Thèse n° 9975

Secure Interface Design Leveraging Hardware/ Software Support

Présentée le 7 juin 2024

Faculté informatique et communications
Laboratoire HexHive
Programme doctoral en informatique et communications

pour l'obtention du grade de Docteur ès Sciences

par

Atri BHATTACHARYYA

Acceptée sur proposition du jury

Prof. C. González Troncoso, présidente du jury
Prof. M. J. Payer, Prof. B. Falsafi, directeurs de thèse
Dr A. Kurmus, rapporteur
Dr A. L. Vahldiek-oberwagner, rapporteur
Prof. S. Kashyap, rapporteur

École
polytechnique
fédérale
de Lausanne

2024

Let us think the unthinkable,
Let us do the undoable,
Let us prepare to grapple with the ineffable itself,
And see if we may not eff it after all.

Douglas Adams

To those who inspired and encouraged my pursuit of knowledge

Acknowledgments

" It's a dangerous business, Frodo, going out your door. You step onto the road, and if you don't keep your feet, there's no knowing where you might be swept off to. ", Bilbo Baggins often said to his young nephew Frodo Baggins. When I left India to start my Master's study at EPFL in 2016, little did I know of the journey that lay ahead. Looking back at the path culminating in the attainment of the PhD, I must thank the many people who accompanied me through my wanderings, sweeping me off on unknown paths and shining light in times of darkness.

Every journey requires guidance, and I would like to first thank my academic advisors, Prof. Mathias Payer and Prof. Babak Falsafi, who led me through my PhD. Mathias has been a source of inspiration throughout the years, with his belief that all systems are definitely and undeniably "broken". Mathias was critical to my successes during the PhD, pushing me to try every new avenue, teaching me to believe in my ideas, and providing the ever present support required to realize ideas. My thesis is the culmination of a long series of ideas and "weekend projects" of increasing depth and clarity. Many of the ideas comprising this thesis result from ideas refined and polished with Mathias' invaluable insights. Mathias' humanity was also key to surviving the unprecedented COVID-19 crisis marking a major fraction of the PhD. I am also deeply indebted to Babak, who hosted me in PARSA during my masters studies in addition to being my PhD co-advisor. Babak is an exceptional leader, and has the ability to push research projects to look beyond the superficial symptoms ailing computer systems, and find the right questions to ask. Babak's belief and insistence on pushing for excellence has been instrumental in deciding my research topics.

I would also like to thank the members of my thesis committee, Prof. Carmela Troncoso, Prof. Sanidhya Kashyap, Dr. Anil Kurmus, and Dr. Anjo Vahldiek-Oberwagner. I greatly appreciate their detailed and constructive feedback on my thesis draft, and for providing a fresh perspective on my research allowing me to iron the final wrinkles in my research.

I have also profited from a set of amazing collaborators helping me realize my research ideas. Dr. Anil Kurmus' sabbatical at EPFL was an incredible opportunity to explore the finer details of microarchitectural side-channel attacks. I have also been fortunate to closely collaborate with other PhD students, primarily Siddharth Gupta, Florian Hofhammer, Andres Sanchez, Yuanlong Li, Uros Tesic and Lana Josipovic. Without our time spent working through tough problems, and working late to meet ambitious deadlines, the PhD journey would have been particularly lacking luster. I would like to particularly thank Siddharth and Florian for teaching me to structure papers into logical sequences of details from an unmanageable heap of ideas and data. I must also thank

my student collaborators, Andrej Gorjan, Lucas Lopez Cendes, and Solène Husseini, who enabled me to explore the ideas which I never had time to do justice to. Alongside the students I had the pleasure to help during teaching assistant duties, they helped me realize my love for teaching, for sharing knowledge, and for learning together.

My love for teaching stems from the efforts of many amazing academics who have injected me with their passion for knowledge. I fondly remember the passionate high school lessons by Mr. Dey, Mr. Sur, Mr. Royan Mrs. Basu, and Mrs. Chakrabarthy. They helped develop an adoration for maths, the sciences, and the arts. My bachelors would have been far less educative without the efforts of Prof. Mainak Chowdhury, Prof. Aloke Dutta, Prof. Achla Raina, and Prof. Suchitra Mathur. The efforts of Prof. Chowdhury and Prof. Dutta formed the basis for my future in low-level topics in computer science. Prof. Raina and Prof. Mathur fascinated me with the use of science to understand society and particularly its language. I am also thankful to my teachers at EPFL for the knowledge imparted during numerous courses through my Masters and PhD studies here. I have found the advice "prepare to throw one away" imparted during the Principles of Computer Systems course invaluable for my PhD research.

I would like to specially extend thanks to Natascha Fontana for seamlessly managing the myriad administrative tasks and for dealing with EPFL's bureaucracy. She managed to wrap her head around the interminable list of funding sources supporting my PhD, and always managed to simplify requirements from our side. Natascha's efforts made dealings with EPFL administration, from organizing and cancelling visiting talks, reimbursing expenses, renewing contracts, hiring interns, and procuring hardware trivial.

No person walks through a PhD alone, and I am deeply grateful for my many colleagues who walked beside me. I would like to particularly thank the members of the Hive and PARSA, my two primary academic families during my time at EPFL. I am indebted to Florian Hofhammer, who has evolved from being a student I supervised for a Master's thesis to a close collaborator to eventually becoming a constant fixture in innumerable rambling discussions over coffee, rants against every conceivable inconvenience, and being a rubber ducky for many random, ambitious or otherwise zany ideas. Without your engineering skills, SecureCells may never have been a paper, and I might not have finished a thesis. Next, I have to thank Nicolas Badoux who has been an unending source of generosity, joy, cakes and chocolate across the years. I must also thank Luca di Bartolomeo for teaching me to unabashedly ask "Why?", and for teaching me to not feel guilty about being constructively competitive. My gratitude also goes out to the other members of the Hive. Ahmad Hazimeh has been my colleague in the lab for the longest, and has shared my earliest days here. Andres Sanchez has been a close collaborator across numerous projects, and will be remembered for his terrible puns. The postdocs, Flavio Toffalini and Marcel Busch brought a fresh dose of life into the lab. I feel like Flavio always has an ear to listen to whatever I need to get off my mind. I look forward to more afternoons playing badminton with the Hive's permanent visitor, Qiang Liu. Marcel has been instrumental in creating many indelible memories, specially on Fridays. I have also enjoyed spending time with the many other members of the lab, Antony Vennard, Lucio Romerio, Jelena Jankovic, Uros Tesic, Gwangmu Lee, Daniele Antonioli, Chibin

Zhang, Jean Michel, Han Zheng, Zhiyao Feng, and Phillip Mao. On the PARSA side, I must thank Siddharth Gupta with whom I shared many experiences including working on multiple projects and having a major skiing accident. Sid also taught me how to write good papers. Ognjen Glamocanin has been a constant presence throughout the years, and I look forward to sharing more burgers with you. I must thank the remaining members, Alex Daglis, Mario Drumond, Arash Pourhabibi, Mark Sutherland, Javier Picorel, Dimitrii Ustiugov, Yuanlong Li, Shanqing Lin, Simla Harma, Yunho Oh, Rafael Pizzaro for welcoming me into the lab across many years. I was also lucky to have an amazing cohort of other PhD peers. Mahyar Emami was present at my first PC build, many intermediate Kashmiri lunches, and will likely help me with my next PC build. I will remember Sahand Kashani-Akhavan, the FPGA wizard, whose humor, humility and technical knowledge always left me astounded. I must thank the many others with whom I spent innumerable light moments in the fellowship room, Athanasios Xygkis, Xinrui Jia, Zeinab Shmeis, and Khashayar Barooti. Finally, I must acknowledge the remaining systems PhDs who contributed to making my PhD life amazing, Rishabh Iyer, David Aksun, Marios Kogias, Adrien Ghosn, Neelu Kalani, Stefan Nikolic, Matthaios Olma, Stella Giannakopoulou, Lana Josipovic, and Andrea Guerrieri.

Next, I want to thank the entire Zenith family, which was a crucial part of my Master's and PhD life, and helped support me particularly through COVID-19 times. The years in Marcelin were my best in Switzerland. The board game group comprising Kirtan Padh, Kenneth Joseph Paul, Christina Grimm, Asli Yörüsün, Soner Serbest, and (more recently) Saskia Thomi have contributed to many periods of fun and intense competition. I am also grateful for the many moments shared with Yassir el Maaroufi, Rahul Gupta, Rasool Ahmad, Sharbatanu Chatterjee, Shankha Nag, and Debdatta Ray. I would also like to thank Nina and Daniel Brissot, who welcomed me to Switzerland with open arms.

Finally, and most importantly, I must thank my family who have steadfastly been the base on which each of my achievements is based. This journey has been as much theirs as mine. I must thank my parents, who taught me to pursue my interests and supported me through thick and thin. My mother has inspired me to teach and to excel, teaching me how to find joy and happiness in every aspect of life. My father taught me to care, for everyone and anyone. I have rarely found people as compassionate as my father. I must also thank my aunt (Mashimoni) and uncle (Meshoi), who have cared for me like a son. I cannot express my gratitude for my grandmother (Dida), grandfather (Dadabhai), and grand-uncle (Chinida) who catalyzed my academic ambitions. Finally, I must acknowledge my hero, my elder brother, has enabled each of my successes and has been a key ingredient to my efforts through more than 30 years. There are no words to describe how he has helped me navigate each phase of my life. I doubt I would have progressed far beyond kindergarten without his active help. I must thank the remaining two members of "Chaarmurti", my cousins. They have always been a source of happiness and relaxation. Neel, Rishi, Dada, I look forward to more great times together.

I thank the various sources of funding supporting this thesis. My PhD research has been partially supported by EPFL, the 2020 IBM Research Fellowship, the European Research Council (grant agreement No. 850868), DARPA (HR001119S0089-AMP-FP-034), ONR (award N00014-18-1-2674 and 13000660-052), the 2023 Qualcomm Innovation Fellowship, Fondation Botnar, and a gift from Intel Corporation.

Lausanne, 2nd April, 2024 Atri Bhattacharyya

Abstract

Computer systems rely heavily on abstraction to manage the exponential growth of complexity across hardware and software. Due to practical considerations of compatibility between components of these complex systems across generations, developers have favoured stable interfaces at crucial boundaries such as between hardware and software, or between the kernel and userspace. While these interfaces have persisted across more than 20 years, the modern computing environment has evolved significantly in terms of security and performance. Our increasingly connected systems share code components of widely varying provenance and legacy interfaces are unable to counter modern threats while maintaining strict performance objectives. Computing requires new interfaces with stronger security guarantees which can also support high performance applications. First, the kernel-user interface remains one of the primary vectors for inter-application attacks as compromising the kernel gives an attacker total control over the system's resources and to other applications on the same system. Second, the virtual memory interface has newly emerged as another crucial interface enabling attackers to remotely compromise systems as applications increasingly execute third-party code, for example JavaScript scripts downloaded from the internet. In this thesis, therefore, we investigate these two key interfaces to improve their security and performance limits.

The kernel-user system call interface suffers from double-fetch bugs for passed-by-reference arguments stored in user memory. Double fetches allow malicious users to compromise the isolation guaranteed at the kernel-user interface to illegally access memory, cause kernel crashes, or to escalate their privileges. The modern multi-user, multiprocessing environment allows the user to change the arguments read by the kernel by modifying the contents of memory from a concurrent thread. Traditional testing techniques cannot eliminate all double-fetch bugs due to the complexity and configurability of the kernel. The extensibility of the kernel further exacerbates the challenge as third-party modules loaded by the kernel may further introduce double-fetches. We present Midas, a systematic mitigation for kernel double-fetches which leverages the kernel's interface to read user memory to guarantee that every kernel read of a user object during a system call will return the same value. Midas's guarantee makes an implicit assumption by kernel developers explicit, protecting the kernel against a class of bugs while incurring merely 3.4% overhead on diverse workloads across the NPB and PTS benchmark suites.

Whereas modern systems software runs code from a plethora of sources with varying degrees of trust, the traditional virtual memory abstraction lacks support for isolating untrusted parts of an application within the same virtual address space. Since all code running within a process execute at the same trust level, buggy or malicious third-party code can compromise the process by directly leaking or modifying memory used by other components of the application. We must redesign the virtual memory interface to allow applications to be compartmentalized, essentially implementing the principle of least privilege by isolating untrusted parts of the application within compartments with limited access to the application's resources. We present SecureCells, a novel architectural interface for intra-address space compartmentalization. SecureCells enables applications to define hardware-enforced memory views for application compartments with accelerated userspace instructions for inter-compartment calls. In microbenchmarks, SecureCells enables a 5-stage in-order core to switch compartments in only 8 cycles reducing the cost of transitions by an order of magnitude compared to the state of the art. We also build a full-system prototype of SecureCells, based on the RISC-V RocketChip core running the seL4 kernel to evaluate userspace benchmarks.

This thesis also presents the first systematization of knowledge for compartmentalization mechanisms, evaluating both qualitative and quantitative properties. We describe relevant security and performance properties for practical compartmentalization, and show how well each mechanism provides each property. A comprehensive review of compartmentalization techniques aims to enable computer systems developers to define a secure, performant and usable interface to support widespread compartmentalization of applications in the future. Our systematization exposes common shortcomings of these mechanisms, pointing future research efforts to opportunities for more comprehensive compartmentalization support.

