2) – give advice about use of memory. https://man7.org/linux/man-pages/man2/madvise.2.html. Viewed: 08/01/23. [Cited on page 190.]

[56] man-db, the on-line manual database. https://man-db.nongnu.org/. Viewed: 24/03/23. [Cited on page 146.]

[57] Monza GitHub Repository: Research unikernel for virtualized services. https://github.com/microsoft/monza. Viewed: 22/02/23. [Cited on pages 34, 134, and 137.]

[58] Mozilla RLBox. https://github.com/mozilla/gecko-dev/tree/master/third_party/rlbox. Viewed: 18/03/23. [Cited on page 146.]

[59] MS Windows NT kernel-mode user and GDI white paper. https://learn.microsoft.com/en-us/previous-versions//cc750820(v=technet.10). Viewed: 25/09/23. [Cited on page 147.]

[60] Newlib: a C library intended for use on embedded systems. https://sourceware.org/newlib/. Viewed: 12/12/2017. [Cited on page 197.]

[61] Nginx documentation: Configuring logging. https://docs.nginx.com/nginx/admin-guide/monitoring/logging/. Viewed: 08/17/23. [Cited on page 189.]

[62] npm package registry. https://www.npmjs.com/. Viewed: 22/02/23. [Cited on page 29.]

[63] nscd(8) – name service cache daemon. https://www.man7.org/linux/man-pages/man8/nscd.8.html. Viewed: 08/17/23. [Cited on page 185.]

[64] OpenBenchmarking.org software repository. https://openbenchmarking.org/. Viewed: 08/01/23. [Cited on page 176.]

[65] OpenBSD innovations. https://www.openbsd.org/innovations.html. Viewed: 24/03/23. [Cited on pages 38 and 146.]

[66] OpenSSH 7.0: Fix to sandbox escape. https://www.openssh.com/txt/release-7.0. Viewed: 24/03/23. [Cited on page 147.]

[67] OpenSSH 7.4: Fix to sandbox escape. https://www.openwall.com/lists/oss-security/2016/12/19/2. Viewed: 24/03/23. [Cited on page 147.]

[68] OpenSSH 9.0: near miss in sshd(8). https://www.openbsd.org/71.html. Viewed: 24/09/23. [Cited on pages 18 and 147.]

[69] OpenSSH breaking due to overly restrictive syscall policy. https://github.com/NixOS/nixpkgs/issues/157451. Viewed: 23/10/03. [Cited on page 145.]

[70] OpenSSH git commit `6283f4bd83ee`: "allow writev in sandbox". https://github.com/openssh/openssh-portable/commit/6283f4bd83ee. Viewed: 23/03/23. [Cited on page 145.]

[71] OSv applications GitHub repository. https://github.com/cloudius-systems/osv-apps. Viewed: 08/01/23. [Cited on pages 165, 167, 177, and 178.]

[72] pledge(2) – restrict system operations. https://man.openbsd.org/pledge.2. Viewed: 24/03/23. [Cited on page 140.]

[73] Privilege separation. https://www.openbsd.org/papers/auug04/mgp00030.html. Viewed: 24/03/23. [Cited on page 147.]

[74] Proton (Valve Software) GitHub repository. https://github.com/ValveSoftware/Proton. Viewed: 08/01/23. [Cited on pages 165, 167, 168, and 197.]

[75] ptrace(2) – process trace. https://man7.org/linux/man-pages/man2/ptrace.2.html. Viewed: 07/31/2023. [Cited on pages 166, 171, and 172.]

[76] ReactOS GitHub repository: A free Windows-compatible Operating System. https://github.com/reactos/reactos. Viewed: 08/01/23. [Cited on pages 165, 167, 168, and 197.]

[77] Redis benchmark: Using the redis-benchmark utility on a Redis server. https://redis.io/docs/management/optimization/benchmarks/. Viewed: 08/01/23. [Cited on page 189.]

[78] Redis test suite. https://github.com/redis/redis/tree/unstable/tests. Viewed: 08/17/23. [Cited on page 175.]

[79] Rumprun Packages GitHub repository: Ready-made packages of software for running on the Rumprun unikernel. https://github.com/rumpkernel/rumprun-packages. Viewed: 08/01/23. [Cited on pages 165, 167, and 177.]

[80] Rust: A language empowering everyone to build reliable and efficient software. www.rust-lang.org/. Viewed: 22/02/23. [Cited on page 141.]

[81] Sandbox libexpat using RLBox (incl. performance discussion). https://bugzilla.mozilla.org/show_bug.cgi?id=1688452. Viewed: 25/09/23. [Cited on page 147.]

[82] seccomp(2) – operate on secure computing state of the process. https://man7.org/linux/man-pages/man2/seccomp.2.html. Viewed: 07/31/2023. [Cited on pages 28, 122, 127, 140, 153, 155, 166, 171, and 172.]

[83] Security Protocol and Data Model (SPDM) specification. https://www.dmtf.org/sites/default/files/standards/documents/DSP0274_1.2.1.pdf. Viewed: 28/01/23. [Cited on page 212.]

[84] snmalloc (Microsoft) GitHub repository: Message passing based allocator. https://github.com/microsoft/snmalloc/tree/main/docs/security. Viewed: 01/02/23. [Cited on page 211.]

[85] Software enabling for Intel® TDX in support of TEE-I/O. https://cdrdv2.intel.com/v1/dl/getContent/742542. Viewed: 28/01/23. [Cited on page 212.]

[86] SPARC application data integrity (ADI). https://www.kernel.org/doc/Documentation/sparc/adi.rst. Viewed: 06/04/23. [Cited on pages 36 and 141.]

[87] strace - Linux syscall tracer. https://strace.io/. Viewed: 08/17/23. [Cited on page 169.]

[88] sudo - execute a command as another user. https://www.sudo.ws/. Viewed: 18/11/23. [Cited on page 109.]

[89] Support for Intel® memory protection extensions. https://www.intel.com/content/www/us/en/support/articles/000059823/processors.html. Viewed: 08/02/23. [Cited on pages 36, 42, 141, and 142.]

[90] sysfs: filesystem for exporting kernel objects. https://www.kernel.org/doc/html/latest/filesystems/sysfs.html. Viewed: 18/03/23. [Cited on page 140.]

[91] Systematization of knowledge papers. https://www.ieee-security.org/TC/SP2024/cfpapers.html#systematization-of-knowledge-papers. Viewed: 18/11/23. [Cited on page 117.]

[92] TCPDUMP: command-line packet analyzer. https://www.tcpdump.org/. Viewed: 24/03/23. [Cited on page 153.]

[93] The `/proc` Filesystem. https://www.kernel.org/doc/html/latest/filesystems/proc.html. Viewed: 18/03/23. [Cited on page 140.]

[94] Unikraft application repository: Applications supported by the Unikraft LibOS. https://github.com/orgs/unikraft/repositories. Viewed: 08/01/23. [Cited on page 177.]

[95] Unikraft *linuxu* debug platform. https://github.com/unikraft/unikraft/tree/staging/plat/linuxu. Viewed: 07/11/23. [Cited on pages 73 and 74.]

[96] Unikraft static binary analysis tool (part of the Loupe artifact). https://github.com/unikraft/loupe/tree/staging/src/static-binary-analyser. Viewed: 08/17/23. [Cited on page 181.]

[97] Unikraft static source analysis tool (part of the Loupe artifact). https://github.com/unikraft/loupe/tree/staging/src/static-source-analyser. Viewed: 08/17/23. [Cited on page 181.]

[98] vsftpd: secure and fast FTP server for UNIX-like systems. https://security.appspot.com/vsftpd.html. Viewed: 24/03/23. [Cited on page 146.]

[99] Wine HQ – a compatibility layer to run Windows applications on POSIX. https://www.winehq.org/about. Viewed: 08/01/23. [Cited on pages 165, 167, 168, and 197.]

[100] wrk GitHub repository: a HTTP benchmarking tool. https://github.com/wg/wrk. Viewed: 08/01/23. [Cited on page 189.]

[101] Zathura PDF reader seccomp sandbox. https://git.pwmt.org/pwmt/zathura/-/blob/develop/zathura/zathura.c. Viewed: 24/03/23. [Cited on page 153.]

[102] Zephyr Project: A proven RTOS ecosystem. https://www.zephyrproject.org/. Viewed: 08/01/23. [Cited on pages 165, 167, 168, 177, and 197.]

[103] Linux git commit `d7b417fa08d1`: "x86/mm: Add DMA support for SEV memory encryption". https://github.com/torvalds/linux/commit/d7b417fa08d1187923c270bc33a3555c2fcff8b9, 2017. Viewed: 01/02/23. [Cited on pages 208 and 210.]

[104] OSv GitHub repository: stub of `io_setup`. https://github.com/cloudius-systems/osv/blob/317d259ab5b0b49a1a114bc837147746e471abc9/core/libaio.cc#L17, 2021. Viewed: 08/21/2022. [Cited on page 178.]

[105] Docker desktop for Mac - support for running x86-64 binaries with Rosetta 2. https://github.com/docker/roadmap/issues/384, 2022. Viewed: 08/01/23. [Cited on pages 165, 167, and 197.]

[106] Open Enclave SDK webpage. https://openenclave.io/sdk/, 2022. Viewed: 23/12/22. [Cited on pages 84 and 115.]

[107] OSS-Fuzz GitHub repository. https://google.github.io/oss-fuzz/, 2022. Viewed: 23/12/22. [Cited on page 94.]

[108] Unikraft: The lightweight virtualization company. https://unikraft.io, 2023. Viewed: 18/11/23. [Cited on page 161.]

[109] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security*, 13(1), November 2009. doi:10.1145/1609956.1609960. [Cited on page 50.]

[110] Yasemin Acar, Michael Backes, Sven Bugiel, Sascha Fahl, Patrick McDaniel, and Matthew Smith. SoK: Lessons learned from Android security research for appified software platforms. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy*, S&P'16, pages 433–451, 2016. doi:10.1109/SP.2016.33. [Cited on pages 18, 38, 120, 133, 150, and 153.]

[111] Adobe Systems Inc. *PostScript Language Reference*. Addison-Wesley Publishing Company, 3 edition, 1999. [Cited on page 27.]

[112] Ioannis Agadakos, Manuel Egele, and William Robertson. Polytope: Practical memory access control for C++ applications. *arXiv pre-print*, 2022. doi:10.48550/arXiv.2201.08461. [Cited on pages 82, 91, 97, 108, and 115.]

[113] Adil Ahmad, Kyungtae Kim, Muhammad Ihsanulhaq Sarfaraz, and Byoungyoung Lee. OBLIVIATE: A data oblivious filesystem for Intel SGX. In *Proceedings of the 25th Annual Network & Distributed System Security Symposium*, NDSS'18, 2018. doi:10.14722/ndss.2018.23284. [Cited on pages 202, 204, 205, 206, and 212.]

[114] Hesham Almatary, Michael Dodson, Jessica Clarke, Peter Rugg, Ivan Gomes, Michal Podhradsky, Peter G. Neumann, Simon W. Moore, and Robert N. M. Watson. CompartOS: CHERI compartmentalization for embedded systems. *arXiv pre-print*, 2022. doi:10.48550/ARXIV.2206.02852. [Cited on pages 35, 82, 128, 131, 134, 135, and 137.]

[115] Hussain M. J. Almohri and David Evans. Fidelius Charm: Isolating unsafe Rust code. In *Proceedings of the 8th ACM Conference on Data and Application Security and Privacy*, CODASPY'18, pages 248–255, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3176258.3176330. [Cited on page 29.]

[116] J. Alves-Foss, P. Oman, C. Taylor, and S. Harrison. The MILS architecture for high-assurance embedded systems. *International Journal of Embedded Systems*, 2:239–247, 2006. doi:10.1504/IJES.2006.014859. [Cited on pages 77 and 120.]

[117] S.R. Jr Ames, R. Schell, and M. Gasser. Security kernel design and

implementation: An introduction. *Computer*, 16(07):14–22, jul 1983. doi:10.1109/MC.1983.1654439. [Cited on page 120.]

[118] James P. Anderson. Computer security technology planning study. Technical report, 1972. FORT WASHINGTON PA. [Cited on page 127.]

[119] Jonathan Appavoo, Marc Auslander, Dilma Da Silva, David Edelsohn, Orran Krieger, Michal Ostrowski, Bryan Rosenburg, Robert Wisniewski, and Jimi Xenidis. Providing a Linux API on the scalable K42 kernel. In *Proceedings of the 2003 USENIX Annual Technical Conference, FREENIX Track*, ATC'03. USENIX Association, 2003. [Cited on pages 165, 167, 168, 170, and 197.]

[120] Arm Ltd. Arm Morello Program. Viewed: 25/06/20. URL: https://developer.arm.com/architectures/cpu-architecture/a-profile/morello. [Cited on pages 37, 49, 82, 89, and 115.]

[121] ARM Ltd. Building a secure system using TrustZone technology, April 2009. Viewed: 24/01/21. URL: https://developer.arm.com/documentation/genc009492/c. [Cited on pages 49, 52, 55, and 82.]

[122] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. SCONE: Secure linux containers with Intel SGX. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'16, pages 689–703. USENIX Association, 2016. [Cited on pages 202 and 205.]

[123] Randall Atkinson. Security architecture for the Internet protocol. https://www.rfc-editor.org/rfc/rfc1825, 1995. Viewed: 28/01/23. [Cited on page 201.]

[124] Vaggelis Atlidakis, Jeremy Andrus, Roxana Geambasu, Dimitris Mitropoulos, and Jason Nieh. POSIX abstractions in modern operating systems: The old, the new, and the missing. In *Proceedings of the 11th European Conference on Computer Systems*, EuroSys'16. Association for Computing Machinery, 2016. doi:10.1145/2901318.2901350. [Cited on pages 168, 197, and 198.]

[125] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):124–142, jan 1995. doi:10.1145/200836.200869. [Cited on pages 35 and 136.]

[126] Michael Backes, Sven Bugiel, Christian Hammer, Oliver Schranz, and Philipp von Styp-Rekowsky. Boxify: Full-fledged app sandboxing for stock Android. In *Proceedings of the 24th USENIX Security Symposium*, USENIX Security'15, pages 691–706, Washington, D.C., August 2015. USENIX Association. [Cited on pages 28 and 122.]