This thesis posits that legacy interfaces between components of modern computing systems inhibits their security guarantees, and explores issues at two major interfaces. We show that principled redesign of interfaces enables the implementation of more secure systems while supporting high-performance application needs, with the design and implementation of Midas and SecureCells to tackle challenges at the kernel-user and intra-process interfaces respectively.

Résumé

Les systèmes informatiques font un lourd usage d'abstraction pour répondre à la croissance exponentielle de la complexité du matériel et du logiciel. A cause des considérations pour maintenir une compatibilité entre des éléments de différentes générations de ces systèmes complexes, les développeurs ont favorisé des interfaces stables aux limites critiques tels que celles entre le matériel et le logiciel ou entre les espaces utilisateurs et noyaux. Alors que ces interfaces perdurent depuis plus de 20 ans, l'environnement informatique moderne a évolué significativement en termes de sécurité et de performance. Ces systèmes sont de plus en plus connectés et partagent des composants de provenance très diverses. Ces anciennes interfaces ne sont pas à même de contrer les menaces modernes tout en maintenant des objectifs de performances stricts. L'informatique nécessite de nouvelles interfaces garantissant une plus grande sécurité tout en gardant possible des applications à haute performance. L'interface utilisateur-noyau reste un des vecteurs principaux d'attaques entre applications car la compromission du noyau permet à l'attaquant de contrôler les ressources du système ainsi que les autres applications installées. Au vue de l'augmentation de code tiers exécuté par les applications, par exemple des scripts Javascript téléchargé depuis l'internet, l'interface de la mémoire virtuelle est en train d'émerger comme une autre interface critique pouvant offrir à des attaquants l'accès au système. Dans cette thèse, nous investiguons donc ces deux interfaces critiques afin d'améliorer leurs limites de performances et de sécurité.

L'interface d'appel noyau-utilisateur souffre de bogues de double récupération pour les arguments passés par référence stockés dans la mémoire utilisateur. Les double récupérations permettent à un utilisateur malveillant de compromettre l'isolation garantie par l'interface noyau-utilisateur pour accéder illégalement à la mémoire, provoquant des crashs du noyau, ou permettant l'escalade de leurs privilèges. L'environnement moderne multi-utilisateur et multi-processus permet à l'utilisateur de modifier les arguments lus par le noyau à différents moments en modifiant le contenu de la mémoire à partir d'un fil concurrent. La complexité du noyau empêche les développeurs de trouver et de corriger tous ses bogues. L'extensibilité du noyau aggrave encore le défi, car des modules tiers chargés par le noyau peuvent également introduire des double récupérations. Nous présentons Midas, une prévention systématique des double récupérations du noyau en exploitant son interface pour accéder à la mémoire utilisateur afin de garantir que chaque lecture demandée par le noyau d'un objet utilisateur lors d'un appel système renverra la même valeur. La garantie de Midas rend explicite une hypothèse implicite des développeurs du noyau, protégeant le noyau contre une classe de bogues tout en entraînant un coût de seulement 3,4% sur les charges de travail diverses des suites de benchmarks NPB et PTS.

Alors que les logiciels système modernes exécutent du code provenant de nombreuses sources avec des degrés de confiance variables, l'abstraction traditionnelle de la mémoire virtuelle ne permet pas l'isolation des parties non fiables d'une application partageant le même espace d'adressage virtuel. Tout le code s'exécutant au sein d'un processus a le même niveau de confiance. Par conséquent, un code tiers défectueux ou malveillant dans un processus peut compromettre le processus en divulguant ou modifiant directement la mémoire utilisée par d'autres composants de l'application. L'interface de la mémoire virtuelle doit être repensée pour permettre aux applications d'être compartimentées, implémentant le principe du moindre privilège en isolant les parties non fiables de l'application dans des compartiments avec un accès limité à certaines ressources de l'application. Nous présentons SecureCells, une nouvelle interface architecturale pour la compartimentation intra-espace d'adressage. SecureCells permet aux applications de définir des vues de mémoire garantie par le matériel pour les compartiments d'application avec des instructions d'espace utilisateur accélérant les appels inter-compartiments. Dans des microbenchmarks, SecureCells permet à un cœur ordré à 5 étages de passer d'un compartiment à un autre en seulement 8 cycles, réduisant le coût des transitions d'un ordre de grandeur par rapport aux meilleures alternatives. Nous construisons également un prototype complet de SecureCells, basé sur le cœur RISC-V RocketChip exécutant le noyau seL4, pour évaluer des benchmarks d'espace utilisateur.

De plus, cette thèse présente la première systématisation des connaissances sur les mécanismes de compartimentation, évaluant à la fois les propriétés qualitatives et quantitatives. Nous décrivons les propriétés de sécurité et de performance pertinentes pour une compartimentation pratique, et montrons dans quelle mesure chaque mécanisme fournit chaque propriété. Une revue complète des techniques de compartimentation vise à permettre aux développeurs de systèmes informatiques de définir de future interfaces sécurisées, performantes et utilisables pour soutenir la compartimentation généralisée des applications. Notre systématisation expose les lacunes communes de ces mécanismes, orientant ainsi les efforts de recherche future vers des opportunités de soutien à une compartimentation plus exhaustive.

Cette thèse soutient que les interfaces entre les composants des systèmes informatiques modernes entravent leurs garanties de sécurité, et explore les problèmes de deux interfaces majeures. Nous montrons qu'une refonte raisonnée de ces interfaces permet la mise en œuvre de systèmes plus sécurisés tout en soutenant les besoins d'applications haute performance, avec la conception et la mise en œuvre de Midas et SecureCells pour résoudre les défis aux interfaces noyau-utilisateur et intra-processus respectivement.

Contents

Acknowledgments **v**

Abstract **ix**

List of Figures **xvii**

List of Tables **xix**

1 Introduction **1**

1.1 Kernel TOCTTOU Protection Overview 5
1.2 Intra-address Space Compartmentalization Overview 6
1.3 Thesis Contributions 8
1.3.1 Midas 9
1.3.2 SecureCells 9
1.4 Thesis Organization and Details 10

2 Midas: Systematic Kernel TOCTTOU Protection **13**

2.1 Introduction 14
2.2 Background 15
2.2.1 Page Tables and Memory Protection 15
2.2.2 Linux Memory Subsystem 16
2.2.3 Supervisor Memory Protection 17
2.2.4 Double-Fetch Bugs 18
2.3 Threat Model 19
2.4 Attack Classification 19
2.5 Midas Design 21
2.5.1 Page State Machine 23
2.5.2 Discussion 26
2.6 Midas Implementation 31
2.6.1 Tracking Page State 32
2.6.2 Kernel Reads from User Memory 32
2.6.3 Handling Faults 35

2.6.4 Syscall Completion . . . 37
2.6.5 File System Writes . . . 38
2.6.6 New Mappings to Protected Pages . . . 38
2.6.7 Discussion . . . 39
2.7 Evaluation . . . 40
2.7.1 Mitigation of CVE-2016-6516 . . . 40
2.7.2 Performance evaluation . . . 41
2.7.3 Overhead breakdown . . . 44
2.7.4 Case study: Nginx . . . 45
2.8 Related Work . . . 46
2.9 Midas Summary . . . 48

3 SecureCells: A Secure Compartmentalized Architecture 49
3.1 Introduction . . . 50
3.2 Background . . . 52
3.2.1 Principles supporting compartmentalization . . . 52
3.2.2 Compartmentalization properties . . . 54
3.3 Objectives For Architectural Isolation . . . 54
3.3.1 Threat Model . . . 55
3.3.2 Security Objectives . . . 57
3.3.3 Performance Objectives . . . 58
3.3.4 Flexibility . . . 59
3.3.5 Alternate Visions for Compartmentalization . . . 60
3.4 SecureCells . . . 62
3.4.1 Access control . . . 63
3.4.2 Userspace Instructions . . . 64
3.4.3 Software Mechanisms . . . 68
3.4.4 Implementation . . . 69
3.5 Evaluation . . . 70
3.5.1 Security Evaluation . . . 71
3.5.2 Performance Microbenchmarks . . . 74
3.5.3 Compartment Switching and Access Control . . . 74
3.5.4 Compartmentalized pipeline . . . 76
3.6 Related Work . . . 78
3.7 Discussion . . . 80
3.8 SecureCells Summary . . . 82

4 SoK: Classifying Compartmentalization Mechanisms 83
4.1 Introduction . . . 84
4.2 Compartmentalization: A Deeper Dive . . . 85
4.2.1 Example: *Browser* Compartmentalization . . . 87

4.2.2 Execution Model and Threat Model . . . 90
4.3 Mechanisms Implement Restrictions . . . 91
4.3.1 Resource Access . . . 91
4.3.2 Operation Arguments . . . 94
4.3.3 Instructions and Operations . . . 94
4.4 Practical Considerations . . . 95
4.5 Evaluating Mechanisms . . . 97
4.5.1 Mechanisms Compared . . . 97
4.5.2 Methodology and Rationale . . . 108
4.5.3 Comparison . . . 108
4.6 Discussion . . . 114
4.6.1 Opportunities . . . 114
4.6.2 Limitations of our Methodology . . . 115
4.7 Summary . . . 116

5 Future Work 117
5.1 Beyond Midas: Double-fetch Protection and More . . . 117
5.2 Widespread Compartmentalization with SecureCells . . . 118

6 Conclusion 121

Appendices 123

A Midas Artifact 125
A.1 Artifact Appendix . . . 125
A.1.1 Description . . . 125
A.1.2 Installation . . . 126
A.1.3 Experiment Workflow . . . 127
A.1.4 Expected Results . . . 128
A.1.5 Artifact meta-information . . . 128

B SecureCells Details 131
B.1 Memory layout of the unified PTable-GTable . . . 131
B.2 Justification for Table 3.1 . . . 133
B.3 Existing mechanisms with SecureCells . . . 134
B.4 SecureCells Implementation Trade-Offs . . . 134
B.5 SecureCells ISA definitions . . . 135
B.6 Accelerating SecureCells Instruction Emulation . . . 137
B.6.1 Generic Trap-and-Emulate Acceleration . . . 137
B.6.2 SecureCells-specific PTable-GTable Access Helpers . . . 138

C Details for SoK on Compartmentalization Mechanisms 141
C.1 Classification Data 141

Bibliography 145

Curriculum Vitae 165

List of Figures

1.1 Illustrating the double-fetch attack at the system call interface 5
1.2 Illustrating intra-process trust components for an application 6

2.1 Example of a double-fetch bug. 18
2.2 State diagram for a page in Midas. 23
2.3 Diagram illustrating Midas preventing double-fetch exploitation 24
2.4 Diagram illustrating Midas preventing multi-page double-fetch exploitation 28
2.5 Illustration of a theoretical speculative leakage attack against Midas' protection. . 30
2.6 Midas' bookkeeping information for a page. 33
2.7 Midas: Flowchart for the transfer function `raw_copy_from_user` 34
2.8 Midas: Flowchart for handling a page fault. 36
2.9 Midas: Flowchart for handling a page fault to a COW page. 36
2.10 Midas performance comparison 42
2.11 Classification of overheads for various benchmarks due to Midas. 44
2.12 Midas: Request rates and throughput for the Nginx server 46

3.1 Browser compartmentalization with three compartments. 56
3.2 Permission transfers for a packet between SecureCells compartments. 56
3.3 SecureCells: Architecture. 62
3.4 SecureCells: State and userspace instructions. 65
3.5 Cross-compartment procedure call in SecureCells. 68
3.6 SecureCells performance comparison: `memcached` 75
3.7 SecureCells performance comparison: Network function 77

4.1 Architecture of a compartmentalized *Browser* 87
4.2 Classification of operations leading to a cross-compartment attack. 92
4.3 Scoring security properties of mechanisms. 109
4.4 Scoring practicality and performance aspects of mechanisms. 110

B.1 Layout of SecureCells' unified PTable-GTable. 132

List of Tables

2.1 Attack vector classification for TOCTTOU exploits. 19
2.2 System calls uninstrumented by Midas. 27
2.3 Prominent syscalls used by OSBench microbenchmarks. 42

3.1 Qualitative comparison of compartmentalization mechanisms 61
3.2 Overview of SecureCells' userspace instructions. 64
3.3 HW configuration of the SecureCells prototype. 71
3.4 FPGA resource utilization for SecureCells' MMU 71
3.5 Compartment switching cost of various compartmentalization mechanisms. 73
3.6 Cycles for emulating SecureCells instructions. 74
3.7 Mapping SecureCells instructions to similar APIs 81

4.1 Summary of privileges restricted by compartmentalization mechanisms 93
4.2 Summary classifying compartmentalization mechanisms. 98

B.1 Mapping SecureCells instructions to the RISC-V prototype instructions 136
B.2 Instruction encodings for SecureCells' prototype 136
B.3 SecureCells exception codes 139
B.4 Mapping SecureCells exception codes to values 140

C.1 Scoring matrix for mechanisms for security guarantees 142
C.2 Scoring matrix for mechanisms for performance and flexibility metrics 143

Chapter 1

Introduction

The computing landscape is one of rapidly growing software and hardware complexity. Modern computing systems inherit many of the abstractions and interfaces developed at the inception of personal computing and mainframe servers, but face a very different computing environment. The complexity and scale of computing systems have increased exponentially over the decades, far outscaling the ability for developers to exhaustively test and verify their systems, leading to an abundance of bugs. Popular software runs hundreds of millions of lines of source code, with millions of lines worth of changes every year. Linux, a widely-used operating system, itself currently accounts for 23 million lines of code, roughly increasing 20x since the turn of the millennium. Transistor counts for processors have roughly followed Moore's law for the last 50 years, and a modern Apple M2 chip has more than a 10^{11} transistors compared to the 10^4 transistors of the Motorola 68k processor from 1979. This scaling in complexity relies on, and also enables, pervasive code sharing. A web browser (e.g., Firefox) along with the underlying operating system (e.g., Linux) accounts for hundreds of millions of lines of code including

- mainline kernel code written by a informal melange of developers distributed across the globe with varying industrial/academic/governmental/individual affiliation,
- numerous device drivers written by the respective hardware vendors,
- shared libraries and modules developed independently and written in a plethora of languages,
- code from websites, often embedding other pages, written by their respective web developers.

Modern systems need to consider the threat of bugs in shared code compromising their security, and implement the necessary mitigations. For example, a web developer must account for the threat that any of the thousands of JavaScript packages their code depends on might be malicious, currently or in the future. The security of computing is reliant on the security properties at the interfaces between components (trusted and untrusted). The abstractions and interfaces at the core of computer systems need to enable systems to face up to today's security challenges.

While computing systems have evolved significantly, interfaces between many parts of this system have remained relatively unchanged. The designs of these interfaces reflect the requirements and threat landscape from their respective design periods, but fail to adequately address modern needs. Linux, for example, started development in the 1990s and is heavily inspired by Unix, initially developed in the 1960s-70s. The Linux kernel system call interface is cognizant of the threat of attacks or faults from untrusted userspace compromising the kernel, reflecting the contemporary shared use of mainframes running applications for a limited set of known users. System calls, therefore, typically check the validity of arguments passed from userspace. However, computing and the associated threat vectors developed over the past 30 years, rendering Linux' original system call interface dated. Linux has evolved from being a hobby project to becoming the operating system underlying critical computer infrastructure forming the backbone of industries and governments. The threat model has simultaneously evolved from accidentally triggered bugs in applications leading to crashes, to sophisticated exploits developed by experienced attackers targeting remote control of the entire system. One consequence of the dated designs of key interfaces is that these interfaces might not adequately mitigate threats that have arisen or worsened after the design of the interface. Since the 1990s, computing has evolved to support increased concurrency and true parallelism, with the rise in popularity of multicore CPUs and multi-threaded programs. For system call arguments stored in user memory and passed by reference, this evolution has enabled data races if the user program modifies the arguments from a second thread while the kernel executes a system call from one thread. This trend extends to other interfaces with widespread usage. The C programming language, dating back to the 1970s, remains in popular use despite the lack of memory safety by default. The rise of the internet and corresponding emergence of cyber warfare has exacerbated the effects malicious exploitation of memory safety bugs, leading to major vulnerabilities (e.g., Heartbleed [90], GHOST [91] and NetUSB [92]). A second consequence is that interfaces might not provide the correct abstractions to efficiently support designs that reflect trust relations within parts of modern applications. For example, the dominant abstraction for isolation within userspace code is a process, and different applications typically execute in different processes to isolate applications from bugs in other applications. The process virtual memory interface is supported by hardware enforced page table permissions across major commercial instruction set architectures (x86, SPARC, ARM) developed across the 1970s and 1980s. Process isolation is well suited to the historical state of software development where software vendors wrote their own applications with limited code sharing between vendors. Processes isolated code for one application, from one software vendor, from other applications, from other software vendors or for another user. With more code within applications originating from third-party developers and less trust in code run within a single process, intra-process isolation has become crucial. Security-critical programs, like browsers, microservices and microkernel operating systems (OSs), refactored to enforce intra-application isolation using processes are limited by the high overheads of this abstraction, and only support coarse-grained isolation to cap performance overheads [11, 86, 88].

Recognizing emerging threats, key interfaces have gradually evolved, fixing bugs and introducing new defense features. Most improvements tend to be incremental. By adding a NX/XD bit to mark pages as non-executable, popular processor architectures added support for Data Execution Prevention (DEP) and prevented code injection exploiting buffer overflows. Over the years, millions of commits have added various patches to Linux. However, many improvements address individual bugs while failing to comprehensively improve an interface's security. Continuing industrial and academic efforts have proposed various improvements to improve the kernel-user boundary and add support for isolation within an application. The security of operating system kernels is an area of active research, and many methodologies have been proposed to fix the issue of data races at the system call interface, primarily focussing on finding and fixing instances of this class of bugs. One group of proposals leverage static analysis [80, 136, 138, 147] to comb through the kernel codebase looking for vulnerable double fetches. These static analysis techniques have progressively improved from matching code against known double-fetch bug patterns to symbolic execution-based analyzers. Alternative proposals leverage dynamic analysis to detect instances of data races at runtime [59, 114, 143] by tracking kernel memory accesses while executing various common workloads (for e.g., booting up, running a browser, playing multimedia). Eliminating the discovered data race bugs by fixing source code allows the kernel to present a more secure interface, though the invulnerability to data races remains an informal assumption rather than a guarantee. Finally, a proposed mitigation [114] repurposes a CPU-specific feature designed for accelerating database transactions to instead dynamically detect user updates between kernel double fetches, rolling back the system call execution to prevent exploitation. Bug squashing techniques, however, are insufficient — they can only fix bugs found — and both static and dynamic analysis are incomplete. The significant churn in code further complicates the challenge, as every change potentially introduces new double-fetch bugs. The proposed mitigation is also inadequate, since the protection depends on a vendor-specific feature which has since also been deprecated on newer processors. The system call interface requires a more principled and reliant mitigation against data race attacks. Similarly, while researchers and processor vendors continue to propose mechanisms for finer-grained isolation within a process' address space, these mechanisms vary in their design goals, and do not adequately support widespread adoption of compartmentalization. Mechanisms which prioritize backward compatibility with existing systems [40, 52, 55, 69, 76] introduce the security benefits of intra-address space isolation but continue to suffer the consequences of other legacy design choices, such as expensive supervisor-mediated context switches. Some mechanisms [52, 96] trade off performance for security, either providing weaker security than processes to provide better performance or making restrictive assumptions on application use cases. A common approach among researchers is to retrofit protections to vulnerable interfaces abusing the side effects of unrelated mechanisms co-existing on the systems, bringing immediate protection to certain systems at the cost of a few fundamental shortcomings. Researchers have applied this approach to both mitigate double-fetch bugs [114] using Intel TSX and to implement compartmentalization [52, 69] using Intel VT-x. This approach is limited by dependence on specific systems (using Intel CPUs, for example) and potentially inhibits concurrently using applications requiring these features for their proper functioning. Most importantly, such defense mechanisms lack principled design and resemble targeted protections rather than fundamental security guarantees baked into interface design.

Security guarantees should inform the design of interfaces — either by extending or redesigning interfaces. At the kernel-user interface, we see that data races require the kernel to access the same argument in user memory at least twice, which allows the user to modify the data in the meantime. In fact, such bugs which are generally called double-fetch bugs commonly (but not exclusively) manifest from the same usage pattern. The kernel first loads arguments once to check their validity, then loads them at a later time in order to use them. This pattern earns these bugs the popular moniker of Time-of-Check to Time-of-Use (TOCTTOU) bugs. Double-fetch bugs contain an implicit assumption by the developer that the fetched data is the same, which may be violated by another thread through a concurrent modification. OS kernels generally use a software interface to access user memory, to manage protections like Supervisor Memory Access Prevention (SMAP), and we can extend this interface with a secure invariant. Within userspace applications, the requirements for isolation have changed drastically: from isolating per-user processes which occupy millisecond-scale scheduling slots to finer grained module or library-level isolation, which demands sub-microsecond operations (for example, switching trust domains). An userspace process needs to be further divided into isolated compartments which can communicate along well-defined APIs. OS-based mechanisms are expensive — even supervisors optimized for inter-process communication (IPC) on commodity hardware achieve microsecond-scale compartment switches at best. We notice that the traditional trusted-computing base (TCB) includes the processor hardware alongside the supervisor. Hence, we can securely delegate particular operations (access control, inter-compartment control, and data flow) from the supervisor to the hardware, improving performance while maintaining the same security guarantees. Improvements and trends in microarchitectural design, such as the move towards virtual memory area-based access control in the core's translation-lookaside buffer (TLB) [51], greatly assist in this transition.

In this thesis, we present secure designs for the user-kernel interface used by processes, and for interfaces between intra-process domains, both of which are security- and performance-critical. Midas provides systematic protection to the user-kernel interfaces against double-fetch attacks by maintaining an invariant: *through a system call's lifetime, every read to a userspace object will return the same value*. Midas can also be extended as a sanitizer, enabling detection of TOCTTOU attacks against the kernel. To validate the design, we also present an implementation of Midas on the Linux kernel. SecureCells, meanwhile, is a compartmentalization mechanism providing isolation between interacting userspace compartments. SecureCells' design is based on three pillars: hardware-enforced access control for isolation, unprivileged instructions for accelerating common operations and flexible software operations where necessary. SecureCells is the first mechanism to combine the security and performance requirements for flexible fine-grained intra-address space compartmentalization.

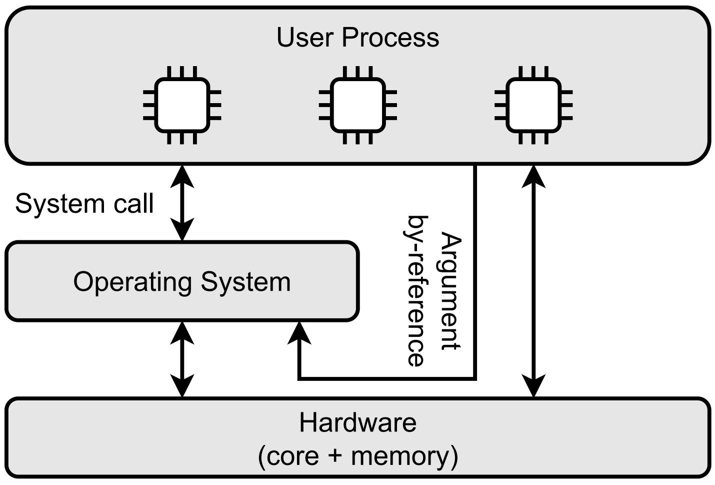


Figure 1.1: Illustrating the double-fetch attack at the system call interface

1.1 Kernel TOCTTOU Protection Overview

The operating system (OS) kernel is a key component of modern computer systems, tasked with multiplexing resources like memory, execution time and I/O among users on a shared machine, or among different tasks by the same user. The kernel is part of the system's trusted computing base (TCB), and interacts with untrusted userspace processes through system calls (syscalls). The userspace/kernel interface is a security-critical barrier, and forms the primary attack vector for attacker processes to compromise an entire system. The kernel must, therefore, implement extensive checks at this interface to protect itself from malicious arguments to syscalls. Most modern OS kernels trace their heritage to systems designed or developed in the 1980's and 90's, and inherit many of their system calls. The Linux, Darwin/XNU (used by MacOS) and FreeBSD kernels are all mostly compatible with the POSIX interface, first defined in 1988 [9]. The POSIX interface, itself, draws inspiration from the UNIX kernel first published in 1971. Over this time, the computing landscape has evolved immensely. Whereas the original UNIX kernel was not designed for multi-tasking, the modern desktop, server or mobile computing environment involves a multi-user, multi-tasking, multi-processing systems connected via internal or internet interfaces. Kernel security has come under ever-increasing threats, and requires stronger protection guarantees.

Untrusted userspace processes interact with the kernel using system calls, passing arguments by value (through registers) or by reference (in memory), as illustrated in Figure 1.1. When an argument is passed by reference, and the kernel loads the same value twice, an attacking user process can leverage the temporal window between the loads to modify the value in memory, potentially triggering a kernel bug. Double-fetch bugs plague operating system kernels, but also extend beyond to the similar OS-hypervisor interface [26–32, 34–36]. For example, the user could pass a buffer, and its corresponding length as arguments, then later maliciously change the length to influence the

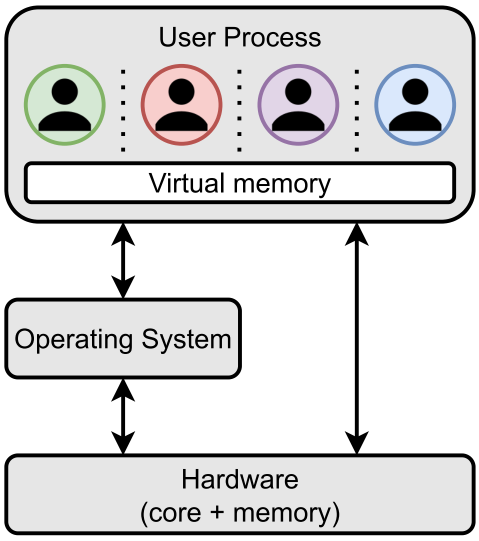


Figure 1.2: Illustrating intra-process trust components for an application

kernel to access memory outside the buffer. A time-of-check to time-of-use (TOCTTOU) violation occurs when the first read is used to validate an argument (example, the length above) and the second read is to use the argument. More generally, a double-fetch bug manifests as system call code which reads the same argument (passed by a user application by reference) two or more times. Double-fetch bugs might be particularly difficult to identify, as the two reads might be in entirely different parts of the kernel, or even in external code loaded through the eBPF interface or as modules. Additional kernel security, such as through system call filters like SecComp [115], could also introduce double-fetches if extended to include "deep argument inspection" (i.e., arguments passed by reference).