[127] Maurice Bailleu, Jörg Thalheim, Pramod Bhatotia, Christof Fetzer, Michio Honda, and Kapil Vaswani. SPEICHER: Securing LSM-based Key-Value stores using shielded execution. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies*, FAST'19, pages 173–190. USENIX Association, 2019. [Cited on pages 202 and 208.]

[128] Inyoung Bang, Martin Kayondo, Hyungon Moon, and Yunheung Paek. TRUST: A compilation framework for in-process isolation to protect safe Rust against untrusted code. In *Proceedings of the 32nd USENIX Security Symposium*, USENIX Security'23. USENIX Association, 2023. [Cited on pages 29, 120, and 122.]

[129] Steve Bannister. Memory tagging extension: Enhancing memory safety through architecture, August 2019. Viewed: 27/10/20. URL: `https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/enhancing-memory-safety`. [Cited on page 60.]

[130] Antonio Barbalace, Robert Lyerly, Christopher Jelesnianski, Anthony Carno, Ho-Ren Chuang, Vincent Legout, and Binoy Ravindran. Breaking the boundaries in heterogeneous-ISA datacenters. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'17. Association for Computing Machinery, 2017. `doi:10.1145/3093337.3037738`. [Cited on page 167.]

[131] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, SOSP'03, pages 164–177, New York, NY, USA, 2003. Association for Computing Machinery. `doi:10.1145/945445.945462`. [Cited on pages 28 and 122.]

[132] Adam Bates, Benjamin Mood, Joe Pletcher, Hannah Pruse, Masoud Valafar, and Kevin Butler. On detecting co-resident cloud instances using network flow watermarking techniques. *International Journal of Information Security*, 13:171–189, 2014. `doi:10.1007/s10207-013-0210-0`. [Cited on page 212.]

[133] Markus Bauer and Christian Rossow. Cali: Compiler assisted library isolation. In *Proceedings of the 16th ACM Asia Conference on Computer and Communications Security*, AsiaCCS'21. Association for Computing Machinery, 2021. doi:10.1145/3433210.3453111. [Cited on pages 18, 33, 35, 39, 58, 61, 77, 82, 85, 91, 97, 108, 111, 115, 120, 128, 130, 131, 134, and 157.]

[134] Andrew Baumann, Jonathan Appavoo, Orran Krieger, and Timothy Roscoe. A fork() in the road. In *Proceedings of the 17th Workshop on Hot Topics in Operating Systems*, HotOS'19, pages 14–22, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3317550.3321435. [Cited on pages 35, 139, 140, 148, and 149.]

[135] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the 22nd Symposium on Operating Systems Principles*, SOSP'09, pages 29–44, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1629575.1629579. [Cited on page 167.]

[136] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with Haven. *ACM Transactions on Computer Systems*, 33(3), aug 2015. doi:10.1145/2799647. [Cited on page 202.]

[137] Daniel J. Bernstein. Some thoughts on security after ten years of qmail 1.0. In *Proceedings of the 1st ACM Workshop on Computer Security Architecture*, CSAW'07, New York, NY, USA, 2007. Association for Computing Machinery. doi:10.1145/1314466.1314467. [Cited on pages 32, 120, 145, and 146.]

[138] Atri Bhattacharyya, Uros Tesic, and Mathias Payer. Midas: Systematic kernel TOCTTOU protection. In *Proceedings of the 31st USENIX Security Symposium*, USENIX Security'22, Boston, MA, August 2022. USENIX Association. [Cited on pages 87 and 90.]

[139] Simon Biggs, Damon Lee, and Gernot Heiser. The jury is in: Monolithic OS design is flawed: Microkernel-based designs improve security. In *Proceedings of the 9th Asia-Pacific Workshop on Systems*, APSys'18, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3265723.3265733. [Cited on pages 41 and 127.]

[140] Leyla Bilge and Tudor Dumitraş. Before we knew it: An empirical study of zero-day attacks in the real world. In *Proceedings of the 19th ACM Conference on Computer and Communications Security*, CCS'12, pages 833–844, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2382196.2382284. [Cited on pages 17 and 27.]

[141] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting applications into reduced-privilege compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 309–322, USA, 2008. USENIX Association. [Cited on pages 18, 120, 128, 134, 140, and 157.]

[142] Kevin Boos, Namitha Liyanage, Ramla Ijaz, and Lin Zhong. Theseus: an experiment in operating system structure and state management. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'20. USENIX Association, November 2020. [Cited on page 77.]

[143] Kevin Boos, Emilio Del Vecchio, and Lin Zhong. A characterization of state spill in modern operating systems. In *Proceedings of the 12th European Conference on Computer Systems*, EuroSys'17, pages 389–404, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3064176.3064205. [Cited on page 137.]

[144] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel, 3rd Edition*. O'Reilly Media, Inc., 3rd edition, November 2005. [Cited on page 49.]

[145] Silas Boyd-Wickizer and Nickolai Zeldovich. Tolerating malicious device drivers in Linux. In *Proceedings of the 2010 USENIX Annual Technical Conference*, ATC'10, pages 117–130, USA, 2010. USENIX Association. [Cited on pages 18, 77, 120, 122, and 212.]

[146] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *Proceedings of the 11th USENIX Conference on Offensive Technologies*, WOOT'17, page 11, USA, 2017. USENIX Association. [Cited on page 203.]

[147] David Brumley and Dawn Song. Privtrans: Automatically partitioning programs for privilege separation. In *Proceedings of the 13th USENIX Security Symposium*, USENIX Security'04, USA, 2004. USENIX Association. [Cited on pages 18, 120, 128, 130, 132, 134, and 157.]

[148] John Bruno, José Brustoloni, Eran Gabber, Avi Silberschatz, and Christopher Small. Pebble: A component-based operating system for embedded applications. In *Proceedings of the Embedded Systems Workshop*, WOES'99, USA, 1999. USENIX Association. [Cited on page 54.]

[149] Alexander Bulekov, Rasoul Jahanshahi, and Manuel Egele. Saphire: Sandboxing PHP applications with tailored system call allowlists. In *Proceedings of the 30th USENIX Security Symposium*, USENIX Security'21. USENIX Association, 2021. [Cited on page 168.]

[150] N. Burow, X. Zhang, and M. Payer. SoK: Shining light on shadow stacks. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy*, S&P'19, pages 985–999, 2019. doi:10.1109/SP.2019.00076. [Cited on pages 29 and 120.]

[151] Canadian Centre for Cyber Security. National cyber threat assessment 2023/2024. https://www.cyber.gc.ca/sites/default/files/ncta-2023-24-web.pdf. Viewed: 07/11/23. [Cited on page 17.]

[152] Claudio Canella, Mario Werner, Daniel Gruss, and Michael Schwarz. Automating seccomp filter generation for Linux applications. In *Proceedings of the 12th Workshop on Cloud Computing Security*, CCSW'21, pages 139–151, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3474123.3486762. [Cited on pages 140 and 168.]

[153] Scott A. Carr and Mathias Payer. DataShield: Configurable data confidentiality and integrity. In *Proceedings of the 12th ACM Asia Conference on Computer and Communications Security*, AsiaCCS'17, pages 193–204, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3052973.3052983. [Cited on pages 128, 130, and 134.]

[154] Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. Hardware support for fast capability-based addressing. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VI, pages 319–327, New York, NY, USA, 1994. Association for Computing Machinery. doi:10.1145/195473.195579. [Cited on pages 36, 141, and 143.]

[155] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. Fast byte-granularity software fault isolation. In *Proceedings of the 22nd ACM Symposium*

on Operating Systems Principles, SOSP'09, pages 45–58, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1629575.1629581. [Cited on pages 36, 77, and 141.]

[156] Sunjay Cauligi, Craig Disselkoen, Daniel Moghimi, Gilles Barthe, and Deian Stefan. SoK: Practical foundations for software Spectre defenses. In *Proceedings of the 2022 IEEE Symposium on Security and Privacy*, S&P'22, 2022. doi:10.1109/SP46214.2022.9833707. [Cited on pages 30 and 150.]

[157] David Cerdeira, Nuno Santos, Pedro Fonseca, and Sandro Pinto. SoK: Understanding the prevailing security vulnerabilities in TrustZone-assisted TEE systems. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy*, S&P'20, pages 1416–1432, 2020. doi:10.1109/SP40000.2020.00061. [Cited on pages 40 and 150.]

[158] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. Sharing and protection in a single-address-space operating system. *ACM Transactions on Computer Systems*, 12(4):271–307, November 1994. doi:10.1145/195792.195795. [Cited on pages 36, 49, 77, 120, and 141.]

[159] Chia che Tsai, Donald E. Porter, and Mona Vij. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *Proceedings of the 2017 USENIX Annual Technical Conference*, pages 645–658, Santa Clara, CA, July 2017. USENIX Association. [Cited on pages 84, 115, 143, 202, 204, and 205.]

[160] Chia che Tsai, Jeongseok Son, Bhushan Jain, John McAvey, Raluca Ada Popa, and Donald E. Porter. Civet: An efficient Java partitioning framework for hardware enclaves. In *Proceedings of the 29th USENIX Security Symposium*, USENIX Security'20. USENIX Association, 2020. [Cited on page 202.]

[161] Stephen Checkoway and Hovav Shacham. Iago attacks: Why the system call API is a bad untrusted RPC interface. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'13, pages 253–264, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2451116.2451145. [Cited on pages 40, 43, 60, 82, 84, 85, 86, 114, 122, 153, 201, and 204.]

[162] Hao Chen, David Wagner, and Drew Dean. Setuid demystified. In *Proceedings of the 11th USENIX Security Symposium*, USENIX Security'02, San Francisco, CA, August 2002. USENIX Association. [Cited on page 30.]

[163] Shuo Chen, David Ross, and Yi-Min Wang. An analysis of browser domain-isolation bugs and a light-weight transparent defense mechanism. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS'07, pages 2–11, New York, NY, USA, 2007. Association for Computing Machinery. doi:10.1145/1315245.1315248. [Cited on pages 18, 38, 140, 146, and 150.]

[164] Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar, and Ravishankar K Iyer. Non-control-data attacks are realistic threats. In *Proceedings of the 14th USENIX Security Symposium*, USENIX Security'05, page 146. USENIX Association, 2005. [Cited on page 89.]

[165] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, New York, NY, USA, 2008. Association for Computing Machinery. doi:10.1145/1346281.1346284. [Cited on pages 28 and 122.]

[166] Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. Shreds: Fine-grained execution units with private memory. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy*, S&P'16, pages 56–71, 2016. doi:10.1109/SP.2016.12. [Cited on pages 18, 35, 40, 120, 128, 134, 136, and 157.]

[167] W. H. Cheung and Anthony H. S. Loong. Exploring issues of operating systems structuring: From microkernel to extensible systems. *ACM SIGOPS Operating Systems Review*, 29(4):4–16, oct 1995. doi:10.1145/219282.219284. [Cited on page 35.]

[168] Yi Chien, Vlad-Andrei Bădoiu, Yudi Yang, Yuqian Huo, Kelly Kaoudis, Hugo Lefeuvre, Pierre Olivier, and Nathan Dautenhahn. CIVSCOPE: Analyzing potential memory corruption bugs in compartment interfaces. In *Proceedings of the 1st Workshop on Kernel Isolation, Safety and Verification*, KISV'23. Association for Computing Machinery, 2023. doi:10.1145/3625275.3625399. [Cited on pages 25, 153, and 159.]

[169] David Chisnall. The definitive guide to the Xen hypervisor (Section 6.3). https://www.informit.com/articles/article.aspx?p=1160234&seqNum=3, 2008. Viewed: 02/02/23. [Cited on page 211.]

[170] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Secure Web applications via automatic partitioning. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, SOSP'07, pages 31–44, New York, NY, USA, 2007. Association for Computing Machinery. doi:10.1145/1294261.1294265. [Cited on pages 34, 128, 132, and 134.]

[171] Abraham A Clements, Naif Saleh Almakhdhub, Saurabh Bagchi, and Mathias Payer. ACES: Automatic compartments for embedded systems. In *Proceedings of the 27th USENIX Security Symposium*, USENIX Security'18, pages 65–82, Baltimore, MD, August 2018. USENIX Association. [Cited on pages 33, 35, 128, 131, and 134.]

[172] Confidential Computing Consortium. Confidential computing consortium members. https://confidentialcomputing.io/members/, 2023. Viewed: 30/01/23. [Cited on page 204.]

[173] R. Joseph Connor, Tyler McDaniel, Jared M. Smith, and Max Schuchard. PKU pitfalls: Attacks on PKU-based memory isolation systems. In *Proceedings of the 29th USENIX Security Symposium*, USENIX Security'20, pages 1409–1426. USENIX Association, August 2020. [Cited on pages 35, 140, and 150.]

[174] Jonathan Corbet. Memory protection keys. *Linux Weekly News*, 2015. https://lwn.net/Articles/643797/. [Cited on pages 49, 52, and 55.]

[175] Manuel Costa, Miguel Castro, Lidong Zhou, Lintao Zhang, and Marcus Peinado. Bouncer: Securing software by blocking bad input. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles*, SOSP'07, pages 117–130, New York, NY, USA, 2007. Association for Computing Machinery. doi:10.1145/1294261.1294274. [Cited on page 75.]

[176] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: End-to-end containment of Internet worms. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, SOSP'05. Association for Computing Machinery, 2005. doi:10.1145/1095810.1095824. [Cited on page 75.]

[177] Victor Costan and Srinivas Devadas. Intel SGX explained. *IACR Cryptology ePrint Archive pre-print*, 2016(86), 2016. https://eprint.iacr.org/2016/086. [Cited on pages 49, 55, 82, and 201.]

[178] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *Proceedings of the 25th USENIX Conference on Security Symposium*, USENIX Security'16, pages 857–874, USA, 2016. USENIX Association. [Cited on page 141.]

[179] Rongzhen Cui, Lianying Zhao, and David Lie. Emilia: Catching Iago in legacy code. In *Proceedings of the 29th Annual Network & Distributed System Security Symposium*, NDSS'22, 2022. doi:10.14722/ndss.2021.24328. [Cited on pages 43, 82, 84, 113, and 115.]

[180] Cody Cutler, M. Frans Kaashoek, and Robert T. Morris. The benefits and costs of writing a POSIX kernel in a high-level language. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'18, pages 89–105, Carlsbad, CA, October 2018. USENIX Association. [Cited on page 77.]

[181] Robert C Daley and Jack B Dennis. Virtual memory, processes, and sharing in Multics. *Communications of the ACM*, 11(5):306–312, 1968. doi:10.1145/800001.811668. [Cited on pages 120 and 134.]

[182] DARPA. Compartmentalization and privilege management (CPM). https://www.darpa.mil/program/compartmentalization-and-privilege-management. Viewed: 07/11/23. [Cited on page 18.]

[183] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. Nested kernel: An operating system architecture for intra-kernel privilege separation. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'15, pages 191–206, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2694344.2694386. [Cited on pages 77, 134, 139, and 142.]

[184] Brooks Davis, Robert N. M. Watson, Alexander Richardson, Peter G. Neumann, Simon W. Moore, John Baldwin, David Chisnall, James Clarke, Nathaniel Wesley Filardo, Khilan Gudka, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, J. Edward Maste, Alfredo Mazzinghi, Edward Tomasz Napierala, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, and Jonathan Woodruff. CheriABI: Enforcing valid pointer provenance and minimizing pointer privilege in the POSIX C run-time environment. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating*

Systems, ASPLOS'19, pages 379–393, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3297858.3304042. [Cited on page 59.]

[185] Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P. Kemerlis. Sysfilter: Automated system call filtering for commodity software. In *Proceedings of the 23rd International Symposium on Research in Attacks, Intrusions and Defenses*, RAID'20, pages 459–474, San Sebastian, October 2020. USENIX Association. [Cited on pages 140 and 168.]

[186] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. Hardbound: Architectural support for spatial safety of the C programming language. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 103–114, New York, NY, USA, 2008. Association for Computing Machinery. doi:10.1145/1346281.1346295. [Cited on pages 36, 42, and 141.]

[187] Tevi Devor and Sion Berkowits. Pin: Intel's dynamic binary instrumentation engine – Pin CGO 2013 tutorial, 2013. https://www.intel.com/content/dam/develop/external/us/en/documents/cgo2013-256675.pdf. [Cited on page 99.]

[188] Udit Dhawan, Catalin Hritcu, Raphael Rubin, Nikos Vasilakis, Silviu Chiricescu, Jonathan M. Smith, Thomas F. Knight, Benjamin C. Pierce, and Andre DeHon. Architectural support for software-defined metadata processing. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'15, pages 487–502, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2694344.2694383. [Cited on pages 36 and 141.]

[189] Xinshu Dong, Hong Hu, Prateek Saxena, and Zhenkai Liang. A quantitative evaluation of privilege separation in Web browser designs. In Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *Proceedings of the 18th European Symposium on Research in Computer Security*, ESORICS'13, pages 75–93, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. doi:10.1007/978-3-642-40203-6_5. [Cited on page 123.]

[190] Vijay D'Silva, Mathias Payer, and Dawn Song. The correctness-security gap in compiler optimization. In *Proceedings of the 2015 IEEE Security and Privacy Workshops*, LangSec'15, pages 73–87, 2015. doi:10.1109/SPW.2015.33. [Cited on page 150.]

[191] Huayi Duan, Cong Wang, Xingliang Yuan, Yajin Zhou, Qian Wang, and Kui Ren. Lightbox: Full-stack protected stateful middlebox at lightning speed. In *Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security*, CCS'19, pages 2351–2367, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3319535.3339814. [Cited on pages 204, 206, and 208.]

[192] Trevor Dunlap, William Enck, and Bradley Reaves. A study of application sandbox policies in Linux. In *Proceedings of the 27th ACM on Symposium on Access Control Models and Technologies*, SACMAT'22, pages 19–30, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3532105.3535016. [Cited on pages 28, 122, 127, 145, 153, and 155.]

[193] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, and J. Alex Halderman. The matter of Heartbleed. In *Proceedings of the 14th Internet Measurement Conference*, IMC'14, pages 475–488, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2663716.2663755. [Cited on page 131.]

[194] Kha Dinh Duy, Kyuwon Cho, Taehyun Noh, and Hojoon Lee. Capacity: Cryptographically-enforced in-process capabilities for modern ARM architectures. In *Proceedings of the 30th ACM SIGSAC Conference on Computer and Communications Security*, CCS'23. Association for Computing Machinery, 2023. doi:10.1145/3576915.3623079. [Cited on pages 128 and 134.]

[195] DWARF Debugging Information Format Committee. DWARF debugging information formation (Version 5). https://dwarfstd.org/doc/DWARF5.pdf, February 2017. Viewed: 07/11/23. [Cited on page 93.]

[196] Jack Edge. Rust for Linux redux. *Linux Weekly News*, 2021. https://lwn.net/Articles/862018/. [Cited on page 49.]

[197] Jack Edge. Rust heads into the kernel? *Linux Weekly News*, 2021. https://lwn.net/Articles/853423/. [Cited on page 49.]

[198] Rick Edgecombe. [PATCH RFC 0/9] PKS write protected page tables. https://lore.kernel.org/lkml/20210505003032.489164-1-rick.p.edgecombe@intel.com/, 2021. Viewed: 28/01/23. [Cited on page 30.]

[199] Kevin Elphinstone and Gernot Heiser. From L3 to SeL4 what have we learnt in 20 years of L4 microkernels? In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, SOSP'13, pages 133–150, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2517349.2522720. [Cited on pages 30, 32, 120, 134, and 146.]

[200] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, SOSP'95, pages 251–266, New York, NY, USA, 1995. Association for Computing Machinery. doi:10.1145/224056.224076. [Cited on pages 19, 49, 50, and 54.]

[201] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: Software guards for system address spaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI'06, pages 75–88, USA, 2006. USENIX Association. [Cited on pages 40 and 134.]

[202] Lawrence G. Esswood. *CheriOS: designing an untrusted single-address-space capability operating system utilising capability hardware and a minimal hypervisor*. PhD thesis, University of Cambridge Computer Laboratory, September 2021. doi:10.48456/tr-961. [Cited on page 134.]

[203] Everest Global, Inc. Confidential computing - the next frontier in data security. https://confidentialcomputing.io/wp-content/uploads/sites/85/2021/10/Everest_Group_-_Confidential_Computing_-_The_Next_Frontier_in_Data_Security_-_2021-10-19.pdf, 2021. Viewed: 20/12/22. [Cited on page 201.]

[204] Norman Feske. Genode foundations. https://genode.org/documentation/genode-foundations-21-05.pdf, 2021. [Cited on pages 68 and 73.]

[205] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure accelerated networking: SmartNICs in the public cloud. In *Proceedings of the 15th USENIX*

Symposium on Networked Systems Design and Implementation, NSDI'18, pages 51–66, Renton, WA, April 2018. USENIX Association. [Cited on page 212.]

[206] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The Flux OSKit: A substrate for kernel and language research. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, SOSP'97, pages 38–51, New York, NY, USA, 1997. Association for Computing Machinery. doi:10.1145/268998.266642. [Cited on pages 52 and 55.]

[207] Fortanix. Fortanix enclave development platform. https://edp.fortanix.com/, 2022. Viewed: 23/12/22. [Cited on pages 84 and 115.]

[208] Tommaso Frassetto, Patrick Jauernig, Christopher Liebchen, and Ahmad-Reza Sadeghi. IMIX: In-process memory isolation extension. In *Proceedings of the 27th USENIX Conference on Security Symposium*, USENIX Security'18, pages 83–97, USA, 2018. USENIX Association. [Cited on pages 36, 141, and 144.]

[209] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In Oscar M. Nierstrasz, editor, *Proceedings of the 7th European Conference on Object-Oriented Programming*, ECOOP'93, pages 406–431, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg. doi:10.1007/3-540-47910-4_21. [Cited on page 160.]

[210] Xinyang Ge, Hsuan-Chi Kuo, and Weidong Cui. Hecate: Lifting and shifting on-premises workloads to an untrusted cloud. In *Proceedings of the 29th ACM SIGSAC Conference on Computer and Communications Security*, CCS'22, pages 1231–1242, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3548606.3560592. [Cited on pages 143, 202, and 206.]

[211] Alain Gefflaut, Trent Jaeger, Yoonho Park, Jochen Liedtke, Kevin J. Elphinstone, Volkmar Uhlig, Jonathon E. Tidswell, Luke Deller, and Lars Reuther. The SawMill multiserver approach. In *Proceedings of the 9th ACM SIGOPS European Workshop*, EW'00, pages 109–114, New York, NY, USA, 2000. Association for Computing Machinery. doi:10.1145/566726.566751. [Cited on page 52.]

[212] Seyedhamed Ghavamnia, Tapti Palit, Azzedine Benameur, and Michalis Polychronakis. Confine: Automated system call policy generation for container attack surface reduction. In *Proceedings of the 23rd International Symposium on Research in Attacks, Intrusions and Defenses*, RAID'20. USENIX Association, 2020. [Cited on page 168.]

[213] Adrien Ghosn, Marios Kogias, Mathias Payer, James R. Larus, and Edouard Bugnion. Enclosure: Language-based restriction of untrusted libraries. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'21, pages 255–267, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3445814.3446728. [Cited on pages 18, 29, 33, 35, 120, 128, 131, 132, 134, and 140.]

[214] Adrien Ghosn, James R. Larus, and Edouard Bugnion. Secured routines: Language-based construction of trusted execution environments. In *Proceedings of the 2019 USENIX Annual Technical Conference*, ATC'19, pages 571–586, Renton, WA, July 2019. USENIX Association. [Cited on page 202.]

[215] Gigabyte. GIGABYTE leads and reveals the first PCIe 5.0 SSD. https://www.gigabyte.com/Press/News/2017, 2022. [Cited on page 204.]

[216] David B Golub, Daniel P Julin, Richard F Rashid, Richard P Draves, Randall W Dean, Alessandro Forin, Joseph Barrera, Hideyuki Tokuda, Gerald Malan, and David Bohman. Microkernel operating system architecture and Mach. In *In Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*. USENIX Association, 1992. [Cited on page 77.]

[217] Li Gong, Marianne Mueller, Hemma Prafullchandra, and Roland Schemers. Going beyond the sandbox: An overview of the new security architecture in the Java development kit 1.2. In *Proceedings of 1st USENIX Symposium on Internet Technologies and Systems*, USITS'97, 1997. [Cited on pages 28, 122, 127, and 153.]

[218] Google. Asylo: An open and flexible framework for enclave applications. https://asylo.dev/, 2022. Viewed: 23/12/22. [Cited on pages 84 and 115.]

[219] Google, Inc. Encryption in transit (whitepaper). https://cloud.google.com/docs/security/encryption-in-transit, 2022. Viewed: 28/01/23. [Cited on page 201.]

[220] Yonatan Gottesman and Yoav Etsion. NeSC: Self-virtualizing nested storage controller. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO'16, 2016. doi:10.1109/MICRO.2016.7783713. [Cited on page 212.]

[221] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems*

Security, EuroSec'17, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3065913.3065915. [Cited on page 203.]

[222] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. KASLR is dead: Long live KASLR. In Eric Bodden, Mathias Payer, and Elias Athanasopoulos, editors, *Proceedings of the 9th International Symposium on Engineering Secure Software and Systems*, ESSoS'17, pages 161–176, Bonn, Germany, 2017. Springer International Publishing. doi:10.1007/978-3-319-62105-0_11. [Cited on page 75.]

[223] Jinyu Gu, Hao Li, Wentai Li, Yubin Xia, and Haibo Chen. EPK: Scalable and efficient memory protection keys. In *Proceedings of the 2022 USENIX Annual Technical Conference*, ATC'22, pages 609–624, Carlsbad, CA, July 2022. USENIX Association. [Cited on pages 141, 144, and 145.]

[224] Jinyu Gu, Xinyue Wu, Wentai Li, Nian Liu, Zeyu Mi, Yubin Xia, and Haibo Chen. Harmonizing performance and isolation in microkernels with efficient intra-kernel isolation and communication. In *Proceedings of the 2020 USENIX Annual Technical Conference*, ATC'20, pages 401–417. USENIX Association, July 2020. [Cited on pages 77 and 146.]

[225] Jinyu Gu, Bojun Zhu, Mingyu Li, Wentai Li, Yubin Xia, and Haibo Chen. A Hardware-Software co-design for efficient Intra-Enclave isolation. In *Proceedings of the 31st USENIX Security Symposium*, USENIX Security'22, pages 3129–3145, Boston, MA, August 2022. USENIX Association. [Cited on pages 18, 97, 108, 120, 157, and 209.]

[226] Le Guan, Jingqiang Lin, Bo Luo, Jiwu Jing, and Jing Wang. Protecting private keys against memory disclosure attacks using hardware transactional memory. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, S&P'15, pages 3–19, 2015. doi:10.1109/SP.2015.8. [Cited on pages 18, 120, and 157.]

[227] Khilan Gudka, Robert N.M. Watson, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Ilias Marinos, Peter G. Neumann, and Alex Richardson. Clean application compartmentalization with SOAAP. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS'15, pages 1016–1031, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2810103.2813611. [Cited on pages 18, 33, 42, 43, 57, 58, 67, 77, 82,

87, 88, 97, 98, 108, 115, 120, 123, 126, 128, 129, 130, 131, 132, 134, 137, 140, 153, 157, and 201.]

[228] Diwaker Gupta, Ludmila Cherkasova, Rob Gardner, and Amin Vahdat. Enforcing performance isolation across virtual machines in Xen. In *Proceedings of the ACM/IFIP/USENIX 7th International Middleware Conference*, Middleware'06, pages 342–362. Springer, 2006. doi:10.1007/11925071_18. [Cited on page 137.]

[229] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the Web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'17. Association for Computing Machinery, 2017. doi:10.1145/3062341.3062363. [Cited on pages 36, 141, and 143.]

[230] Munawar Hafiz, Ralph Johnson, and Raja Afandi. The security architecture of qmail. In *Proceedings of the 11th Conference on Patterns Language of Programming*, PLoP'04, 2004. [Cited on pages 32, 145, and 146.]