A systematic mitigation for double-fetch bugs must guarantee that a system call will always read the same argument values. Therefore, the mitigation must prohibit or hide all concurrent changes to memory objects accessed by the kernel during the execution of a system call, including writes from threads in the same process, other processes, or from concurrently executing system calls. The userspace memory access interface can be tasked with providing the required guarantee for argument accesses.

1.2 Intra-address Space Compartmentalization Overview

The complexity and rapid pace of change of modern software systems inevitably leads to a plethora of bugs across the stack. Open source projects regularly encounter and fix a steadily increasing stream of bugs and vulnerabilities in their codebases [124–126]. System developers heavily rely on abstraction and isolation to tackle application complexity stemming from interacting subsystems, an extensive list of shared libraries, plugins, interpreted code and on-demand downloaded code, interacting over untrusted I/O interfaces such as the network, disks, various accelerators, and peripherals. Each software component hides much of its complexity behind an accessible interface (commonly called Application Programming Interfaces or APIs). Traditional threat models have resulted in isolation at a few security-critical interfaces. The operating system (OS) kernel tasked with system management is already isolated from userspace processes running untrusted applications. Commercial hardware provides the abstraction of privilege levels, allowing the kernel to reliably isolate itself within a separate level. The kernel isolates applications from different users and different applications from the same user using a common abstraction: processes. Processes protect applications from other faulting or malicious applications, with isolated per-process virtual memory spaces and kernel resources. These abstractions provide systems crucial security and robustness guarantees. Applications cannot access other applications' memory spaces, or the kernel's data. The kernel can gracefully handle an application faulting, killing the corresponding process without affecting itself or other applications. Essentially, these abstractions work to mirror the trust relations between components of the massive code base. However, existing interfaces fail to protect systems against more recent threat models.

Rapid development in computing, supercharged by the explosion of the internet, has resulted in applications which cannot trust all code executing within its process. A bug in the one logging module allowed attackers to leverage the Log4J vulnerability to compromise entire server applications, and remotely take over the machines running these applications. A browser, for example, contains hundreds of shared libraries and executes code downloaded from untrusted websites. To prevent website code, controlled by a remote adversary, from directly accessing local resources, modern browsers are already compartmentalized into two components: an internet-facing rendering engine running in one process interacting with a separate local system-facing kernel process using a well-defined API over remote procedure calls (RPCs). This architecture is motivated by the browser's strong security requirements, but remains limited by the coarse-grained abstraction of isolation (processes) available on traditional systems. Bugs in the sandbox within the rendering engine can still compromise all other components in the sandbox, including the just-in-time compiler. The first key limitations of the process abstraction is that all code within a process is equally privileged and can equally access all of that process' resources including memory. The second limitation of this abstraction is that interactions between processes require system calls incurring microsecond-scale overheads. The first limitation prevents applications from implementing and enforcing barriers expressing the complex trust relations between code components. Applications rely on complex software isolation techniques like sandboxing and software fault isolation, which are buggy at scale. The second downside limits how finely applications can be decomposed into processes, due to performance overhead considerations. However, the process abstraction is flexible and widely supported, and remains the mechanism of choice for usable isolation.

Modern software requires an intra-address space compartmentalization mechanism that provides strong isolation for application components running within the same address space (see Figure 1.2), with low-overhead nanosecond-scale operations to support compartments with short nanosecond-scale execution timescales, all while maintaining the flexibility to support a variety of software trust relationships. In this thesis, we highlight that the limitations of the process abstraction stem from the software-hardware design of virtual memory. First, page-based virtual memory requires permission and translation tracking at page granularity, and near-core permission caching buffers (TLBs) whose entry count cannot scale with the rate of growth of memory. Second, the privileged kernel is tasked with changing between memory permissions (involving changing page tables) and incurs the unacceptable cost of kernel entry and exits.

Systems can implement secure and performant compartmentalization by moving key checks and operations from the supervisor into the hardware. While the hardware is part of the trusted-computing base, its view of virtual memory remains rooted in the designs of the 80s. A compartmentalized abstraction of virtual memory, with the hardware capable of tracking compartments and enforcing the requisite permissions to memory, can eliminate the kernel overheads while preserving strong security checks. Further, the hardware can accelerate specific common compartmentalization operations for data and control flow if it is aware of compartments. Finally, operations which do not benefit from hardware acceleration can be retained in software, retaining the accompanying flexibility.

1.3 Thesis Contributions

This thesis aims to protect systems by redesigning interfaces to satisfy the security and performance requirements of modern and future computing systems, against emerging threat models. While the security of current systems is of paramount importance, the performance of these systems must also satisfy strict deployment requirements. Foremost, we prioritize the security of our proposed interfaces, and consider performance as a crucial secondary requirement. For the user-kernel interface, we consider compatibility with the existing system call semantics as an essential requirement. For intra-address space compartmentalization, we deem the flexibility of the interface to support varying software use-cases to be vital for adoption.

We present two redesigned security- and performance-critical interfaces, specifically the user-kernel boundary and intra-process trust domain boundaries. We add strongly-guaranteed protection against double-fetch bugs to the system call interface. Further, we introduce a intra-address space mechanism for isolating untrusted application parts. Finally, we present a comprehensive survey of existing and proposed compartmentalization mechanisms to enable a principled comparison of these mechanisms.

1.3.1 Midas

Midas presents a multi-versioning concurrency control mechanism, inspired from database systems, for maintaining a key invariant during user data accesses from system calls: *through a system call's lifetime, every read to a userspace object will return the same value*. A *security property* derived from this invariant is enforced — Midas uses kernel metadata to track userspace pages accessed, maintains page-table permissions to enforce immutability and leverages page faults to on-demand duplicate pages where necessary to preserve an original copy for a system call. A *correctness property* is also described in this thesis, showing how the system's execution remains correct under execution with Midas. Since, the user-kernel interface is performance-critical and affects the system's performance on syscall-intensive workloads, Midas' design optimizes for low overheads. As concurrent writes to system call arguments are practically non-existent for well-behaved programs, Midas strives to minimize expensive page duplications and relies on snapshotting for protecting accesses.

Midas has numerous use cases. First, Midas defends against existing, even potentially unknown, double-fetch bugs on current and future systems. Second, Midas can protect the kernel against double-fetch bugs in dynamically-added code such as modules and eBPF code. Third, Midas can enable system call filters to securely examine arguments passed by reference without introducing vulnerable double-fetches. Finally, Midas can protect older systems, where modules or other vulnerable components lack bug-fixing updates, with a single update to the kernel core.

We have implemented a Midas prototype for the Linux kernel, demonstrating its practicality, and evaluated the system's performance during system-call dependent workloads from the NAS Parallel Benchmark Suite (NPB) and the Phoronix Test Suite (PTS). Midas results in an average performance overhead of 3.7% on NPB and 3.4% on PTS. We also perform a security evaluation to demonstrate that Midas successfully stops an attack against a vulnerable system call.

1.3.2 SecureCells

In this work, we also present a comprehensive set of objectives for a compartmentalization mechanism to support widespread adoption. SecureCells presents a novel virtual memory architecture for secure, high-performance, flexible intra-address space compartmentalization. SecureCells maintains the strong security guarantees of process-based isolation for fine-grained compartments within a process, with a mix of hardware and software support. SecureCells tracks the compartment executing on a core, and implements access control to memory regions based on a supervisor determined permission table stored in memory. Additionally, SecureCells provides unprivileged instructions to implement fast compartmentalization operations, specifically inter-compartment calls, zero-copy permission transfer to data regions, and to manage lifetimes for data regions. Each of these instructions includes specific checks and controls to maintain specific security properties, including preventing privilege escalation, code injection and data races. Finally, SecureCells delegates operations to software when the corresponding hardware implementation would bring no advantage.

This work also describes our prototype SecureCells implementation, including the RTL description of an in-order core based on the RISC-V RocketChip design, a QEMU port for quick emulation, porting of the seL4 microkernel operating system, and simplified versions of server benchmarks. We investigate the performance characteristics of our prototype core, using microbenchmarks designed to test the limits of access control, compartment switching and dataflow between compartments, and compare them to related work. SecureCells' in-order core can switch between compartments (a key performance metric) in as few as 8 cycles, which compares favorably to state-of-the-art compartmentalization mechanisms and is orders of magnitude faster than the traditional process abstraction. We also demonstrate that SecureCells can help isolate the networking and data storage of a `memcached`-like benchmark with a small ($<3\%$) overhead even for the smallest requests. These improvements are a direct consequence of tailoring the software-hardware interface to the requirements of modern programs.

Thesis statement

Critical interfaces underlying computing systems must adapt to support mitigating emerging threats and to enable the performance demanded by modern applications. Interfaces at key trust boundaries, e.g., between untrusted intra-process components and at the kernel-userspace border, especially require strong isolation at low overheads.

1.4 Thesis Organization and Details

Thesis Organization. This thesis is distributed across three chapters. Chapter 2 describes Midas, the systematic mitigation to kernel double-fetch bugs. Chapter 3 describes SecureCells, a novel secure and performant mechanism for intra-address space compartmentalization. Finally, Chapter 4 contains a comprehensive comparison of compartmentalization mechanisms along qualitative and quantitative metrics.

Bibliographic Notes. This thesis was supervised by my advisors, Prof. Mathias Payer and Prof. Babak Falsafi. Sections of the thesis describe projects conducted in collaboration with academic peers, namely Uros Tesic, Florian Hofhammer, Yuanlong Li, Siddharth Gupta, and Andres Sanchez. This thesis contains contributions from the following conference publications:

- Atri Bhattacharyya, Uros Tesic, and Mathias Payer. "Midas: Systematic Kernel TOCTTOU Protection". In: *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*. Ed. by Kevin R. B. Butler and Kurt Thomas. USENIX Association, 2022, pp. 107–124. ISBN: 978-1-939133-31-1. URL: `https://www.usenix.org/conference/usenixsecurity22/presentation/bhattacharyya`

- Atri Bhattacharyya, Florian Hofhammer, Yuanlong Li, Siddharth Gupta, Andrés Sánchez, Babak Falsafi, and Mathias Payer. "SecureCells: A Secure Compartmentalized Architecture". In: *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*. IEEE, 2023, pp. 2921–2939. ISBN: 978-1-6654-9336-9. DOI: 10.1109/SP46215.2023.10179472. URL: `https://doi.org/10.1109/SP46215.2023.10179472`

During the course of his PhD studies, the author of this thesis also contributed to other projects resulting in the publications listed below. The results from these publications are not included in this thesis.

- Lana Josipovic, Atri Bhattacharyya, Andrea Guerrieri, and Paolo Ienne. "Shrink It or Shed It! Minimize the Use of LSQs in Dataflow Designs". In: *International Conference on Field-Programmable Technology, FPT 2019, Tianjin, China, December 9-13, 2019*. IEEE, 2019, pp. 197–205. ISBN: 978-1-7281-2943-3. DOI: 10.1109/ICFPT47387.2019.00031. URL: `https://doi.org/10.1109/ICFPT47387.2019.00031`
- Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. "SMoTherSpectre: Exploiting Speculative Execution through Port Contention". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*. Ed. by Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz. ACM, 2019, pp. 785–800. ISBN: 978-1-4503-6747-9. DOI: 10.1145/3319535.3363194. URL: `https://doi.org/10.1145/3319535.3363194`
- Atri Bhattacharyya, Andrés Sánchez, Esmaeil Mohammadian Koruyeh, Nael B. Abu-Ghazaleh, Chengyu Song, and Mathias Payer. "SpecROP: Speculative Exploitation of ROP Chains". In: *23rd International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2020, San Sebastian, Spain, October 14-15, 2020*. Ed. by Manuel Egele and Leyla Bilge. USENIX Association, 2020, pp. 1–16. ISBN: 978-1-939133-18-2. URL: `https://www.usenix.org/conference/raid2020/presentation/bhattacharyya`
- Siddharth Gupta, Atri Bhattacharyya, Yunho Oh, Abhishek Bhattacharjee, Babak Falsafi, and Mathias Payer. "Rebooting Virtual Memory with Midgard". In: *48th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2021, Valencia, Spain, June 14-18, 2021*. IEEE, 2021, pp. 512–525. ISBN: 978-1-6654-3333-4. DOI: 10.1109/ISCA52012.2021.00047. URL: `https://doi.org/10.1109/ISCA52012.2021.00047`

Chapter 2

Midas: Systematic Kernel TOCTTOU Protection

Double-fetch bugs are a plague across all major operating system kernels. These bugs occur when data is fetched twice across the user/kernel trust boundary while allowing concurrent modification, violating an implicit assumption at this interface. Double-fetches enable an attacker to illegally access memory, cause denial of service, or to escalate privileges. So far, the only protection against double-fetch bugs is to detect and fix them. However, existing methods to find double-fetch bugs are incomplete. The double-fetch problem also fundamentally prohibits efficient, kernel-based stateful system call filtering. Thus, we propose *Midas* to mitigate double-fetch bugs and secure the system call interface. Midas creates on-demand snapshots and copies of accessed data, enforcing the key invariant that throughout a system call's lifetime, every read to a userspace object will return the same value.

Midas shows no noticeable drop in performance when evaluated on compute-bound workloads. On system call heavy workloads, Midas incurs 0.2–14% performance overhead, while protecting the kernel against any TOCTTOU attacks. On average, Midas shows a 3.4% overhead on diverse workloads across two benchmark suites.