[231] Munawar Hafiz and Ralph E. Johnson. Evolution of the MTA architecture: the impact of security. *Software: Practice and Experience*, 38(15):1569–1599, 2008. doi:10.1002/spe.880. [Cited on pages 32, 120, 145, and 146.]

[232] Marcus Hähnel, Weidong Cui, and Marcus Peinado. High-resolution side channels for untrusted operating systems. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, ATC'17, pages 299–312, USA, 2017. USENIX Association. [Cited on page 203.]

[233] Istvan Haller, Yuseok Jeon, Hui Peng, Mathias Payer, Cristiano Giuffrida, Herbert Bos, and Erik Van Der Kouwe. TypeSan: Practical type confusion detection. In *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security*, CCS'16, pages 517–528. Association for Computing Machinery, October 2016. doi:10.1145/2976749.2978405. [Cited on page 86.]

[234] Norm Hardy. The confused deputy: (or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, 22(4):36–38, 1988. doi:10.1145/54289.871709. [Cited on pages 19, 21, 40, 84, 85, 86, 122, and 153.]

[235] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of μ-kernel-based systems. In *Proceedings*

of the 16th ACM Symposium on Operating Systems Principles, SOSP'97, pages 66–77, New York, NY, USA, 1997. Association for Computing Machinery. doi:10.1145/268998.266660. [Cited on page 77.]

[236] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. Hodor: Intra-process isolation for high-throughput data plane libraries. In *Proceedings of the 2019 USENIX Annual Technical Conference*, ATC'19, pages 489–504, Renton, WA, July 2019. USENIX Association. [Cited on pages 34, 36, 53, 61, 77, 82, 91, 115, 134, 140, and 142.]

[237] Gernot Heiser, Kevin Elphinstone, Jerry Vochteloo, Stephen Russell, and Jochen Liedtke. The Mungi single-address-space operating system. *Software: Practice and Experience*, 28(9):901–928, July 1999. doi:10.1002/(SICI)1097-024X(19980725)28:9<901::AID-SPE181>3.0.CO;2-7. [Cited on pages 36, 49, 77, 120, 141, and 143.]

[238] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. MINIX 3: A highly reliable, self-repairing operating system. *ACM SIGOPS Operating Systems Review*, 40(3):80–89, July 2006. doi:10.1145/1151374.1151391. [Cited on pages 30, 32, 49, 77, 120, and 146.]

[239] Felicitas Hetzelt, Martin Radev, Robert Buhren, Mathias Morbitzer, and Jean-Pierre Seifert. VIA: Analyzing device interfaces of protected virtual machines. In *Proceedings of the 37th Annual Computer Security Applications Conference*, ACSAC'21, pages 273–284, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3485832.3488011. [Cited on page 202.]

[240] Owen S. Hofmann, Sangman Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel. InkTag: Secure applications on an untrusted operating system. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'13, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2451116.2451146. [Cited on pages 28, 115, and 122.]

[241] Jon Howell, Bryan Parno, and John R. Douceur. Embassies: Radically refactoring the Web. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'13. USENIX Association, 2013. [Cited on page 197.]

[242] Jon Howell, Bryan Parno, and John R. Douceur. How to run POSIX apps in a minimal picoprocess. In *Proceedings of the 2013 USENIX Annual Technical Conference*, ATC'13. USENIX Association, 2013. [Cited on pages 165, 167, 169, 178, and 197.]

[243] Terry Ching-Hsiang Hsu, Kevin Hoffman, Patrick Eugster, and Mathias Payer. Enforcing least privilege memory views for multithreaded applications. In *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security*, CCS'16, pages 393–405, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2976749.2978327. [Cited on pages 123, 128, 134, 139, and 140.]

[244] Hong Hu, Zheng Leong Chua, Zhenkai Liang, and Prateek Saxena. Identifying arbitrary memory access vulnerabilities in privilege-separated software. In Günther Pernul, Peter Y A Ryan, and Edgar Weippl, editors, *Proceedings of the 20th European Symposium on Research in Computer Security*, ESORICS'15, pages 312–331, Cham, 2015. Springer International Publishing. doi:10.1007/978-3-319-24177-7_16. [Cited on pages 19, 40, 43, 82, 84, 85, 86, 92, 98, 113, 120, 122, 131, 138, 150, 153, and 201.]

[245] Jinhan Hu, Andrei Iosifescu, and Robert LiKamWa. LensCap: Split-process framework for fine-grained visual privacy control for augmented reality apps. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys'21, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3458864.3467676. [Cited on page 133.]

[246] Wei Hu, Armaiti Ardeshiricham, and Ryan Kastner. Hardware information flow tracking. *ACM Computing Surveys*, 54(4), may 2021. doi:10.1145/3447867. [Cited on pages 36, 141, and 150.]

[247] Jie Huang, Oliver Schranz, Sven Bugiel, and Michael Backes. The ART of app compartmentalization: Compiler-based library privilege separation on stock Android. In *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security*, CCS'17, pages 1037–1049, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3133956.3134064. [Cited on pages 29, 128, 132, 133, and 134.]

[248] Yongzhe Huang, Vikram Narayanan, David Detweiler, Kaiming Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. KSplit: Automating device driver isolation. In *Proceedings of the 16th USENIX Symposium on Operating Systems*

Design and Implementation, OSDI'22, pages 613–631, Carlsbad, CA, July 2022. USENIX Association. [Cited on pages 35, 128, 131, 134, 153, and 212.]

[249] Zhen Huang, Trent Jaeger, and Gang Tan. Fine-grained program partitioning for security. In *Proceedings of the 14th European Workshop on Systems Security*, EuroSec'21, pages 21–26, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3447852.3458717. [Cited on page 131.]

[250] Zhen Huang, David Lie, Gang Tan, and Trent Jaeger. Using safety properties to generate vulnerability patches. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy*, S&P'19, pages 539–554, 2019. doi:10.1109/SP.2019.00071. [Cited on page 75.]

[251] Galen C. Hunt and James R. Larus. Singularity: Rethinking the software stack. *ACM SIGOPS Operating Systems Review*, 41(2):37–49, April 2007. doi:10.1145/1243418.1243424. [Cited on page 77.]

[252] Bumjin Im, Fangfei Yang, Chia-Che Tsai, Michael LeMay, Anjo Vahldiek-Oberwagner, and Nathan Dautenhahn. The Endokernel: Fast, secure, and programmable subprocess virtualization. *arXiv pre-print*, 2021. doi:10.48550/arXiv.2108.03705. [Cited on pages 82, 97, 103, 108, and 111.]

[253] Innovate UK. Digital security by design: Securing the future of the digital economy. https://iuk.ktn-uk.org/wp-content/uploads/2021/01/4204_Innovate_Digital-Security_A4_CB_final.pdf. Viewed: 07/11/23. [Cited on page 18.]

[254] Chris Jaikaran. Cybersecurity: Selected cyberattacks, 2012-2022. Technical report, US Congressional Research Service, 2023. URL: https://crsreports.congress.gov/product/pdf/R/R46974. [Cited on page 17.]

[255] Abhinav Jangda, Bobby Powers, Emery D. Berger, and Arjun Guha. Not so fast: Analyzing the performance of WebAssembly vs. native code. In *Proceedings of the 2019 USENIX Annual Technical Conference*, ATC'19, pages 107–120, USA, 2019. USENIX Association. [Cited on page 144.]

[256] Ira Ray Jenkins, Prashant Anantharaman, Rebecca Shapiro, J. Peter Brady, Sergey Bratus, and Sean W. Smith. Ghostbusting: Mitigating Spectre with intraprocess memory isolation. In *Proceedings of the 7th Symposium on Hot Topics in the Science of Security*, HotSoS'20, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3384217.3385627. [Cited on pages 18, 30, and 120.]

[257] Samuel Jero, Nathan Burow, Bryan Ward, Richard Skowyra, Roger Khazan, Howard Shrobe, and Hamed Okhravi. TAG: Tagged architecture guide. *ACM Computing Surveys*, 55(6), dec 2022. doi:10.1145/3533704. [Cited on pages 36, 141, and 150.]

[258] Xuancheng Jin, Xuangan Xiao, Songlin Jia, Wang Gao, Dawu Gu, Hang Zhang, Siqi Ma, Zhiyun Qian, and Juanru Li. Annotating, tracking, and protecting cryptographic secrets with CryptoMPK. In *Proceedings of the 2022 IEEE Symposium on Security and Privacy*, S&P'22, 2022. doi:10.1109/SP46214.2022.9833650. [Cited on pages 18, 120, and 157.]

[259] Evan Johnson, Evan Laufer, Zijie Zhao, Dan Gohman, Shravan Narayan, Stefan Savage, Deian Stefan, and Fraser Brown. WaVe: a verifiably secure WebAssembly sandboxing runtime. In *Proceedings of the 2023 IEEE Symposium on Security and Privacy*, S&P'23, 2023. doi:10.1109/SP46215.2023.10179357. [Cited on pages 36, 141, and 150.]

[260] Evan Johnson, David Thien, Yousef Alhessi, Shravan Narayan, Fraser Brown, Sorin Lerner, Tyler McMullen, Stefan Savage, and Deian Stefan. SFI safety for native-compiled Wasm. In *Proceedings of the 28th Annual Network & Distributed System Security Symposium*, NDSS'21, 2021. [Cited on pages 36, 141, and 150.]

[261] Alexander Jung, Hugo Lefeuvre, Charalampos Rotsos, Pierre Olivier, Daniel Oñoro Rubio, Felipe Huici, and Mathias Niepert. Wayfinder: Towards automatically deriving optimal OS configurations. In *Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys'21, pages 115–122, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3476886.3477506. [Cited on pages 25, 47, 68, 128, and 159.]

[262] M. Frans Kaashoek, Dawson R Engler, Gregory R Ganger, Héctor M Briceno, Russell Hunt, David Mazieres, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application performance and flexibility on Exokernel systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, SOSP'97. Association for Computing Machinery, 1997. doi:10.1145/268998.266644. [Cited on pages 49 and 50.]

[263] Svilen Kanev, Sam Likun Xi, Gu-Yeon Wei, and David Brooks. Mallacc: Accelerating memory allocation. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*,

ASPLOS'17, pages 33–45, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3037697.3037736. [Cited on page 61.]

[264] Antti Kantee. *Flexible Operating System Internals: The Design and Implementation of the Anykernel and Rump Kernels*. PhD thesis, Department of Computer Science and Engineering, Aalto University, 2012. URL: http://urn.fi/URN:ISBN:978-952-60-4917-5. [Cited on page 77.]

[265] Antti Kantee. The rise and fall of the operating system. *USENIX ;login: Magazine*, 40(5), 2015. [Cited on pages 165 and 167.]

[266] C.N. Keltcher, K.J. McGrath, A. Ahmed, and P. Conway. The AMD Opteron processor for multiprocessor servers. *IEEE Micro*, 23(2), 2003. doi:10.1109/MM.2003.1196116. [Cited on page 192.]

[267] Vasileios P. Kemerlis, Michalis Polychronakis, and Angelos D. Keromytis. ret2dir: Rethinking kernel isolation. In *Proceedings of the 23rd USENIX Security Symposium*, USENIX Security'14, pages 957–972, San Diego, CA, August 2014. USENIX Association. [Cited on page 84.]

[268] Christoph Kerschbaumer and Christian Holler. Effectively fuzzing the IPC layer in Firefox. https://blog.mozilla.org/attack-and-defense/2021/01/27/effectively-fuzzing-the-ipc-layer-in-firefox/. Viewed: 25/09/23. [Cited on page 147.]

[269] Arslan Khan, Dongyan Xu, and Dave Jing Tian. EC: Embedded systems compartmentalization via intra-kernel isolation. In *Proceedings of the 2023 IEEE Symposium on Security and Privacy*, S&P'23, pages 2990–3007, 2023. doi:10.1109/SP46215.2023.10179285. [Cited on pages 28, 41, 65, 122, and 153.]

[270] Mustakimur Rahman Khandaker, Yueqiang Cheng, Zhi Wang, and Tao Wei. COIN attacks: On insecurity of enclave untrusted interfaces in SGX. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'20, pages 971–985, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3373376.3378486. [Cited on pages 40, 42, 138, 150, 153, 201, and 204.]

[271] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. Address Space Layout Permutation (ASLP): Towards fine-grained randomization of

commodity software. In *Proceedings of the 22nd Annual Computer Security Applications Conference*, ACSAC'06, pages 339–348, 2006. doi:10.1109/ACSAC.2006.9. [Cited on pages 87 and 88.]

[272] Douglas Kilpatrick. Privman: A library for partitioning applications. In *Proceedings of the 2003 USENIX Annual Technical Conference*, ATC'03, pages 273–284. USENIX Association, 2003. [Cited on pages 49, 120, 123, and 134.]

[273] Yonghae Kim, Jaekyu Lee, and Hyesoon Kim. Hardware-based always-on heap memory safety. In *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO'20, pages 1153–1166, 2020. doi:10.1109/MICRO50266.2020.00095. [Cited on pages 36, 42, and 141.]

[274] Vladimir Kiriansky and Carl Waldspurger. Speculative buffer overflows: Attacks and defenses. *arXiv pre-print*, 2018. doi:10.48550/arXiv.1807.03757. [Cited on page 150.]

[275] Paul Kirth, Mitchel Dickerson, Stephen Crane, Per Larsen, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. PKRU-Safe: Automatically locking down the heap between safe and unsafe languages. In *Proceedings of the 17th European Conference on Computer Systems*, EuroSys'22, pages 132–148, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3492321.3519582. [Cited on pages 29 and 120.]

[276] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. OSv — optimizing the operating system for virtual machines. In *Proceedings of the 2014 USENIX Annual Technical Conference*, ATC'14, pages 61–72. USENIX Association, June 2014. [Cited on pages 77, 165, 166, 167, 168, 177, 178, 197, and 198.]

[277] Chris Kjellqvist, Mohammad Hedayati, and Michael L. Scott. Safe, fast sharing of Memcached as a protected library. In *Proceedings of the 49th International Conference on Parallel Processing*, ICPP'20, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3404397.3404443. [Cited on page 61.]

[278] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. SeL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on*

Operating Systems Principles, SOSP'09, pages 207–220, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1629575.1629596. [Cited on pages 30, 32, 49, 68, 73, 120, and 146.]

[279] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy*, S&P'19, 2019. doi:10.1109/SP.2019.00002. [Cited on page 76.]

[280] Matthew Kolosick, Shravan Narayan, Evan Johnson, Conrad Watt, Michael LeMay, Deepak Garg, Ranjit Jhala, and Deian Stefan. Isolation without taxation: Near-zero-cost transitions for WebAssembly and SFI. *Proceedings of the ACM on Programming Languages*, 6(POPL), jan 2022. doi:10.1145/3498688. [Cited on pages 141 and 144.]

[281] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. No need to hide: Protecting safe regions on commodity hardware. In *Proceedings of the 12th European Conference on Computer Systems*, EuroSys'17, pages 437–452, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3064176.3064217. [Cited on pages 36, 42, 82, 91, 115, 137, 141, 142, and 144.]

[282] Jakob Koschel, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. TagBleed: Breaking KASLR on the isolated kernel address space using tagged TLBs. In *Proceedings of the 2020 IEEE European Symposium on Security and Privacy*, EuroS&P'20, pages 309–321, 2020. doi:10.1109/EuroSP48549.2020.00027. [Cited on page 84.]

[283] John Alistair Kressel, Hugo Lefeuvre, and Pierre Olivier. Software compartmentalization trade-offs with hardware capabilities. In *Proceedings of the 12th Workshop on Programming Languages and Operating Systems*, PLOS'23, pages 49–57, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3623759.3624550. [Cited on pages 25, 37, and 64.]

[284] Orran Krieger, Marc Auslander, Bryan Rosenburg, Robert W. Wisniewski, Jimi Xenidis, Dilma Da Silva, Michal Ostrowski, Jonathan Appavoo, Maria Butrico, Mark Mergen, Amos Waterland, and Volkmar Uhlig. K42: building a complete operating system. In *Proceedings of the 1st European Conference on*

Computer Systems, EuroSys'06. Association for Computing Machinery, 2006. doi:10.1145/1217935.1217949. [Cited on page 197.]

[285] Simon Kuenzer, Vlad-Andrei Bădoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Ştefan Teodorescu, Costi Răducanu, Cristian Banu, Laurent Mathy, Răzvan Deaconescu, Costin Raiciu, and Felipe Huici. Unikraft: Fast, specialized unikernels the easy way. In *Proceedings of the 16th European Conference on Computer Systems*, EuroSys'21, pages 376–394, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3447786.3456248. [Cited on pages 25, 35, 49, 52, 53, 54, 60, 68, 77, 161, 162, 165, 166, 167, 168, 176, 177, and 197.]

[286] Hsuan-Chi Kuo, Dan Williams, Ricardo Koller, and Sibin Mohan. A Linux in unikernel clothing. In *Proceedings of the 15th European Conference on Computer Systems*, EuroSys'20, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3342195.3387526. [Cited on pages 165, 167, 197, and 198.]

[287] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-pointer integrity. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'14, pages 147–163, Broomfield, CO, October 2014. USENIX Association. [Cited on page 30.]

[288] Albert Kwon, Udit Dhawan, Jonathan M. Smith, Thomas F. Knight, and Andre DeHon. Low-fat pointers: Compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. In *Proceedings of the 20th ACM Conference on Computer and Communications Security*, CCS'13, pages 721–732, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2508859.2516713. [Cited on page 37.]

[289] P. Ladisa, H. Plate, M. Martinez, and O. Barais. SoK: Taxonomy of attacks on open-source software supply chains. In *Proceedings of the 2023 IEEE Symposium on Security and Privacy*, S&P'23, pages 167–184, Los Alamitos, CA, USA, may 2023. IEEE Computer Society. doi:10.1109/SP46215.2023.00010. [Cited on pages 29 and 150.]

[290] Benjamin Lamowski, Carsten Weinhold, Adam Lackorzynski, and Hermann Härtig. Sandcrust: Automatic sandboxing of unsafe components in Rust. In *Proceedings of the 9th Workshop on Programming Languages and Operating*

Systems, PLOS'17, pages 51–57, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3144555.3144562. [Cited on page 29.]

[291] Butler W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, oct 1973. doi:10.1145/362375.362389. [Cited on page 152.]

[292] Butler W. Lampson. Protection. *ACM SIGOPS Operating Systems Review*, 8(1):18–24, jan 1974. doi:10.1145/775265.775268. [Cited on pages 84, 91, 135, 136, and 142.]

[293] Michael Larabel. Linux picks up fix for latest "confused deputy" weakness going back to 2.6.12 kernel, 2021. https://www.phoronix.com/scan.php?page=news_item&px=Linux-Confused-Deputy-2.6.12. [Cited on pages 82 and 84.]

[294] Julia Lawall and Gilles Muller. Coccinelle: 10 years of automated evolution in the Linux kernel. In *Proceedings of the 2018 USENIX Annual Technical Conference*, ATC'18, pages 601–614. USENIX Association, 2018. [Cited on page 55.]

[295] Doug Lea. A memory allocator. http://gee.cs.oswego.edu/dl/html/malloc.html, 1996. Viewed: 22/02/23. [Cited on page 74.]

[296] Hojoon Lee, Chihyun Song, and Brent Byunghoon Kang. Lord of the x86 rings: A portable user mode privilege separation architecture on x86. In *Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security*, CCS'18, pages 1441–1454, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3243734.3243748. [Cited on pages 18, 36, 120, 141, and 157.]

[297] Hojoon Lee, Chihyun Song, and Brent Byunghoon Kang. Harnessing the x86 intermediate rings for intra-process isolation. *IEEE Transactions on Dependable and Secure Computing*, 20(4):3251–3268, 2023. doi:10.1109/TDSC.2022.3192524. [Cited on pages 36 and 141.]

[298] Hugo Lefeuvre, Vlad-Andrei Bădoiu, Yi Chien, Felipe Huici, Nathan Dautenhahn, and Pierre Olivier. Assessing the impact of interface vulnerabilities in compartmentalized software. In *Proceedings of the 30th Annual Network & Distributed System Security Symposium*, NDSS'23, 2023. doi:10.14722/ndss.2023.24117. [Cited on pages 24, 25, 79, 120, 122, 132, 136, 137, 138, 140, 143, 150, 153, 201, 204, 205, 208, and 211.]

[299] Hugo Lefeuvre, Vlad-Andrei Bădoiu, Alexander Jung, Stefan Lucian Teodorescu, Sebastian Rauch, Felipe Huici, Costin Raiciu, and Pierre Olivier. FlexOS: Towards flexible OS isolation. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'22, pages 467–482, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3503222.3507759. [Cited on pages 23, 24, 25, 46, 82, 91, 104, 115, 123, 126, 128, 129, 130, 131, 132, 134, 138, 139, 140, 153, 162, and 209.]

[300] Hugo Lefeuvre, Vlad-Andrei Bădoiu, Stefan Teodorescu, Pierre Olivier, Tiberiu Mosnoi, Răzvan Deaconescu, Felipe Huici, and Costin Raiciu. FlexOS: Making OS isolation flexible. In *Proceedings of the 18th Workshop on Hot Topics in Operating Systems*, HotOS'21. Association for Computing Machinery, June 2021. doi:10.1145/3458336.3465292. [Cited on pages 24, 25, 60, 82, 91, and 115.]

[301] Hugo Lefeuvre, David Chisnall, Marios Kogias, and Pierre Olivier. Towards (really) safe and fast confidential I/O. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, HotOS'23, pages 214–222, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3593856.3595913. [Cited on pages 24, 25, 160, and 199.]

[302] Hugo Lefeuvre, Gaulthier Gain, Vlad-Andrei Bădoiu, Daniel Dinca, Vlad-Radu Schiller, Costin Raiciu, Felipe Huici, and Pierre Olivier. Loupe: Driving the development of OS compatibility layers. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'24. Association for Computing Machinery, 2024. doi:10.1145/3617232.3624861. [Cited on pages 24, 25, and 162.]

[303] Hugo Lefeuvre, Gaulthier Gain, Daniel Dinca, Alexander Jung, Simon Kuenzer, Vlad-Andrei Badoiu, Razvan Deaconescu, Laurent Mathy, Costin Raiciu, Pierre Olivier, and Felipe Huici. Unikraft and the coming of age of unikernels. *USENIX ;login: Magazine*, 2021. [Cited on page 167.]

[304] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. doi:10.1007/978-3-642-17511-4_20. [Cited on pages 60 and 141.]

[305] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communications*, 14(7):1280–1297, 1996. doi:10.1109/49.536480. [Cited on pages 49 and 77.]

[306] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proceedings of the 6th USENIX Conference on Operating Systems Design and Implementation*, OSDI'04, pages 17–30. USENIX Association, 2004. [Cited on pages 62, 77, and 212.]

[307] Guanyu Li, Dong Du, and Yubin Xia. Iso-UniK: lightweight multi-process unikernel through memory protection keys. *Cybersecurity*, 3(1):11, May 2020. doi:10.1186/s42400-020-00051-9. [Cited on page 77.]

[308] Jialin Li, Samantha Miller, Danyang Zhuo, Ang Chen, Jon Howell, and Thomas Anderson. An incremental path towards a safer OS kernel. In *Proceedings of the 18th Workshop on Hot Topics in Operating Systems*, HotOS'21, pages 183–190, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3458336.3465277. [Cited on pages 54, 76, and 104.]

[309] Mengyuan Li, Yinqian Zhang, Zhiqiang Lin, and Yan Solihin. Exploiting unprotected I/O operations in AMD secure encrypted virtualization. In *Proceedings of the 28th USENIX Security Symposium*, USENIX Security'19, pages 1257–1272. USENIX Association, 2019. [Cited on pages 201 and 203.]

[310] Yanlin Li, Jonathan McCune, James Newsome, Adrian Perrig, Brandon Baker, and Will Drewry. MiniBox: A two-way sandbox for x86 native code. In *Proceedings of the 2014 USENIX annual technical conference*, ATC'14, pages 409–420. USENIX Association, 2014. [Cited on page 115.]

[311] J. Liedtke. On micro-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, SOSP'95, pages 237–250, New York, NY, USA, 1995. Association for Computing Machinery. doi:10.1145/224056.224075. [Cited on pages 18, 28, 30, 32, 120, 122, and 146.]

[312] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N. Asokan. PAC it up: Towards pointer integrity using ARM pointer

authentication. In *Proceedings of the 28th USENIX Security Symposium*, USENIX Security'19, pages 177–194, Santa Clara, CA, August 2019. USENIX Association. [Cited on pages 42, 87, and 89.]

[313] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, Christof Fetzer, and Peter Pietzuch. Glamdring: Automatic application partitioning for Intel SGX. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, ATC'17, pages 285–298, USA, 2017. USENIX Association. [Cited on pages 18, 42, 43, 120, 128, 134, 138, 157, and 204.]

[314] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *Proceedings of the 27th USENIX Security Symposium*, USENIX Security'18, pages 973–990, Baltimore, MD, August 2018. USENIX Association. [Cited on pages 50 and 76.]

[315] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. Light-Weight contexts: An OS abstraction for safety and performance. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'16, pages 49–64, Savannah, GA, November 2016. USENIX Association. [Cited on pages 18, 97, 108, 120, 134, 140, and 157.]

[316] Bin Liu, Bin Liu, Hongxia Jin, and Ramesh Govindan. Efficient privilege de-escalation for ad libraries in mobile apps. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys'15, pages 89–103, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2742647.2742668. [Cited on pages 29 and 133.]

[317] Peiming Liu, Gang Zhao, and Jeff Huang. Securing unsafe Rust programs with XRust. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE'20, pages 234–245, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3377811.3380325. [Cited on page 29.]

[318] Shen Liu, Gang Tan, and Trent Jaeger. PtrSplit: Supporting general pointers in automatic program partitioning. In *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security*, CCS'17, pages 2359–2371,

New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3133956.3134066. [Cited on pages 33, 77, 82, 91, 115, 128, 130, 132, 134, and 140.]

[319] Shen Liu, Dongrui Zeng, Yongzhe Huang, Frank Capobianco, Stephen McCamant, Trent Jaeger, and Gang Tan. Program-Mandering: Quantitative privilege separation. In *Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security*, CCS'19, pages 1023–1040, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3319535.3354218. [Cited on pages 33, 41, 42, 43, 128, 130, 131, 132, and 134.]

[320] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS'15, pages 1607–1619, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2810103.2813690. [Cited on pages 33, 35, 120, 128, 130, 132, and 134.]

[321] Kangjie Lu, Chengyu Song, Byoungyoung Lee, Simon P. Chung, Taesoo Kim, and Wenke Lee. ASLR-Guard: Stopping address space leakage for code reuse attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS'15, pages 280–291, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2810103.2813694. [Cited on pages 87 and 88.]

[322] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200. Association for Computing Machinery, 2005. doi:10.1145/1065010.1065034. [Cited on page 93.]

[323] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'13, pages 461–472. Association for Computing Machinery, 2013. doi:10.1145/2490301.2451167. [Cited on pages 35, 49, 77, and 167.]

[324] Toshiyuki Maeda and Akinori Yonezawa. Kernel mode Linux: Toward an operating system protected by a type theory. In Vijay A. Saraswat, editor, *Proceedings of the 2003 Annual Asian Computing Science Conference*, ASIAN'03, pages 3–17, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. doi:10.1007/978-3-540-40965-6_2. [Cited on page 49.]

[325] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My VM is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP'17, pages 218–233. Association for Computing Machinery, 2017. doi:10.1145/3132747.3132763. [Cited on pages 49 and 50.]

[326] Valentin J.M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 47(11):2312–2331, 2021. doi:10.1109/TSE.2019.2946563. [Cited on page 114.]

[327] Yandong Mao, Haogang Chen, Dong Zhou, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Software fault isolation with API integrity and multi-principal modules. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP'11, pages 115–128, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/2043556.2043568. [Cited on pages 35, 42, 82, 126, 134, 138, and 140.]

[328] Ilias Marinos, Robert N.M. Watson, and Mark Handley. Network stack specialization for performance. In *Proceedings of the ACM SIGCOMM 2014 Conference*, SIGCOMM'14, pages 175–186, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2619239.2626311. [Cited on page 52.]

[329] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. ClickOS and the art of network function virtualization. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'14, pages 459–473, Seattle, WA, April 2014. USENIX Association. [Cited on pages 49, 50, 52, and 167.]