2.1 Introduction

The operating system (OS) kernel provides isolation between processes and is a key component of the system's trusted computing base. Each *untrusted* userspace process runs under a dedicated user in its own address space and must request resources (such as communication channels or changes to its address space) from the *trusted* kernel. The userspace/kernel interface forms an explicit trust barrier; all data that crosses this boundary in either direction must be carefully checked by the kernel. Userspace processes attack the kernel by issuing system calls (syscalls) that then trigger kernel bugs, elevating the privileges of the process. A common class of kernel bugs are so-called *double-fetch* bugs [117, 128, 137, 143]. They occur when higher-privileged code, such as the kernel, reads the same data from the lower-privileged address space twice. Double-fetch bugs are a *race condition* between threads of different privileges. A *time-of-check to time-of-use (TOCTTOU)* violation occurs when the first read is used to check a condition while the second read is used to modify state. An example of a double fetch bug is when the kernel reads the length of a buffer from userspace, allocates a kernel buffer, then reads the length a second time to finally copy the data from userspace to the kernel. An attacker may concurrently overwrite the length of the buffer (with a larger number) after allocation, causing the memory copy to overflow the kernel buffer. Double-fetch bugs are a frequent problem in kernels and hypervisors [26–32, 34–36]. Watson [140] blames an unfixable TOCTTOU constellation as a reason for the generic insecurity of *syscall wrappers*. Syscall filtering wrappers require that data read from userspace for the initial check remains the same when the kernel later uses it for computation. Therefore, such filters can currently only check arguments passed by value. Midas enables "deep argument inspection" for SecComp [115, 116] and Janus [134] (i.e., checks arguments passed by reference). Without Midas, such inspection is impossible: these checks introduce double fetches, and consequently TOCTTOU bugs.

To mitigate double-fetch bugs in the kernel, a system must prohibit *concurrent changes*[1]to memory accessed by the syscall. Attackers may find crafty ways to trigger such concurrent writes, including: *i*) direct writes from userspace (e.g., from concurrent threads), *ii*) kernel writes from syscalls (e.g., from concurrent syscalls), *iii*) modifying address space mappings, *iv*) concurrent `write` syscalls to a file that alters mapped file pages, and *v*) storing arguments on device-backed pages, leveraging devices to trigger concurrent writes. To prevent attacks, all concurrent writes must be prohibited.

"Through a syscall's lifetime, every read to a userspace object will return the same value."

We base our defense on one key invariant, stated above. From this invariant we derive a *security property* ensuring that every read during the execution of a syscall is tracked. Subsequent reads

[1]The attacker model includes both concurrent and parallel writes. For readability, this chapter refers to both writes as "concurrent".

from the same address will always return the same value. For performance, multiple versions of an object may exist simultaneously, depending on when the syscall was started and how many concurrent syscalls are in flight. Orthogonally, we derive a *correctness property* that ensures the sharing of the correct version among inflight syscalls. All writes end up on the most recent version of the objects, guaranteeing forward progress. While we implement this invariant in our Midas prototype for the Linux kernel, this defense applies to any modern OS kernel.

Our evaluation of Midas demonstrates low performance overhead. On workloads from the NAS Parallel Benchmarks suite, Midas shows an average performance overhead of 3.7%. Similarly, its performance overhead on more kernel-intensive workloads from the Phoronix Test Suite is 3.4% (with negligible memory overhead). Our security evaluation demonstrates how Midas successfully stops all attacks against vulnerable syscalls. Our contributions are:

- Distillation of TOCTTOU attack vectors into an invariant that protects the kernel against malicious concurrent modifications,
- Midas, a design that prohibits and detects TOCTTOU attacks against modern kernels, prohibiting their exploitation, enabling developers to detect TOCTTOU bugs, and providing the foundation for safe syscall interposition and validation, and
- An efficient implementation of Midas for the Linux kernel that exhibits low (3.4%) performance overhead.

2.2 Background

Midas orchestrates several mechanisms within the Linux memory subsystem to provide its protection guarantees. Linux uses architecturally defined per-address space page tables to define mappings to pages. Midas protects these pages by temporarily marking them read-only in the page tables. This section provides background information necessary to reason about why and how Midas protects syscalls from concurrent writes.

2.2.1 Page Tables and Memory Protection

Virtually all contemporary architectures (e.g., x86, ARM, SPARC, and RISC-V) implement separate virtual and physical address spaces (AS) based on fixed-size regions called pages. Programs execute in their virtual address space while caches and main memory are accessed with physical addresses. Architectures rely on page tables orchestrated by the operating system to translate between these address spaces and to protect memory accesses. Page tables are arranged as radix trees, where different bits of the virtual address are used as indices into successive levels of the page table.

At the leaf page table, a unique page table entry (PTE) stores the translation and protection information for a page.

A PTE in x86-64 is a 64-bit value holding, among others, the following metadata: a *Present bit (P)* to mark the PTE's validity; *Protection bits (NX, R/W, U/S)* to restrict the type of access and the privilege level of the accessing code; *Software-usable bits (SW1-SW4)* that are ignored by the MMU and used by the operating system to store metadata; and a *Page Frame Number (PFN)* to identify the page's physical address. The *U/S* separates kernel (supervisor) pages from user pages. The *R/W* controls whether pages are writable or not.

An access using a virtual address first reads the corresponding PTE's present bit to check its validity. Then, the access checks whether the access is allowed from the executing code's privilege level by checking the U/S bit and whether the read/write access is allowed by checking the R/W bit. When all checks pass, the processor uses the PFN to find the data in the caches or in memory. When a check fails, the processor raises a protection fault/exception and moves control to an OS-specified exception handler.

Reading PTEs from a multi-level page table is an expensive operation, and modern processors cache PTEs in caches known as Translation Lookaside Buffers (TLBs) to reduce the cost of subsequent accesses. On most architectures, the OS is responsible for keeping TLBs coherent with the page table, necessitating entries to be flushed from TLBs when the corresponding PTE is updated. Modern multiprocessing CPUs have a TLB for each core, and PTE information may be cached in one or more TLBs. Therefore, when the OS changes a PTE value, the OS is also responsible for flushing remote TLBs (i.e., on another cores) as required. Commercial-off-the-shelf systems typically rely on an inter-process interrupt (IPI) to the OS running on a remote core followed by a local TLB invalidation. Due to the scalability limitations of broadcasting IPIs to remote cores, the TLB invalidation process is highly optimized in software, and has received dedicated hardware support in more recent processors and in research proposals [108, 132, 150].

2.2.2 Linux Memory Subsystem

Linux implements various abstractions—including processes, files, and shared memory—using the architecture's page tables. All threads within a Linux process share a single address space, and consequently use the same page table for translation and protection. Each page within the process' virtual address space may be mapped or unmapped. Mapped pages have separate read/write/execute permissions. Programs typically have write-execute exclusion, meaning code pages cannot be written to and data pages cannot be executed. These permissions map directly to page-table bits. Pages in Linux may also be copy-on-write (COW) pages, which are mapped read-only in multiple address spaces, but duplicated when any process writes to it, resulting in a separate copy.

Linux maintains userspace and kernel mappings to memory in distinct parts of the virtual address space. The PTE entries for kernel mappings, located in the top half of the address space, have the U/S bit set. These kernel mappings are identical for all address spaces, and are kept consistent across the corresponding page tables. In contrast, the PTE entries for userspace mappings, located in the bottom half of the address space, have the U/S bit reset. A userspace page has at least one userspace mapping and at least one kernel mapping. Shared userspace memory is implemented by mapping a page in more than one address space.

Files in Linux occupy a separate namespace (rooted at /). However, when files are read or written, parts of the file are cached in the kernel's page cache (which consists of pages mapped in the kernel's address space). Moreover, programs can explicitly map pages from a file, in which case the corresponding pages from the page cache are also mapped at userspace addresses in the process' page table. Mapped file pages can therefore be accessed by the file-system driver using kernel addresses, and userspace programs using userspace addresses. Userspace pages not backed by a file are called *anonymous pages*.

Processes in Linux can also share memory pages, which might be set up implicitly by memory mapping pages from the same files, or explicitly through named shared memory objects. The same physical page might be mapped in two or more processes' virtual address spaces, and be accessible through the respective virtual addresses in each process. Linux maintains a reverse map for each physical page mapped to user virtual memory, linking from the pages' metadata to the PTEs for each mapping. Reverse mappings enable Midas to precisely and efficiently identify all possible addresses that may be used to modify the contents of an user page.

2.2.3 Supervisor Memory Protection

Kernel accesses to userspace memory use userspace mappings, introducing the risk of the kernel confusing userspace data structures for kernel data structures. An attacker can exploit this behavior via bugs in the kernel. Essentially, the attacker needs to set up either data structures or code within its accessible memory, then exploit a kernel bug to make the kernel use these data structures, or execute this code. This class of attacks is known as confused-deputy attacks, since the privileged software (kernel) tasked with isolating userspace processes confuses its own code/data with the user's.

Architectures and OSs have mitigated these vulnerabilities by introducing *supervisor memory protection*. Under supervisor memory protection, kernel read/write/execute access to userspace memory raises a fault (depending on the state of a per-core system register). On x86-64, these features are known as Supervisor Memory Access Protection (SMAP) for data accesses, and Supervisor Memory Execution Protection (SMEP) for code accesses. Bits in the CR4 register track whether these protections are active, and the privileged `stac`/`clac` instructions are used to quickly

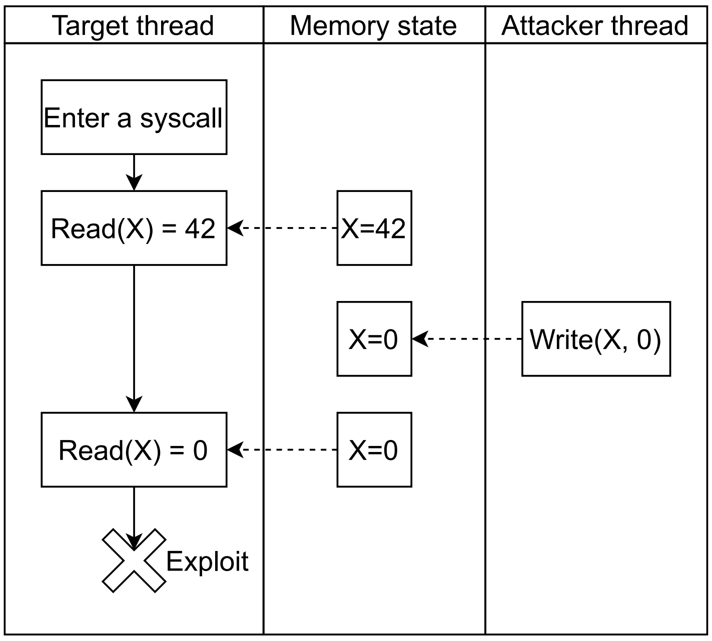


Figure 2.1: Example of a double-fetch bug.

enable and disable SMAP. In the OS, all accesses to userspace memory are made explicit, using *transfer functions* to read from and write to userspace memory. Any unintended access outside these functions causes a hardware fault, indicating a kernel bug or an attack. Linux implements the `copy_{from/to}_user` functions, which use the access control instructions to disable SMAP before accessing userspace data, and then re-enable SMAP afterwards. Transfer functions make kernel accesses to userspace data explicit, allowing Midas to reliably track and protect *every* kernel fetches from userspace memory.

2.2.4 Double-Fetch Bugs

Double-fetch bugs occur when a privileged environment (such as the kernel) reads untrusted memory multiple times, returning different values each time. Such a situation is depicted in Figure 2.1, where the value of `X` in memory is changed by an attacker between two reads by the target thread. Exploiting such a bug requires a race condition i.e. accesses to memory in a particular order across threads. A specific variety is the *time-of-check to time-of-use* (TOCTTOU) bug which occurs when the first fetch validates an object's value and the second fetch uses the same object's value. TOCTTOU bugs are widely studied in file systems, where the API makes it possible to swap the file after validating the access rights [43, 97, 103, 127, 141]. TOCTTOU bugs affect both kernel [59, 137] and dynamically-loaded driver code [32, 33]. Wang et al. [137] showed that double fetches appear not only in kernels, but wherever there is a trust boundary to cross (e.g., kernel—hypervisor [143] and hardware—kernel boundaries [79]).

	Userspace	Kernel	Device
Existing mapping	Intra AS Cross AS	User mapping Kernel mapping	DMA MMIO page

New mapping	`mmap`
	`clone`
	`swap`
		`mm_populate`	New
			DMA/
			MMIO page

Table 2.1: Attack vector classification for TOCTTOU exploits.

2.3 Threat Model

The attacker has access to a user account on the target machine. They can execute arbitrary userspace code, including syscalls. Some system calls have double-fetch vulnerabilities which the attacker wishes to exploit (e.g., for privilege escalation). The attacker may execute arbitrary sequences of syscalls on multiple CPU cores in parallel, or concurrently on the same core. However, the attacker is not allowed direct access to devices, as only the privileged root user is permitted this capability.

Midas mitigates any unintended corruption or information leakage *in the kernel* or *in other user processes* that arises through double-fetch bugs. Hardware attacks such as Rowhammer [89] or side-channels [65], and file-system TOCTTOU attacks [97, 103, 127, 141] are out of scope.

2.4 Attack Classification

Midas guards data processed during a syscall's execution against concurrent modification. We label the data fetched twice as vulnerable data. In this section, we classify attacks based on two criteria: the privilege level of the writer, and whether the mapping used for writing exists at the time of the first read. Table 2.1 summarizes our classification. Importantly, this classification helps understand existing attacks and how to protect against them, and where future attacks (bugs) may arise. The device column corresponds to attacks where a device (e.g., a network card, GPU, FPGA) is responsible for modifying vulnerable data. Watson [140] describes a subset of the following attack vectors.

Existing userspace mappings to a page can be used to modify vulnerable data which the targeted syscall is reading. Userspace can directly write to a mapped page, irrespective of whether the mapping is in the same address space or not. Alternatively, a concurrently executing syscall can also modify the vulnerable data in a *confused-deputy* attack. When the attacker passes a pointer to the vulnerable data to the syscall as a user buffer in which the syscall can return some data, the kernel's write to the buffer can modify vulnerable data. For example, the `read` syscall takes an argument pointing to a user buffer where the contents of a file will be copied to. Another example is `rt_sigaction`, where the kernel writes to a user buffer pointed to by the `oldact` argument. In both of these attacks, the malicious write uses a userspace mapping. *A protection mechanism must account for all userspace mappings to pages containing vulnerable data at the time of the targeted syscall's first read.*

Existing kernel mappings to a page also mapped in userspace can be leveraged by an attacker in a confused-deputy attack. Here, the attacker maps a file-backed page from the page cache into a userspace process and then passes as an argument in this page to the target syscall. The attacker then triggers a concurrent `write` syscall to modify the vulnerable data using kernel mappings for the page cache pages. The kernel does not explicitly track kernel addresses mapping to a page, but the file-system driver does explicitly find the page before writing to it. *A protection mechanism must instrument file-system drivers to account for writes via kernel mappings to vulnerable data.*

The kernel might create new mappings to the vulnerable data between the double fetches by the target syscall, bypassing any protective permissions installed by the transfer function in PTEs at the time of first read. An attacker can call `mmap` and `clone` to create a new mapping to the vulnerable data before writing to it. The page-table mapping might not be created at the time of the malicious syscall, but lazily when the attacker writes to the vulnerable data due to demand paging. In a more involved variant, the attacker can use the kernel as a confused deputy which touches the unmapped page and maps it in, then writes to the vulnerable data. In all the above vectors, the function populating pages for a process (`mm_populate` on Linux) is creating the new mapping. *A protection mechanism must instrument any syscalls and other kernel mechanisms which can create new mappings.*

Swapping may also create a new page-mapping. If the attacker writes to a page that was previously swapped to disk, but later swapped in to be read by the target syscall in a different address space, the kernel might lazily reinstate the attacker's mapping to the page. *The swapping mechanism must be protected.*

Midas protects against all the previously-listed attack vectors. In the absence of any other syscall which can create new userspace mappings to vulnerable data, Midas' protection is complete against writes from both user and kernel code.

Finally, a device might modify vulnerable data if it is either allowed to DMA (direct memory access) to the page, or if the page is memory mapped (MMIO) and is actually backed by the device. In the latter case, external factors can change the vulnerable data. Existing discretionary access control rules typically prevent users (except a superuser) from mapping device-backed pages into their address spaces. Such users are also disallowed from configuring DMA devices. Thus, device modifications to vulnerable data fall outside our threat model and are not protected by Midas. However, Midas can be extended to protect against modifications by DMA devices on processors supporting IOMMUs or similar methods for access control [94]. As a superuser can modify kernel code via kernel modules, protecting against attacks from this user falls outside our threat model.

2.5 Midas Design

Midas maintains a single, core, *invariant*: ***through a syscall's lifetime, every read to a userspace object will return the same value***. By construction, the invariant guarantees that double-fetches in syscall code will read the same data, *eliminating TOCTTOU bugs*. Midas maintains the invariant by tracking *snapshots* of objects when first accessed, lazily making *copies* when the object is concurrently written and accessing the correct copy on subsequent reads. Copies are only maintained during syscalls' lifetimes, and are released as soon as no syscall needs it. Consequently, each userspace object has a single copy when no syscalls are running. The invariant also means that only accesses to userspace objects by the kernel need to be protected. Accesses to userspace objects from userspace and kernel objects by kernel code remains unaffected.

Midas' implementation builds on the protection mechanisms provided by existing virtual memory implementations. On modern platforms, virtual memory protection is set up by the OS at page granularity by setting bits in page table entries (PTEs). These permission bits are checked by the hardware on memory access, efficiently enforcing the permissions, and raising a fault when they are violated. For performance, Midas implements its invariant at page granularity, not object granularity: when a syscall reads from userspace, every page touched by that read is covered, not merely the bytes read. Page-granularity protections are conservative compared to byte-granularity protection and Midas maintains its invariant. As a side effect of its implementation, Midas does not distinguish accesses to different parts of a page (intra-page false sharing). False sharing leads to unnecessary page duplications, incurring performance overhead on highly shared pages, but does not affect correctness.

For an object spanning multiple pages, Midas' design sequentially protects each page before reading from it. The leading pages containing the object are protected before the later pages, allowing an attacker to potentially modify the later pages before the syscall first reads them. However, the attacker is prevented from modifying any of these pages after the syscall's first read, ensuring that double fetches respect the invariant. If the syscall code contains a TOCTTOU bug, the modification will be visible to the first fetch itself (which is used for checking for validity of the data) and will lead to the data being rejected straightaway. Midas' invariant therefore prevent exploitation of double-fetch vulnerabilities even when the fetched objects span multiple pages. We elaborate on this case with an example in Section 2.5.2.

A major requirement for Midas is to allow concurrent access to pages by user/kernel code running in parallel with a syscall which reads from the same pages. This requirement prevents deadlocks and improves performance *vis-a-vis* a naïve design which blocks all other tasks writing to pages already read by a syscall until the syscall completes. The naïve design can deadlock because it introduces dependencies between tasks for forward progress, which we illustrate in the following example of a system with two tasks (A and B):

- Task A issues a blocking syscall which reads a user page and blocks, then
- Task B writes to the same user page before issuing a syscall which resumes task A.

In this case, if Task A's read to the page precedes Task B's write, Task B will be blocked waiting for A to complete its syscall. Task A will also remain blocked waiting for Task B's syscall, introducing a circular dependency, leading to deadlock. The naïve design also introduces unnecessary delays in other cases, such as the one described below, again with two tasks (C and D):

- Task C reads from a page and sleeps for a long while, but does not read from the page a second time, then
- Task D writes to the same page after task C has read from it, and blocks until Task C completes and is unnecessarily delayed.

A more performant approach is to duplicate the concurrently accessed page: the copy is kept for task C for future fetches, and task D can write to the original and proceed without delays.

Midas must maintain multiple versions of a page read by a syscall to maintain its invariant in the face of concurrent writes. Midas introduces *snapshots* and *copies* to keep track of page versions. Snapshots are logical views of the page's contents at a particular time, while the actual contents are stored in one of many copies. Each snapshot maps to a copy, allowing the contents of the page at the time of creating the snapshot to be read. If multiple snapshots are taken without intervening writes to the page, these snapshots will map to a single copy, reducing Midas' space overheads and performance overheads for creating copies. Midas maintains a snapshot of every page when first read by a syscall. On a double fetch by the same syscall, the copy mapped to the snapshot is accessed, ensuring that the data read is the same as the first time. The latest copy of the page is used for all writes, by the syscall as well as from concurrently running tasks, updating the page as seen from userspace. Midas' design draws parallels to multi-version concurrency control methods for databases based on snapshot isolation [146]. Transactions read from a snapshot of the database state from when they started, and writes update the up-to-date state of the database. *Essentially, Midas is a multi-versioning system for pages where syscalls read from immutable versions to prevent TOCTTOU bugs, and syscalls and userspace both write to a single mutable version holding the latest state of the page.*

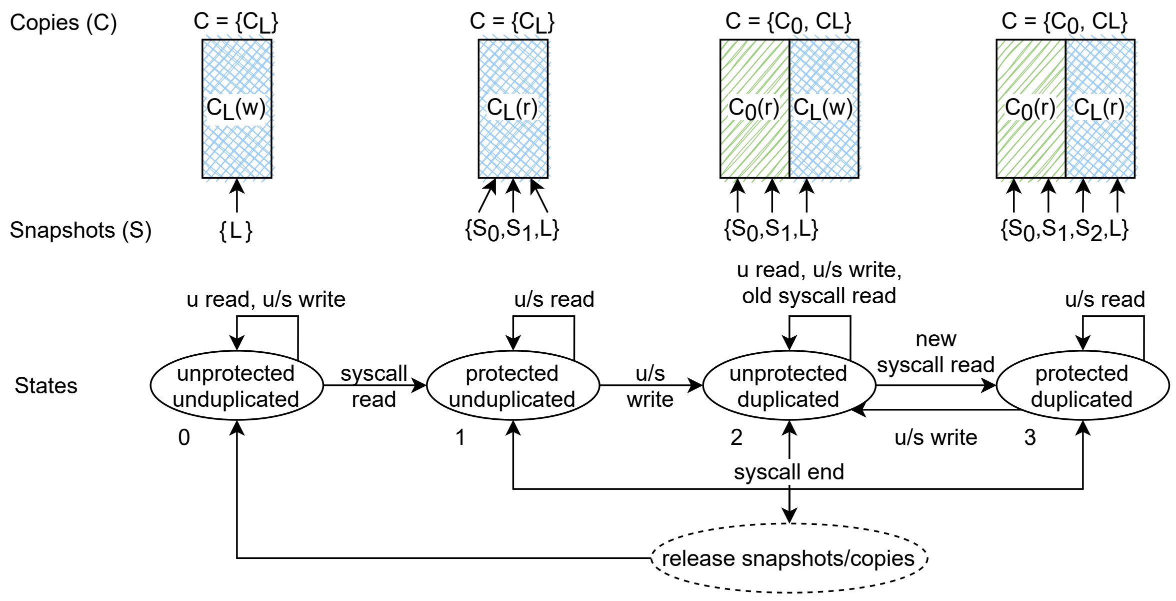


Figure 2.2: State diagram for a page in Midas. Reads/writes from userspace/syscall code are marked (u)/(s) respectively. Shading is used to represent the mapping from snapshots to copies.

2.5.1 Page State Machine

To track multiple versions of the contents of a page when being concurrently accessed by numerous tasks, from userspace or during a syscall, Midas implicitly maintains a per-user page state machine. For a page, its corresponding state machine *i*) tracks snapshots for currently executing syscalls which have read it, *ii*) tracks copies of the page, and *iii*) maintains the mapping between snapshots and copies necessary for providing the correct contents to subsequent reads.

Figure 2.2 shows the state machine for a single page. At every state, the page has two associated sets:

1. the copies set $C = \{C_L, C_0, \ldots\}$ holds multiple copies of the page over time, and
2. the snapshots set $S = \{L, S_0, S_1, \ldots\}$ tracks logical versions of the page, each corresponding to one executing syscall and each mapping to a copy.

Reads from kernel code in a syscall use the *snapshot's corresponding copy*. Writes from user/kernel code and reads from userspace access the *latest copy* C_L, which is mapped in processes' address spaces. All other copies are read-only (no matter what the original page protection is), and are used for providing snapshots to syscalls. Read-only pages only use states 0 and 1, and writes lead to segmentation faults (as they do on non-Midas systems). Knowing which state the page is in allows Midas to differentiate between faults due to Midas protecting pages and faults due to actual

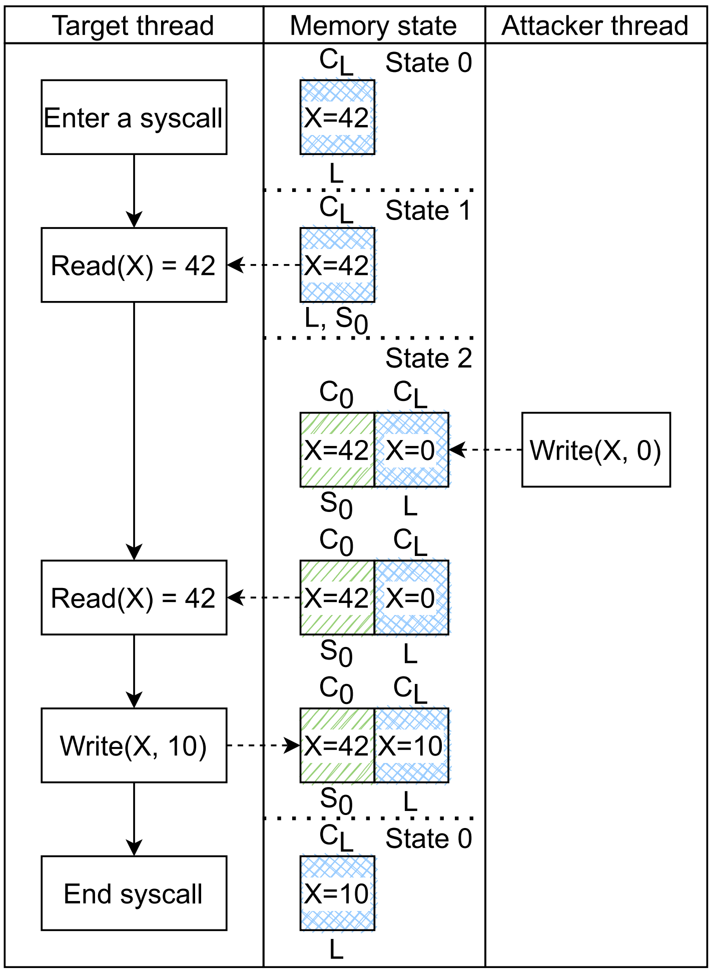


Figure 2.3: Diagram illustrating Midas preventing exploitation of a double fetch of object `X`.

permissions violations in userspace programs or the kernel. The latest copy C_L of read-only pages remains read-only in both protected states (1 and 3). In the following paragraphs, we describe how the state machine for a single, writable user page transitions between its states, what triggers each transition, and what changes are made to the copies and snapshot sets on a transition. In Figure 2.3, we illustrate how the state machine protects the syscall from Figure 2.1.