[330] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. CCFI: Cryptographically enforced control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS'15,

pages 941–951, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2810103.2813676. [Cited on page 87.]

[331] M. Masmano, I. Ripoll, A. Crespo, and J. Real. TLSF: a new dynamic memory allocator for real-time systems. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems*, ECRTS'04, pages 79–88, 2004. doi:10.1109/EMRTS.2004.1311009. [Cited on page 74.]

[332] Steven McCanne and Van Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *Proceedings of the USENIX Winter 1993 Conference*, USENIX'93, page 2, USA, 1993. USENIX Association. [Cited on page 152.]

[333] Derrick McKee, Yianni Giannaris, Carolina Ortega Perez, Howard Shrobe, Mathias Payer, Hamed Okhravi, and Nathan Burow. Preventing kernel hacks with HAKC. In *Proceedings of the 29th Annual Network & Distributed System Security Symposium*, NDSS'22, 2022. doi:10.14722/ndss.2022.24026. [Cited on pages 35, 41, 134, and 141.]

[334] Samuel Mergendahl, Nathan Burow, and Hamed Okhravi. Cross-language attacks. In *Proceedings of the 29th Annual Network & Distributed System Security Symposium*, NDSS'22, 2022. doi:10.14722/ndss.2022.24078. [Cited on pages 29 and 120.]

[335] Dirk Merkel. Docker: lightweight Linux containers for consistent development and deployment. *Linux Journal*, 2014(239), 2014. https://dl.acm.org/doi/10.5555/2600239.2600241. [Cited on pages 28 and 122.]

[336] Zeyu Mi, Dingji Li, Zihan Yang, Xinran Wang, and Haibo Chen. Skybridge: Fast and secure inter-process communication for microkernels. In *Proceedings of the 14th European Conference on Computer Systems*, EuroSys'19, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3302424.3303946. [Cited on page 77.]

[337] Microsoft. Microsoft digital defense report 2022. https://query.prod.cms.rt.microsoft.com/cms/api/am/binary/RE5bUvv. Viewed: 07/11/23. [Cited on page 17.]

[338] Alyssa Milburn, Erik Van Der Kouwe, and Cristiano Giuffrida. Mitigating information leakage vulnerabilities with type-based data isolation. In *Proceedings*

of the 2022 IEEE Symposium on Security and Privacy, S&P'22, pages 1049–1065, 2022. doi:10.1109/SP46214.2022.9833675. [Cited on pages 18 and 120.]

[339] Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006. [Cited on pages 121 and 151.]

[340] MITRE, Common Weakness Enumeration. CWE 189: Numeric errors. https://cwe.mitre.org/data/definitions/189.html. Viewed: 24/07/23. [Cited on page 89.]

[341] Lucian Mogosanu, Ashay Rane, and Nathan Dautenhahn. MicroStache: A lightweight execution context for in-process safe region isolation. In Michael Bailey, Thorsten Holz, Manolis Stamatogiannakis, and Sotiris Ioannidis, editors, *Proceedings of the 21st International Symposium on Research in Attacks, Intrusions and Defenses*, RAID'18, pages 359–379, Cham, 2018. Springer International Publishing. doi:10.1007/978-3-030-00470-5_17. [Cited on pages 36, 141, and 144.]

[342] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. RockSalt: Better, faster, stronger SFI for the x86. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'12, pages 395–404, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2254064.2254111. [Cited on pages 36 and 141.]

[343] Myoung Jin Nam, Periklis Akritidis, and David J Greaves. FRAMER: A tagged-pointer capability system with memory safety applications. In *Proceedings of the 35th Annual Computer Security Applications Conference*, ACSAC'19, pages 612–626, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3359789.3359799. [Cited on pages 36 and 141.]

[344] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. Retrofitting fine grain isolation in the Firefox renderer. In *Proceedings of the 29th USENIX Security Symposium*, USENIX Security'20, pages 699–716. USENIX Association, August 2020. [Cited on pages 18, 19, 38, 40, 42, 53, 58, 65, 77, 82, 84, 87, 88, 89, 90, 91, 97, 98, 99, 111, 115, 120, 122, 125, 128, 129, 131, 132, 134, 136, 138, 139, 140, 147, 148, 201, and 209.]

[345] Shravan Narayan, Craig Disselkoen, Daniel Moghimi, Sunjay Cauligi, Evan Johnson, Zhao Gang, Anjo Vahldiek-Oberwagner, Ravi Sahita, Hovav Shacham, Dean

Tullsen, and Deian Stefan. Swivel: Hardening WebAssembly against Spectre. In *Proceedings of the 30th USENIX Security Symposium*, USENIX Security'21, pages 1433–1450. USENIX Association, August 2021. [Cited on pages 18, 30, and 120.]

[346] Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, Zhaofeng Li, Gerd Zellweger, and Anton Burtsev. RedLeaf: Isolation and communication in a safe operating system. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'20. USENIX Association, November 2020. [Cited on pages 18, 35, 77, 120, 134, and 137.]

[347] Vikram Narayanan, Yongzhe Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. Lightweight kernel isolation with virtualization and VM functions. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE'20, pages 157–171, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3381052.3381328. [Cited on pages 134 and 139.]

[348] Ruslan Nikolaev and Godmar Back. VirtuOS: An operating system with kernel virtualization. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, SOSP'13, pages 116–132, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2517349.2522719. [Cited on pages 62, 77, 134, 137, and 212.]

[349] Ruslan Nikolaev, Mincheol Sung, and Binoy Ravindran. LibrettOS: A dynamically adaptable multiserver-library OS. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE'20, pages 114–128, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3381052.3381316. [Cited on pages 62, 77, 126, 134, and 137.]

[350] Robert M. Norton. *Hardware Support for Compartmentalization*. PhD thesis, University of Cambridge Computer Laboratory, May 2016. doi:10.48456/tr-887. [Cited on page 120.]

[351] Pierre Olivier, Antonio Barbalace, and Binoy Ravindran. The case for intra-unikernel isolation. *Proceedings of the 10th Workshop on Systems for Post-Moore Architectures*, April 2020. [Cited on page 77.]

[352] Pierre Olivier, Daniel Chiba, Stefan Lankes, Changwoo Min, and Binoy Ravindran. A binary-compatible unikernel. In *Proceedings of the 15th ACM*

SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE'19, pages 59–73, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3313808.3313817. [Cited on pages 49, 77, 165, 167, 168, 169, 176, 177, and 197.]

[353] Pierre Olivier, Hugo Lefeuvre, Daniel Chiba, Stefan Lankes, Changwoo Min, and Binoy Ravindran. A syscall-level binary-compatible unikernel. *IEEE Transactions on Computers*, 71(9):2116–2127, 2022. doi:10.1109/TC.2021.3122896. [Cited on pages 25, 165, 167, 168, 197, and 198.]

[354] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. The spy in the sandbox: Practical cache attacks in JavaScript and their implications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS'15, pages 1406–1418, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2810103.2813708. [Cited on page 150.]

[355] Sebastian Österlund, Koen Koning, Pierre Olivier, Antonio Barbalace, Herbert Bos, and Cristiano Giuffrida. kMVX: Detecting kernel information leaks with multi-variant execution. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'19, pages 559–572. Association for Computing Machinery, 2019. doi:10.1145/3297858.3304054. [Cited on page 88.]

[356] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. Documenting and automating collateral evolutions in Linux device drivers. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems*, EuroSys'08, pages 247–260, New York, NY, USA, 2008. Association for Computing Machinery. doi:10.1145/1352592.1352618. [Cited on pages 55 and 57.]

[357] Shankara Pailoor, Xinyu Wang, Hovav Shacham, and Isil Dillig. Automated policy synthesis for system call sandboxing. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA), 2020. doi:10.1145/3428203. [Cited on page 168.]

[358] Soyeon Park, Sangho Lee, and Taesoo Kim. Memory protection keys: Facts, key extension perspectives, and discussions. *IEEE Security & Privacy*, 21(3):8–15, 2023. doi:10.1109/MSEC.2023.3250601. [Cited on page 150.]

[359] Soyeon Park, Sangho Lee, Wen Xu, HyunGon Moon, and Taesoo Kim. libmpk: Software abstraction for Intel Memory Protection Keys (Intel MPK). In *Proceedings of the 2019 USENIX Annual Technical Conference*, ATC'19, pages 241–254, Renton, WA, July 2019. USENIX Association. [Cited on pages 82, 91, 97, 115, 120, 134, 141, 144, and 145.]

[360] Gabriel Parmer and Richard West. Mutable protection domains: Towards a component-based system for dependable and predictable computing. In *Proceedings of the 28th IEEE International Real-Time Systems Symposium*, RTSS'07, pages 365–378, 2007. doi:10.1109/RTSS.2007.27. [Cited on pages 54, 123, 126, 128, 134, and 137.]

[361] Gabriel Parmer and Richard West. Mutable protection domains: Adapting system fault isolation for reliability and efficiency. *IEEE Transactions on Software Engineering*, 38(4):875–888, 2012. doi:10.1109/TSE.2011.61. [Cited on pages 123, 128, 134, and 137.]

[362] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972. doi:10.1145/361598.361623. [Cited on page 104.]

[363] Andrea Parri. hv_netvsc: Add validation for untrusted Hyper-V values. https://lkml.org/lkml/2020/9/16/380, 2022. Viewed: 30/01/23. [Cited on page 207.]

[364] Paul Pearce, Adrienne Porter Felt, Gabriel Nunez, and David Wagner. AdDroid: Privilege separation for applications and advertisers in Android. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, AsiaCCS'12, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2414456.2414498. [Cited on page 133.]

[365] D. Peng, C. Liu, T. Palit, P. Fonseca, A. Vahldiek-Oberwagner, and M. Vij. μSWITCH: Fast kernel context isolation with implicit context switches. In *Proceedings of the 2023 IEEE Symposium on Security and Privacy*, S&P'23, 2023. doi:10.1109/SP46215.2023.10179284. [Cited on pages 134 and 140.]

[366] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. AFLNET: A greybox fuzzer for network protocols. In *Proceedings of the 13th IEEE International Conference on Software Testing, Validation and Verification*, ICST'20, pages 460–465, 2020. doi:10.1109/ICST46399.2020.00062. [Cited on page 114.]

[367] Rishabh Poddar, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. SafeBricks: Shielding network functions in the cloud. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'18, pages 201–216. USENIX Association, 2018. [Cited on pages 206 and 208.]

[368] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. Rethinking the library OS from the top down. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 291–304, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/1950365.1950399. [Cited on pages 165 and 167.]

[369] Bobby Powers, John Vilk, and Emery D. Berger. Browsix: Bridging the gap between UNIX and the browser. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'17. Association for Computing Machinery, 2017. doi:10.1145/3093336.3037727. [Cited on pages 177 and 197.]

[370] Christian Priebe, Divya Muthukumaran, Joshua Lind, Huanzhou Zhu, Shujie Cui, Vasily A Sartakov, and Peter Pietzuch. SGX-LKL: Securing the host OS interface for trusted execution. *arXiv pre-print*, 2019. doi:10.48550/arXiv.1908.1114. [Cited on pages 84 and 115.]

[371] Sergej Proskurin, Marius Momeu, Seyedhamed Ghavamnia, Vasileios P. Kemerlis, and Michalis Polychronakis. xMP: Selective memory protection for kernel and user space. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy*, S&P'20, pages 563–577, 2020. doi:10.1109/SP40000.2020.00041. [Cited on pages 35 and 134.]

[372] Niels Provos. Improving host security with system call policies. In *Proceedings of the 12th USENIX Security Symposium*, USENIX Security'03, USA, 2003. USENIX Association. [Cited on page 140.]

[373] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *Proceedings of the 12th USENIX Security Symposium*, USENIX Security'03, USA, 2003. USENIX Association. [Cited on pages 18, 21, 28, 32, 38, 39, 120, 122, 123, 145, 147, and 152.]

[374] Martin Radev and Mathias Morbitzer. Exploiting interfaces of secure encrypted virtual machines. In *Proceedings of the 4th Reversing and Offensive-Oriented*

Trends Symposium, ROOTS'20, pages 1–12, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3433667.3433668. [Cited on pages 201 and 203.]

[375] FL. Rawson. Experience with the development of a microkernel-based, multi-server operating system. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems*, HotOS'97. IEEE, May 1997. doi:10.1109/HOTOS.1997.595173. [Cited on pages 165 and 167.]

[376] Ali Raza, Thomas Unger, Matthew Boyd, Eric B Munson, Parul Sohal, Ulrich Drepper, Richard Jones, Daniel Bristot De Oliveira, Larry Woodman, Renato Mancuso, Jonathan Appavoo, and Orran Krieger. Unikernel Linux (UKL). In *Proceedings of the 18th European Conference on Computer Systems*, EuroSys'23, pages 590–605, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3552326.3587458. [Cited on page 167.]

[377] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. Optimizing seed selection for fuzzing. In *Proceedings of the 23rd USENIX Security Symposium*, USENIX Security'14, pages 861–875, San Diego, CA, August 2014. USENIX Association. [Cited on pages 94 and 99.]

[378] Charles Reis, Alexander Moshchuk, and Nasko Oskov. Site isolation: Process separation for Web sites within the browser. In *Proceedings of the 28th USENIX Security Symposium*, USENIX Security'19, pages 1661–1678, Santa Clara, CA, August 2019. USENIX Association. [Cited on pages 18, 38, and 146.]

[379] Don Reisinger. Akamai Heartbleed patch not a fix after all. https://www.cnet.com/news/privacy/akamai-heartbleed-patch-not-a-fix-after-all/, 2014. Viewed: 23/12/22. [Cited on pages 88 and 108.]

[380] Kui Ren, Cong Wang, and Qian Wang. Security challenges for the public cloud. *IEEE Internet Computing*, 16(1):69–73, 2012. doi:10.1109/MIC.2012.14. [Cited on page 201.]

[381] E. Rescorla. The Transport Layer Security (TLS) protocol version 1.3. https://www.rfc-editor.org/rfc/rfc8446, 2018. Viewed: 28/01/23. [Cited on pages 201 and 205.]