State 0. A page starts as (`unprotected, unduplicated`). In this state, there is a single copy C_L and a single "snapshot" L. The snapshot L refers to the latest version of the page which changes over time, and is the only mutable snapshot. All processes where this page is mapped have unrestricted userspace read and write access, and unrestricted kernel write access. The remaining operation, a read from kernel code, triggers a transition to State 1. In Figure 2.3, the snapshot L initially contains the value 42.

State 1. The page in State 0 transitions to the (`protected, unduplicated`) state as soon as a syscall reads from it. First, Midas marks the page's latest copy C_L read-only in all processes where the page is mapped, trapping writes to the page but permitting concurrent userspace reads to continue. A new snapshot, S_0 linked to this syscall is allocated for this page. For the rest of its lifetime, this syscall will only read this page from this snapshot. Both snapshots S_0 and L refer to the same copy C_L (shown by the blue cross-thatch in Figure 2.2). Prior to any writes to this page, any other syscalls which also read the page get their own snapshots (e.g., S_1) all pointing to the single copy C_L. The page's read-only status causes the hardware to fault on any write, notifying Midas to transition the page to State 2. In Figure 2.3, the page transitions to State 1 when the syscall first reads it, and adds a snapshot S_0.

State 2. A page in State 1 transitions to the `(unprotected, duplicated)` state on any write from user or kernel code. Midas duplicates the old contents of the page from copy C_L, creating a read-only copy C_0 (shown by green shading in Figure 2.2). Snapshots except L (i.e. S_0 and S_1) previously mapping to C_L are mapped to the copy C_0. The write then modifies the latest copy C_L, which is made writable again. Note how, in this state, any read using the snapshots S_0 or S_1 reads from the unmodified copy C_0 while writes directly affect C_L. Certain syscalls such as `rt_sigaction` both read and write from the same user page. A write by `rt_sigaction` to the page it has previously read will update the page's latest copy C_L, but not the duplicate copy C_0. Midas' write policy ensures that the copy C_L always holds the latest contents of the page, up-to-date with all the writes to the page, from both user and kernel code. Further, Midas does not need to merge writes from userspace and syscall code on a syscall's completion, since both directly modify the same copy C_L. All other copies C_i are immutable. When the attacker writes to the page in Figure 2.3, the page moves to State 2, linking the snapshot S_0 to a copy holding the original value 42. The writes from both the attacker and the syscall itself both affect the copy C_L, but the read from the syscall accesses the snapshot S_0 and reads the same value as the first time.

State 3. A separate syscall subsequently reading the page in State 2 transitions it to the `(protected, duplicated)` state. The new snapshot, S_2, points to the latest copy C_L. State 3 is similar to State 1, except that there are different copies of the page used for reading by different syscalls. The syscall for which S_0 was allocated will read from the copy C_0, while the syscall for which S_2 was allocated will read from copy C_L. On a write, the page transitions to State 2 and is duplicated again, creating another copy C_1: snapshot S_2 maps to C_1 while snapshots S_1 and S_0 continue to map to C_0.

Releasing snapshots. Midas uses snapshots to enable a syscall to read the same data from a page during its lifetime and releases snapshots when syscalls complete. Releasing a snapshot is possibly accompanied by a state transition and the release of the mapped copy. If S_i mapped to the latest copy C_L, Midas cannot free the copy since userspace is using it. In this case, the page must be in State 1 or 3, and C_L is read-only. After removing S_i, if L is the sole remaining snapshot mapped to C_L, Midas makes the page writable, moving to State 0 or 2 from State 1 or 3 respectively. If S_i is mapped to any other duplicate C_i, Midas frees the copy along with the snapshot if S_i is the last remaining snapshot mapped to C_i. If the page was in State 2, C_L was writable and unmapped by any snapshot, so Midas changes the page to State 0. This transition is shown in Figure 2.3, where the snapshot S_0 and the copy C_0 are both discarded. If the page was in State 3, C_L was read-only and mapped by some other snapshot, so Midas moves the page to State 1. Recall that all snapshots S_i except L are immutable. Any data written by the syscalls directly affect L. Therefore, dropping a snapshot S_i is trivial and does not require writes from the syscall to be merged into the latest copy.

2.5.2 Discussion

Correctness of syscalls directly updating snapshot L. Midas' design lets all writes, including those from syscalls, to directly update the latest copy of the page C_L and this property maintains correctness of system execution. We now show that there is a valid *safe* execution trace of a system not protected by Midas which generates the same sequence of writes to the page, and therefore generates the same contents of the page when the syscall ends. We define a *safe* trace as one that has no writes to vulnerable data between double fetches by the kernel, and therefore does not trigger any existing TOCTTOU bugs. By showing that the final contents of memory after a Midas syscall has a corresponding execution without Midas (which we assume to be correct) leading to the same contents, we can conclude that the execution of the Midas syscall is also correct. For this proof, we assume that no syscall reads the same object after writing to it (r-w-r pattern). Such syscalls do not exist in the Linux kernel, and are discussed below. Therefore, our syscalls write to an object after completing all of their reads of that object.

Consider a page holding a single-byte object O_0, and the sequence of operations to this byte during a Midas syscall be $Ops = \{Op_0, Op_1, \ldots\}$. Each operation is a tuple $(r/w, k/u)$ specifying whether the operation was a read or a write, and whether the operation was due to a user or kernel instruction. Suppose there was no attempt to exploit a TOCTTOU bug, i.e., between any two read operations by the same syscall, there was no write to this object. In this case, Midas reads the same value from its snapshot of the object as is present on the latest version. The same sequence of operations on a non-Midas system would be valid and safe, since the object value does not change between the kernel's double fetch and the syscall reads the same value on this system.

Assume there was an attempt to exploit a TOCTTOU bug: a write Op_1 exists between two syscall reads Op_0 and Op_2. Midas protects the syscall ensuring that Op_2 does not see the effect of Op_1 by reading from a snapshot instead of the latest copy C_L. Since our syscalls are assumed to not contain any r-w-r pattern, any writes by the syscall happen after Op_2. Let us assume that the syscall's write is Op_3. We can generate a valid, safe execution on a non-Midas system by moving the attacker's write to after the last read by the syscall, i.e., $Ops = \{Op_0, Op_2, Op_1, Op_3\}$. The syscall in this system reads the same value both times, and hence has the same execution as that in the Midas case. The value of the object when the syscall completes is that written by Op_3 in

System Call	Exemption reason
`futex`	Relies on concurrent write
`execve`	Remaps address space
`write`	Invulnerable, improves performance

Table 2.2: System calls uninstrumented by Midas.

both cases (or that written by Op_1 when the syscall does not have a final write). Since the syscall has the same execution and the final value of the object is the same, the execution of the Midas system is the same as that of the non-Midas system. In general, any trace of operations on a Midas system can be translated to a valid, safe trace on a non-Midas system by moving malicious writes to an object to just after the last double fetch of that object. Multiple syscalls in Midas can therefore write to the same object without affecting correctness, because an equivalent, valid, safe non-Midas trace exists where all the writes have been postponed, in the same order to after the double fetch reads.

Exemptions. Syscalls such as `futex` rely on user data changing between double fetches to implement their functionality and cannot be protected by Midas. These syscalls are listed in Table 2.2. The `futex` syscall implements a fast synchronization mechanism for userspace and relies on atomic writes from concurrent userspace threads to update a condition the syscall is waiting for. Subjecting a `futex` syscall to Midas' invariant will prevent it from ever waking up the waiting task. Such syscalls cannot be protected by Midas, and we implement an exemption list to prevent transitions in the state machines of pages read by these syscalls. The code for exempted syscalls must be manually inspected for double-fetch vulnerabilities. Crucially, exempting these syscalls from Midas' protection does not affect the security of other syscalls containing double fetches. Any writes from these syscalls are subject to the same rules described in the state machine, and cannot break Midas' invariant. Midas can also implement finer-grained exemptions based on syscall parameters. Those were not necessary for Linux.

Syscalls with read-write-read patterns. A (hypothetical) syscall that reads from an object, writes to it, and then reads back the updated object cannot be protected using Midas. Midas' invariant will ensure that the second read is identical to the first, and does not reflect the intermediate write. Such syscalls must remain exempt from Midas' instrumentation. During extensive tests, we did not find any other syscall which exhibits this behavior in the Linux kernel.

Syscalls with false sharing. Another hypothetical type of syscall could struggle with Midas' instrumentation due to false sharing. Suppose a page contains two objects, O_0 and O_1, and a syscall sequentially reads O_0 then O_1. Due to Midas' invariant being enforced at page granularity and false sharing of the page between these objects, Midas guarantees that the value of object O_1 read is the same as what was contained when it first read object O_0. A syscall requiring the

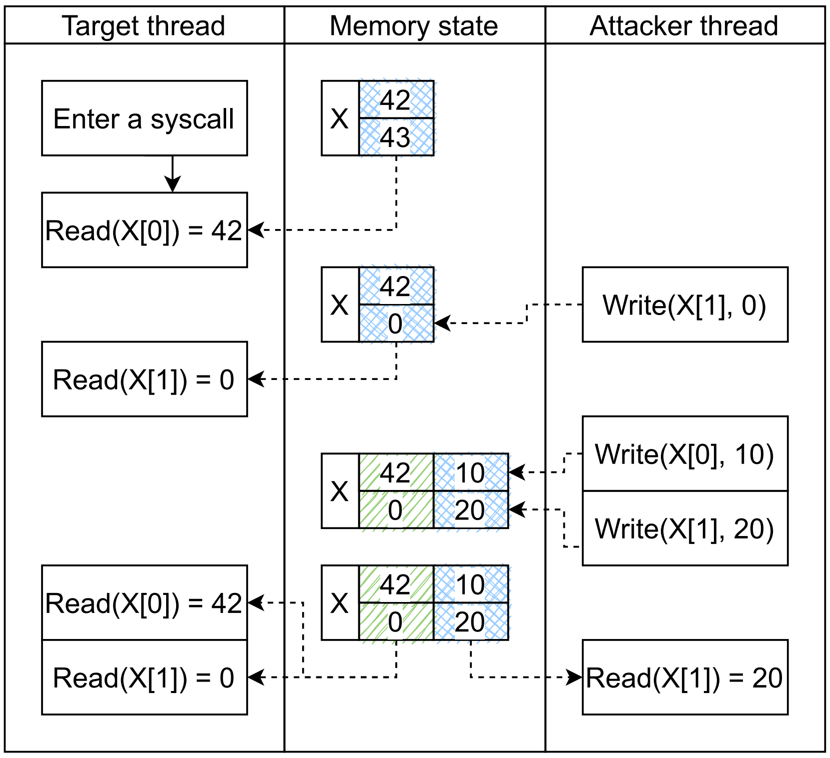


Figure 2.4: Diagram illustrating Midas preventing exploitation of a double fetch of an object `X` spanning two pages.

value of O_1 to change between these two points in time would not work with Midas' protections. Such a hypothetical syscall, requiring concurrent modifications to its arguments, could exist to support some synchronization mechanism similar to a `futex` and can be safely exempt from Midas' invariant. During extensive tests, we did not find any other syscall which exhibits this behavior in the Linux kernel.

Example: Objects spanning multiple pages. Figure 2.4 shows Midas protecting a syscall which has a double fetch for an object spanning multiple pages. Here, the two pages containing the object `X` are accessed as `X[0]` and `X[1]`. The attacker tries to attack the syscall by changing the value of the second page: *i*) between the syscall's first reads of `X[0]` and `X[1]`, and *ii*) between the first and second fetches of X. Midas ensures both fetches return `X=(42,0)`. Critically, any existing TOCTTOU bugs are not triggered since both fetches read the same, possibly invalid, value of the object. Note how the situation is identical to one where the malicious write to `X[1]` happens before the syscall starts.

Preventing deadlocks by design. Midas' design is free of deadlocks, and exempts syscalls which require violation of its invariant from triggering particular state-machine transitions. Userspace reads always succeed, using the latest copy C_L of the accessed page. Writes from userspace and kernel code succeed directly if the page is in State 0 or 2, and trigger a fault otherwise. Handling these faults involves creating a new copy of the page and setting the page writable. Reading from kernel code involves creating a new snapshot and setting the page read-only. None of the aforementioned operations relies on other operations on the same page to complete and all are finite time. None of the operations on a page rely on operations on other pages. A single, per-page lock can serialize operations on that page and assure forward progress.