[382] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and William S. Beebee. Enhancing server availability and security through failure-oblivious computing. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation*, OSDI'04. USENIX Association, December 2004. [Cited on page 169.]

[383] Elijah Rivera, Samuel Mergendahl, Howard Shrobe, Hamed Okhravi, and Nathan Burow. Keeping safe Rust safe with Galeed. In *Proceedings of the 37th Annual Computer Security Applications Conference*, ACSAC'21, pages 824–836, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3485832.3485903. [Cited on pages 29 and 120.]

[384] Nick Roessler, Lucas Atayde, Imani Palmer, Derrick McKee, Jai Pandey, Vasileios P. Kemerlis, Mathias Payer, Adam Bates, Jonathan M. Smith, Andre DeHon, and Nathan Dautenhahn. μSCOPE: A methodology for analyzing least-privilege compartmentalization in large software artifacts. In *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses*, RAID'21, pages 296–311, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3471621.3471839. [Cited on pages 18, 33, 41, 43, 120, 123, 128, 130, 131, 132, and 153.]

[385] Nick Roessler and André DeHon. SCALPEL: Exploring the limits of tag-enforced compartmentalization. *ACM Journal on Emerging Technologies in Computing Systems*, 18(1), sep 2021. doi:10.1145/3461673. [Cited on pages 128 and 153.]

[386] Nick Roessler and André DeHon. Protecting the stack with metadata policies and tagged hardware. In *Proceedings of the 2018 IEEE Symposium on Security and Privacy*, S&P'18, pages 478–495, 2018. doi:10.1109/SP.2018.00066. [Cited on pages 36 and 141.]

[387] J. M. Rushby. Design and verification of secure systems. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles*, SOSP'81, pages 12–21, New York, NY, USA, 1981. Association for Computing Machinery. doi:10.1145/800216.806586. [Cited on pages 36, 77, 120, 127, 141, 143, and 144.]

[388] J. M. Rushby. Proof of separability a verification technique for a class of security kernels. In Mariangiola Dezani-Ciancaglini and Ugo Montanari, editors, *Proceedings of the 1982 International Symposium on Programming*, pages 352–367, Berlin,

Heidelberg, 1982. Springer Berlin Heidelberg. doi:10.1007/3-540-11494-7_23. [Cited on page 120.]

[389] Rusty Russell. VirtIO: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Systems Review*, 42(5):95–103, 2008. doi:10.1145/1400097.1400108. [Cited on page 208.]

[390] Mark Russinovich, Edward Ashton, Christine Avanessians, Miguel Castro, Amaury Chamayou, Sylvan Clebsch, Manuel Costa, Cédric Fournet, Matthew Kerner, Sid Krishna, Julien Maffre, Thomas Moscibroda, Kartik Nayak, Olya Ohrimenko, Felix Schuster, Roy Schwartz, Alex Shamis, Olga Vrousgou, and Christoph M. Wintersteiger. CCF: A framework for building confidential verifiable replicated services. Technical report, Microsoft Research and Microsoft Azure, 2019. URL: https://github.com/microsoft/CCF/blob/main/CCF-TECHNICAL-REPORT.pdf. [Cited on pages 202, 204, and 205.]

[391] Mark Russinovich, Manuel Costa, Cédric Fournet, David Chisnall, Antoine Delignat-Lavaud, Sylvan Clebsch, Kapil Vaswani, and Vikas Bhatia. Toward confidential cloud computing. *Communications of the ACM*, 64(6):54–61, May 2021. doi:10.1145/3453930. [Cited on pages 28, 122, 199, 201, and 204.]

[392] J.H. Saltzer and M.D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975. doi:10.1109/PROC.1975.9939. [Cited on pages 17, 26, 82, 120, 121, 151, and 152.]

[393] Michael Sammler, Deepak Garg, Derek Dreyer, and Tadeusz Litak. The high-level benefits of low-level sandboxing. *Proceedings of the ACM on Programming Languages*, 4(POPL), dec 2019. doi:10.1145/3371100. [Cited on page 150.]

[394] Vasily A. Sartakov, Daniel O'Keeffe, David Eyers, Lluís Vilanova, and Peter Pietzuch. Spons & Shields: Practical isolation for trusted execution. In *Proceedings of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE'21, pages 186–200, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3453933.3454024. [Cited on page 209.]

[395] Vasily A. Sartakov, Lluís Vilanova, and Peter Pietzuch. CubicleOS: A library OS with software componentisation for practical isolation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'21, pages 546–558, New York, NY, USA,

2021. Association for Computing Machinery. doi:10.1145/3445814.3446731. [Cited on pages 18, 33, 35, 55, 65, 68, 73, 77, 82, 91, 97, 104, 108, 115, 120, 126, 128, 134, 138, and 157.]

[396] Hiroshi Sasaki, Miguel A. Arroyo, M. Tarek Ibn Ziad, Koustubha Bhat, Kanad Sinha, and Simha Sethumadhavan. Practical byte-granular memory blacklisting using Califorms. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO'19, pages 558–571, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3352460.3358299. [Cited on pages 36, 42, and 141.]

[397] Stefan Savage and Brian N. Bershad. Some issues in the design of an extensible operating system (panel statement). In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation*, OSDI'94. USENIX Association, 1994. [Cited on pages 21, 40, 41, 122, 140, and 150.]

[398] Roger R. Schell, Peter J. Downey, and Gerald J. Popek. Preliminary notes on the design of secure military computer systems. Technical report, 1973. Electronic Systems Division, Hanscom AFB, MA. [Cited on page 121.]

[399] David Schrammel, Samuel Weiser, Richard Sadek, and Stefan Mangard. Jenny: Securing syscalls for PKU-based memory isolation systems. In *Proceedings of the 31st USENIX Security Symposium*, USENIX Security'22. USENIX Association, August 2022. [Cited on pages 82, 97, 103, and 140.]

[400] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. Donky: Domain keys – efficient in-process isolation for RISC-V and x86. In *Proceedings of the 29th USENIX Security Symposium*, USENIX Security'20, pages 1677–1694. USENIX Association, August 2020. [Cited on pages 36, 49, 53, 60, 77, 82, 91, 115, 123, 127, 141, 143, and 144.]

[401] Sergej Schumilo, Cornelius Aschermann, Andrea Jemmett, Ali Abbasi, and Thorsten Holz. Nyx-Net: Network fuzzing with incremental snapshots. In *Proceedings of the 17th European Conference on Computer Systems*, EuroSys'22, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3492321.3519591. [Cited on pages 43, 114, and 147.]

[402] Jaebaek Seo, Daehyeok Kim, Donghyun Cho, Taesoo Kim, and Insik Shin. FlexDroid: Enforcing in-app privilege separation in Android. In *Proceedings of*

the 23rd Annual Network & Distributed System Security Symposium, NDSS'16, 2016. doi:10.14722/ndss.2016.23485. [Cited on page 133.]

[403] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Annual Technical Conference*, ATC'12, pages 309–318, Boston, MA, June 2012. USENIX Association. [Cited on pages 50 and 93.]

[404] Kosta Serebryany. Continuous fuzzing with libFuzzer and AddressSanitizer. In *Proceedings of the 2016 IEEE Cybersecurity Development (SecDev)*, pages 157–157. IEEE, 2016. doi:10.1109/SecDev.2016.043. [Cited on pages 92 and 114.]

[405] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS'04, pages 298–307, New York, NY, USA, 2004. Association for Computing Machinery. doi:10.1145/1030083.1030124. [Cited on page 30.]

[406] Shashi Shekhar, Michael Dietz, and Dan S. Wallach. AdSplit: Separating smartphone advertising from applications. In *Proceedings of the 21st USENIX Security Symposium*, USENIX Security'12, pages 553–567. USENIX Association, 2012. [Cited on page 133.]

[407] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. Occlum: Secure and efficient multitasking inside a single enclave of Intel SGX. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'20, pages 955–970, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3373376.3378469. [Cited on pages 202, 205, and 209.]

[408] Lei Shi, Yuming Wu, Yubin Xia, Nathan Dautenhahn, Haibo Chen, Binyu Zang, and Jinming Li. Deconstructing Xen. In *Proceedings of the 24th Annual Network & Distributed System Security Symposium*, NDSS'17, 2017. doi:10.14722/ndss.2017.23455. [Cited on pages 28, 122, and 134.]

[409] Rui Shu, Peipei Wang, Sigmund A Gorski III, Benjamin Andow, Adwait Nadkarni, Luke Deshotels, Jason Gionta, William Enck, and Xiaohui Gu. A study of security isolation techniques. *ACM Computing Surveys*, 49(3), oct 2016. doi:10.1145/2988545. [Cited on pages 28, 38, 122, 150, and 152.]

[410] Igor Smolyar, Muli Ben-Yehuda, and Dan Tsafrir. Securing self-virtualizing Ethernet devices. In *Proceedings of the 24th USENIX Security Symposium*, USENIX Security'15, pages 335–350, Washington, D.C., August 2015. USENIX Association. [Cited on page 212.]

[411] Livio Soares and Michael Stumm. FlexSC: Flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 33–46, USA, 2010. USENIX Association. [Cited on page 135.]

[412] Chengyu Song, Hyungon Moon, Monjur Alam, Insu Yun, Byoungyoung Lee, Taesoo Kim, Wenke Lee, and Yunheung Paek. HDFI: Hardware-assisted data-flow isolation. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy*, S&P'16, pages 1–17, 2016. doi:10.1109/SP.2016.9. [Cited on pages 36 and 141.]

[413] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. SoK: Sanitizing for security. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy*, S&P'19, pages 1275–1295. IEEE, 2019. doi:10.1109/SP.2019.00010. [Cited on page 141.]

[414] Evgeniy Stepanov and Konstantin Serebryany. MemorySanitizer: fast detector of uninitialized memory use in C++. In *Proceedings of the 2015 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO'15, pages 46–55. IEEE, 2015. doi:10.1109/CGO.2015.7054186. [Cited on pages 87 and 88.]

[415] Raoul Strackx, Pieter Agten, Niels Avonds, and Frank Piessens. Salus: Kernel support for secure process compartments. *EAI Endorsed Transactions on Security and Safety*, 2(3), 1 2015. doi:10.4108/sesa.2.3.e1. [Cited on pages 18, 120, 134, and 157.]

[416] Mengtao Sun and Gang Tan. NativeGuard: Protecting Android applications from third-party native libraries. In *Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless & Mobile Networks*, WiSec'14, pages 165–176, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2627393.2627396. [Cited on page 133.]

[417] Mincheol Sung, Pierre Olivier, Stefan Lankes, and Binoy Ravindran. Intra-unikernel isolation with Intel memory protection keys. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution*

Environments, VEE'20, pages 143–156, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3381052.3381326. [Cited on pages 29, 77, 82, and 120.]

[418] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007. [Cited on pages 92 and 114.]

[419] Michael M. Swift, Muthukaruppan Annamalai, Brian N. Bershad, and Henry M. Levy. Recovering device drivers. *ACM Transactions on Computer Systems*, 24(4):333–360, nov 2006. doi:10.1145/1189256.1189257. [Cited on page 137.]

[420] Michael M. Swift, Steven Martin, Henry M. Levy, and Susan J. Eggers. Nooks: An architecture for reliable device drivers. In *Proceedings of the 10th ACM SIGOPS European Workshop*, EW'02, pages 102–107, New York, NY, USA, 2002. Association for Computing Machinery. doi:10.1145/1133373.1133393. [Cited on pages 77 and 212.]

[421] L. Szekeres, M. Payer, T. Wei, and D. Song. SoK: Eternal war in memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, S&P'13, pages 48–62, 2013. doi:10.1109/SP.2013.13. [Cited on pages 61, 87, 89, and 150.]

[422] A. S. Tanenbaum, J. N. Herder, and H. Bos. Can we make operating systems reliable and secure? *Computer*, 39(5):44–51, 2006. doi:10.1109/MC.2006.156. [Cited on page 59.]

[423] Willy Tarreau. Nolibc: a minimal C-library replacement shipped with the kernel. https://lwn.net/Articles/920158/, January 2023. Viewed: 28/11/23. [Cited on pages 177 and 197.]

[424] Jörg Thalheim, Harshavardhan Unnibhavi, Christian Priebe, Pramod Bhatotia, and Peter Pietzuch. Rkt-Io: A direct I/O stack for shielded execution. In *Proceedings of the 16th European Conference on Computer Systems*, EuroSys'21, pages 490–506, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3447786.3456255. [Cited on pages 201, 202, 206, and 208.]

[425] The Linux Kernel Development Community. The kernel address sanitizer (KASAN). https://www.kernel.org/doc/html/v5.10/dev-tools/kasan.html, 2020. Viewed: 25/01/21. [Cited on page 50.]

[426] The Linux Kernel Development Community. dm-crypt: Linux kernel device-mapper crypto target. https://www.kernel.org/doc/html/latest/admin-guide/device-mapper/dm-crypt.html, 2023. Viewed: 28/01/23. [Cited on page 201.]

[427] The Linux Kernel Development Community. Filesystem-level encryption (fscrypt). https://www.kernel.org/doc/html/latest/filesystems/fscrypt.html, 2023. Viewed: 28/01/23. [Cited on page 201.]

[428] Deepu Thomas and Seth Juarez. Windows Subsystem for Linux (WSL) overview. https://learn.microsoft.com/en-us/archive/blogs/wsl/windows-subsystem-for-linux-overview, 2016. Viewed: 28/11/23. [Cited on pages 165, 167, 168, and 197.]

[429] Bohdan Trach, Alfred Krohmer, Franz Gregor, Sergei Arnautov, Pramod Bhatotia, and Christof Fetzer. ShieldBox: Secure middleboxes using shielded execution. In *Proceedings of the 2018 Symposium on SDN Research*, SOSR'18, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3185467.3185469. [Cited on pages 201, 202, 204, 206, and 208.]

[430] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A. Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E. Porter. Cooperation and security isolation of library OSes for multi-process applications. In *Proceedings of the 9th European Conference on Computer Systems*, EuroSys'14. Association for Computing Machinery, 2014. doi:10.1145/2592798.2592812. [Cited on pages 165, 167, 168, 177, and 197.]

[431] Chia-Che Tsai, Bhushan Jain, Nafees Ahmed Abdul, and Donald E. Porter. A study of modern Linux API usage and compatibility: What to support when you're supporting. In *Proceedings of the 11th European Conference on Computer Systems*, EuroSys'16, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2901318.2901341. [Cited on pages 165, 166, 167, 168, 180, 181, and 197.]

[432] Chia-Che Tsai, Donald E. Porter, and Mona Vij. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *Proceedings of the 2017 USENIX Annual Technical Conference*, ATC'17. USENIX Association, 2017. [Cited on pages 165, 167, 168, and 197.]

[433] UK National Cyber Security Centre. Annual review 2022. https://www.ncsc.gov.uk/files/NCSC-Annual-Review-2022.pdf. Viewed: 07/11/23. [Cited on page 17.]

[434] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, efficient in-process isolation with protection keys (MPK). In *Proceedings of the 28th USENIX Security Symposium*, USENIX Security'19, pages 1221–1238, Santa Clara, CA, August 2019. USENIX Association. [Cited on pages 18, 30, 35, 53, 61, 77, 82, 91, 97, 108, 115, 120, 122, 127, 134, 139, 140, 142, and 157.]

[435] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Conference on Security Symposium*, USENIX Security'18, pages 991–1008, USA, 2018. USENIX Association. [Cited on page 203.]

[436] Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio D. Garcia, and Frank Piessens. A tale of two worlds: Assessing the vulnerability of enclave shielding runtimes. In *Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security*, CCS'19, pages 1741–1758, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3319535.3363206. [Cited on pages 40, 82, 84, 113, 115, 150, 201, and 204.]

[437] Maarten Van Steen and Andrew S Tanenbaum. *Distributed systems*. Maarten van Steen Leiden, The Netherlands, 3 edition, 2020. [Cited on page 137.]

[438] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, and Jonathan M. Smith. BreakApp: Automated, flexible application compartmentalization. In *Proceedings of the 25th Annual Network & Distributed System Security Symposium*, NDSS'18, 2018. doi:10.14722/ndss.2018.23131. [Cited on pages 18, 33, 35, 120, 122, 128, 131, and 132.]

[439] Anthony Velte and Toby Velte. *Microsoft virtualization with Hyper-V*. McGraw-Hill, Inc., 2009. [Cited on page 208.]

[440] Lluís Vilanova, Muli Ben-Yehuda, Nacho Navarro, Yoav Etsion, and Mateo Valero. CODOMs: Protecting software with code-centric memory domains. In

Proceeding of the 41st Annual International Symposium on Computer Architecuture, ISCA'14, pages 469–480, Minneapolis, Minnesota, USA, 2014. IEEE Press. doi:10.1145/2678373.2665741. [Cited on pages 36 and 141.]

[441] Laurent Vivier. Virtio-net failover: An introduction. https://www.redhat.com/en/blog/virtio-net-failover-introduction, 2022. Viewed: 30/01/23. [Cited on page 210.]

[442] Alexios Voulimeneas, Jonas Vinck, Ruben Mechelinck, and Stijn Volckaert. You shall not (by)pass! Practical, secure, and fast PKU-based sandboxing. In *Proceedings of the 17th European Conference on Computer Systems*, EuroSys'22, pages 266–282. Association for Computing Machinery, 2022. doi:10.1145/3492321.3519560. [Cited on pages 140 and 150.]

[443] Dmitry Vyukov. Syzkaller: adventures in continuous coverage-guided kernel fuzzing. BlueHat IL, https://msrnd-cdn-stor.azureedge.net/bluehat/bluehatil/2019/assets/doc/syzkaller%20Adventures%20in%20Continuous%20Coverage-guided%20Kernel%20Fuzzing.pdf, 2020. Viewed: 23/12/22. [Cited on page 113.]

[444] D. Wagner and R. Dean. Intrusion detection via static analysis. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, S&P'01, 2000. doi:10.1109/SECPRI.2001.924296. [Cited on page 168.]

[445] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, SOSP'93, pages 203–216, New York, NY, USA, 1993. Association for Computing Machinery. doi:10.1145/168619.168635. [Cited on pages 36, 120, and 141.]

[446] K. G. Walter, S. I. Schaen, W. F. Ogden, W. C. Rounds, D. G. Shumway, D. D. Schaeffer, K. J. Biba, F. T. Bradshaw, S. R. Ames, and J. M. Gilligan. Structured specification of a security kernel. In *Proceedings of the International Conference on Reliable Software*, ICRS'75, pages 285–293, New York, NY, USA, 1975. Association for Computing Machinery. doi:10.1145/800027.808450. [Cited on page 121.]

[447] Helen J. Wang, Chris Grier, Alexander Moshchuk, Samuel T. King, Piali Choudhury, and Herman Venter. The multi-principal OS construction of the Gazelle Web browser. In *Proceedings of the 18th USENIX Security Symposium*, USENIX Security'09. USENIX Association, 2009. [Cited on pages 18, 38, and 146.]

[448] Jason Wang and Ariel Adam. Hardening Virtio for emerging security use cases. https://www.redhat.com/en/blog/hardening-virtio-emerging-security-usecases, 2022. Viewed: 30/01/23. [Cited on page 207.]

[449] Jun Wang, Xi Xiong, and Peng Liu. Between mutual trust and mutual distrust: Practical fine-grained privilege separation in multithreaded applications. In *Proceedings of the 2015 USENIX Annual Technical Conference*, ATC'15, pages 361–373, Santa Clara, CA, July 2015. USENIX Association. [Cited on pages 134 and 140.]

[450] Yan Wang, Chao Zhang, Zixuan Zhao, Bolun Zhang, Xiaorui Gong, and Wei Zou. MAZE: Towards automated heap Feng Shui. In *Proceedings of the 30th USENIX Security Symposium*, USENIX Security'21, pages 1647–1664. USENIX Association, August 2021. [Cited on page 106.]

[451] Nicholas Wanninger, Joshua Bowden, Kirtankumar Shetty, and Kyle Hale. Isolating functions at the hardware limit with Virtines. In *Proceedings of the 17th European Conference on Computer Systems*, EuroSys'22. Association for Computing Machinery, 2022. doi:10.1145/3492321.3519553. [Cited on pages 82, 131, and 134.]

[452] Robert N. M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. Capsicum: Practical capabilities for UNIX. In *Proceedings of the 19th USENIX Security Symposium*, USENIX Security'10, pages 29–45, USA, 2010. USENIX Association. [Cited on page 140.]

[453] Robert NM Watson, Peter G Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, Graeme Barnes, David Chisnall, Jessica Clarke, Brooks Davis, Lee Eisen, Nathaniel Wesley Filardo, Franz A. Fuchs, Richard Grisenthwaite, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, Simon W Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alex Richardson, Peter Rugg, Peter Sewell, Stacey Son, and Hongyan Xia. Capability hardware enhanced RISC instructions: CHERI instruction-set architecture (Version 9). Technical Report UCAM-CL-TR-987, University of Cambridge, 2023. doi:10.48456/tr-987. [Cited on pages 58 and 64.]

[454] Robert N.M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and

Munraj Vadera. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, S&P'15, pages 20–37. IEEE, 2015. doi:10.1109/SP.2015.9. [Cited on pages 36, 42, 49, 55, 63, 82, 84, 87, 88, 89, 115, 125, 135, 141, 143, and 209.]

[455] Samuel Weiser, Mario Werner, Ferdinand Brasser, Maja Malenko, Stefan Mangard, and Ahmad-Reza Sadeghi. TIMBER-V: Tag-isolated memory bringing fine-grained enclaves to RISC-V. In *Proceedings of the 26th Annual Network & Distributed System Security Symposium*, NDSS'19, 2019. doi:10.14722/ndss.2019.23068. [Cited on pages 36 and 141.]

[456] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and performance in the Denali isolation kernel. *ACM SIGOPS Operating Systems Review*, 36(SI):195–209, December 2003. doi:10.1145/844128.844147. [Cited on page 120.]

[457] Luca Wilke, Jan Wichelmann, Mathias Morbitzer, and Thomas Eisenbarth. Sevurity: No security without integrity : Breaking integrity-free memory encryption with minimal assumptions. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy*, S&P'20, pages 1483–1496, 2020. doi:10.1109/SP40000.2020.00080. [Cited on pages 126, 201, and 203.]

[458] Paul R. Wilson. Pointer swizzling at page fault time: Efficiently supporting huge address spaces on standard hardware. *SIGARCH Comput. Archit. News*, 19(4):6–13, 1991. doi:10.1145/122576.122577. [Cited on page 88.]

[459] Emmett Witchel, Josh Cates, and Krste Asanović. Mondrian memory protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, pages 304–316, New York, NY, USA, 2002. Association for Computing Machinery. doi:10.1145/605397.605429. [Cited on page 141.]

[460] Jonathan Woodruff, Robert NM Watson, David Chisnall, Simon W Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G Neumann, Robert Norton, and Michael Roe. The CHERI capability model: Revisiting RISC in an age of risk. In *Proceedings of the 41st International Symposium on Computer Architecture*, ISCA'14, pages 457–468. IEEE Press, 2014. doi:10.1145/2678373.2665740. [Cited on page 143.]

[461] Cory Wright. djbdns: more than just a mouthful of consonants. *Linux Journal*, 2008(173), 2008. https://dl.acm.org/doi/abs/10.5555/1412202.1412205. [Cited on page 146.]

[462] Yongzheng Wu, Sai Sathyanarayan, Roland H. C. Yap, and Zhenkai Liang. Codejail: Application-transparent isolation of libraries with tight program interactions. In Sara Foresti, Moti Yung, and Fabio Martinelli, editors, *Proceedings of the 17th European Symposium on Research in Computer Security*, ESORICS'12, pages 859–876, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. doi:10.1007/978-3-642-33167-1_49. [Cited on pages 82, 91, 97, and 115.]

[463] Yongzheng Wu, Jun Sun, Yang Liu, and Jin Song Dong. Automatically partition software into least privilege components using dynamic data dependency analysis. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ASE'03, pages 323–333, 2013. doi:10.1109/ASE.2013.6693091. [Cited on pages 33, 43, 120, 128, and 130.]

[464] Jianhao Xu, Kangjie Lu, Zhengjie Du, Zhu Ding, Linke Li, Qiushi Wu, Mathias Payer, and Bing Mao. Silent bugs matter: A study of compiler-introduced security bugs. In *Proceedings of the 32nd USENIX Security Symposium*, USENIX Security'23. USENIX Association, 2023. [Cited on page 150.]

[465] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, S&P'15, pages 640–656, 2015. doi:10.1109/SP.2015.45. [Cited on page 203.]

[466] Carter Yagemann, Mohammad A. Noureddine, Wajih Ul Hassan, Simon Chung, Adam Bates, and Wenke Lee. Validating the integrity of audit logs against execution repartitioning attacks. In *Proceedings of the 28th ACM SIGSAC Conference on Computer and Communications Security*, CCS'21, pages 3337–3351, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3460120.3484551. [Cited on page 138.]

[467] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the 2009 IEEE Symposium on Security and Privacy*, S&P'09, pages 79–93, 2009. doi:10.1109/SP.2009.25. [Cited on pages 36 and 141.]

[468] M. Young, A. Tevanian, R. Rashid, D. Golub, and J. Eppinger. The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, SOSP'87, pages 63–76, New York, NY, USA, 1987. Association for Computing Machinery. doi:10.1145/41457.37507. [Cited on pages 30, 32, 120, and 146.]

[469] Ziqi Yuan, Siyu Hong, Rui Chang, Yajin Zhou, Wenbo Shen, and Kui Ren. VDom: Fast and unlimited virtual domains on multiple architectures. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS'23, pages 905–919, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3575693.3575735. [Cited on pages 36, 120, 141, and 144.]

[470] Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik. APISan: Sanitizing API usages through semantic Cross-Checking. In *Proceedings of the 25th USENIX Security Symposium*, USENIX Security'16, pages 363–378, Austin, TX, August 2016. USENIX Association. [Cited on pages 87, 89, 90, 114, and 130.]

[471] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Secure program partitioning. *ACM Transactions on Computer Systems*, 20(3):283–328, aug 2002. doi:10.1145/566340.566343. [Cited on pages 128, 130, 131, 132, and 134.]

[472] Bin Zeng, Gang Tan, and Greg Morrisett. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS'11, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/2046707.2046713. [Cited on pages 36 and 141.]

[473] Tao Zhang, Boris Pismenny, Donald E. Porter, Dan Tsafrir, and Aviad Zuck. Rowhammering storage devices. In *Proceedings of the 13th ACM Workshop on Hot Topics in Storage and File Systems*, HotStorage'21, pages 77–85, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3465332.3470871. [Cited on page 212.]

[474] Xiao Zhang, Amit Ahlawat, and Wenliang Du. AFrame: Isolating advertisements from mobile applications in Android. In *Proceedings of the 29th Annual Computer Security Applications Conference*, ACSAC'13, New York, NY, USA,

2013. Association for Computing Machinery. doi:10.1145/2523649.2523652. [Cited on page 133.]

[475] Yiming Zhang, Jon Crowcroft, Dongsheng Li, Chengfen Zhang, Huiba Li, Yaozheng Wang, Kai Yu, Yongqiang Xiong, and Guihai Chen. KylinX: A dynamic library operating system for simplified and efficient cloud virtualization. In *Proceedings of the 2018 USENIX Annual Technical Conference*, ATC'18, pages 173–186. USENIX Association, July 2018. [Cited on page 62.]

[476] Zenong Zhang, George Klees, Eric Wang, Michael Hicks, and Shiyi Wei. Registered report: Fuzzing configurations of program options. In *Proceedings of the 1st International Fuzzing Workshop*, FUZZING'22, 2022. doi:10.14722/fuzzing.2022.23008. [Cited on pages 94 and 99.]

[477] Zhe Zhou, Zhou Li, and Kehuan Zhang. All your VMs are disconnected: Attacking hardware virtualized network. In *Proceedings of the 7th ACM Conference on Data and Application Security and Privacy*, CODASPY'17, pages 249–260, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3029806.3029810. [Cited on page 212.]

[478] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. Small world with high risks: A study of security threats in the npm ecosystem. In *Proceedings of the 28th USENIX Security Symposium*, USENIX Security'19, pages 995–1010, Santa Clara, CA, August 2019. USENIX Association. [Cited on page 29.]