Detecting double fetches. Midas' state machine for pages enables the precise detection of double fetch bugs, turning it into an effective sanitizer and developer debugging tool in addition to being an efficient mitigation. When a syscall first reads from a user page, it creates a snapshot of that page. On future reads, the snapshot is used in order to maintain the invariant. While reading from a page, implementations must check if a snapshot exists for the syscall: if yes, the snapshot is used for the read, otherwise a new snapshot is created and then used for the read. The existence of a snapshot means the syscall had previously read from this page and had then created this snapshot, implying a double fetch. Unfortunately, this approach is prone to false positives due to false sharing. The two reads might read from the same page, but access entirely disjoint bytes. Midas currently reports double fetches at page granularity. A precise sanitizer could maintain a bitmask of accessed bytes to prune false positives.

Double-fetch exploition using speculative execution. Speculative execution attacks (SEA) on modern processors leverage speculative execution of instructions to bypass permission checks and illegally access and leak information. Assuming the presence of specific microarchitectural optimizations, an attacker may speculatively bypass Midas' protection, and leverage a double-fetch bug to leak information from the kernel. However, an attacker cannot mount an SEA to corrupt kernel state as processors eventually discard the results from incorrect speculation and roll back to the correct execution path where Midas prevents double fetch exploitation. Moreover, the SEA attack relies on load-to-store forwarding between logical threads on a simultaneously multi-threaded (SMT) processor, an optimization described in an academic proposal [54] but not implemented in contemporary commercial processors. Finally, the entire sequence from the attacker's illegal write to the victim syscall leaking sensitive kernel information must complete within the attacking thread's speculative window, typically lasting up to a couple of hundred cycles. Consequently, the victim syscall must use the value from the second read immediately after userspace memory copy. Given the requirements for this attack, including experimental hardware optimizations, strict timing requirements and the need for specific code patterns, we consider this attack theoretical, rather than practical. A hardware fix for this attack scenario is possible, restricting results from potentially faulting writes from being speculatively forwarded to a different thread. When the attacker and victim execute on separate physical cores, this attack is impossible as speculative writes are not forwarded across cores. Next, we describe the steps of this potential attack vector.

For this attack, we assume that the attacker and victim run on two logical hardware threads running on a single physical core on an SMT processor. The steps of the attack are described

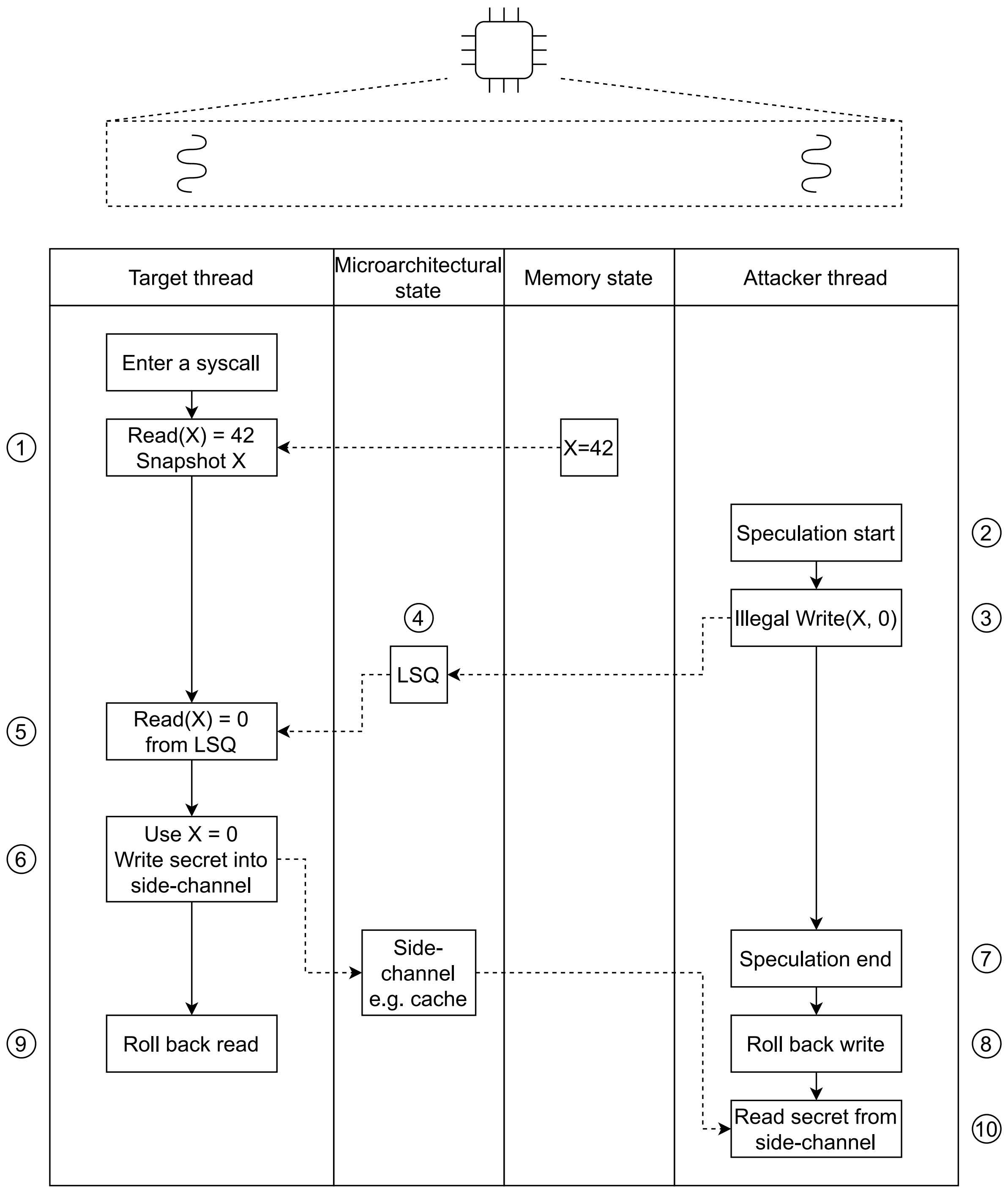


Figure 2.5: Illustration of a theoretical speculative leakage attack against Midas' protection. The attacker uses the core's shared load-store queue (LSQ) to forward the value from the attacker's illegal speculative write to a protected page to the victim syscall's read. The incorrect speculation will be rolled back, but can be used to transiently leak a kernel secret into a side-channel.

below, and illustrated in Figure 2.5.

1. The victim syscall fetches a user object for the first time, marking the corresponding page as protected (State 1),
2. The attacker thread starts speculation behind a long-latency operation (e.g., a last-level cache miss for a data read),
3. The attacker speculatively writes to the protected page,
4. The attacker's write is held in the core's shared load-store queue, and the page fault (for writing to a read-only page) is delayed till the commit phase,
5. The victim syscall initiates the second fetch, and speculatively reads the attacker's write from the shared load-store queue,
6. The result from the speculative read is forwarded to dependent instructions in the victim, causing a side-channel leakage of kernel secrets.
7. The attacker finishes the long-latency instruction and ends speculation,
8. The core rolls back the attacker execution, including the illegal write,
9. The core rolls back the victim's execution based on the illegal read, and
10. Both the attacker and victim continue on an architecturally legal execution path. The attacker reads the victim's secret from a side-channel.

2.6 Midas Implementation

Our Midas prototype implements the state machine described in Section 2.5 on Linux version 5.11, targeting the x86-64 architecture. A page protected by Midas transitions between states on either a kernel read to user memory, or when user or kernel code writes to protected, read-only memory (see Figure 2.2). Midas can be implemented on any operating system kernel that *i*) systematically uses transfer functions for reading from userspace, and *ii*) on any architecture which implements hardware-controlled access control to memory through page tables. The first requirement enables Midas to implement transitions on kernel reads from user memory. The Linux kernel uses the `raw_copy_from_user` interface which we instrument for our prototype. The second requirement causes the hardware to raise a fault on writes to Midas-protected pages, directing execution on the processor to a pre-defined exception handler in the OS. Our prototype instruments Linux' fault handler in the function `handle_pte_fault` to implement the write-triggered transmissions from states 2 and 4. Overall, our prototype adds around $1,100$ lines of code and modifies 17. Our design allows the changes to be mostly limited to the memory subsystem, and in general does not require individual syscalls to be modified. Only one syscall (`clone`) required code modification.

2.6.1 Tracking Page State

Midas needs to track the state for every userspace page, including its snapshot and copy sets. Figure 2.6 shows the data structures used to track a page's state in our prototype. Linux maintains a `struct page` object for every frame of physical memory. We augment `struct page` with a list holding the snapshots for this page, excluding the latest snapshot L. Each snapshot has a pointer to its copy. In the figure, the snapshots S_1 and S_0 share the copy C_0. We are aware of the strong aversion of the Linux kernel developer community towards increasing the size of `struct page`. An alternate implementation can use a hashmap to map from a page's frame number to its snapshots list or reuse existing data members (e.g., `struct list_head lru` which can be used as a generic list by page owners).

Each page table entry for a user page in different address spaces maps the copy C_L, enabling userspace to directly access the page with reads (and writes for writable pages). We use one software-controlled bit (SW3) in the page table entries to track the protection status of the page, and another (SW2)[2] to track the original protections for the page. SW3 is set whenever the page is in one of the two protected states (1 and 3). On a write-triggered protection fault, SW3 can be read to efficiently determine if the fault was due to Midas' protection mechanisms, triggering a state change, or due to buggy software accessing a page with illegal permissions, triggering a signal to the task. Other architectures might have fewer software-usable bits in the page table, and implementations of Midas would require storing the protection status of pages in a separate data structure. The duplication status of the page is implicitly encoded in the snapshots: the page is duplicated when any of its snapshots holds a pointer to a copy other than C_L.

Changing a page's protection state requires PTE updates in all address spaces where the page is mapped. The page's `struct page` structure includes a reverse-map listing for all of these pages, and the corresponding virtual address in each. Our prototype uses this mapping to change PTE permissions across all address spaces for a page.

2.6.2 Kernel Reads from User Memory

Syscalls reading from user memory the first time triggers the allocation of a new snapshot. If the page is not protected (states 1 and 3), the read also triggers a state change where the kernel protects the page in all address space that it is mapped in. Figure 2.7 shows the flowchart of the steps implemented by the kernel function `raw_copy_from_user` for reading from user memory. This function also uses the kernel's `mark_page_accessed` interface to move the page to the "Active" state for the kernel's swapping mechanism, making the page ineligible for being swapped out. We also implement `get_user` and `unsafe_get_user` (used by the kernel for small reads) as a call to

[2]The SW2 bit is alternatively used by the experimental Software Dirty Pages feature of Linux, and cannot be run alongside Midas in our prototype.

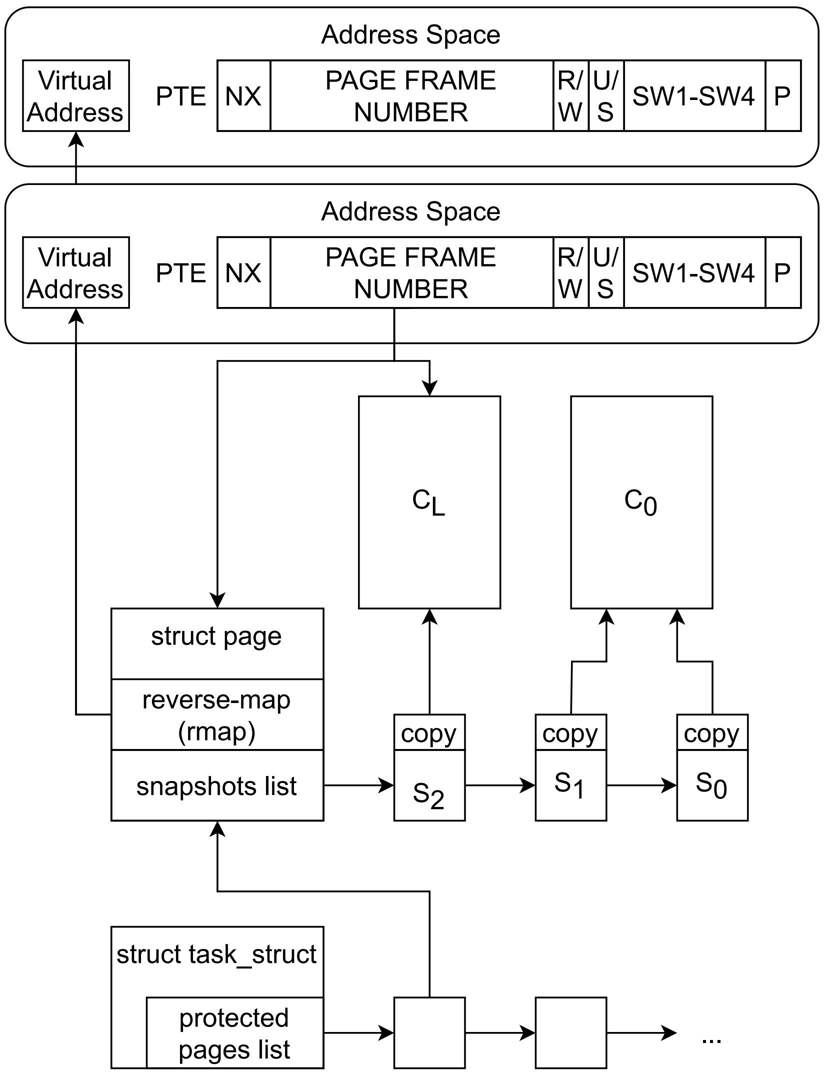


Figure 2.6: Bookkeeping information for a page.

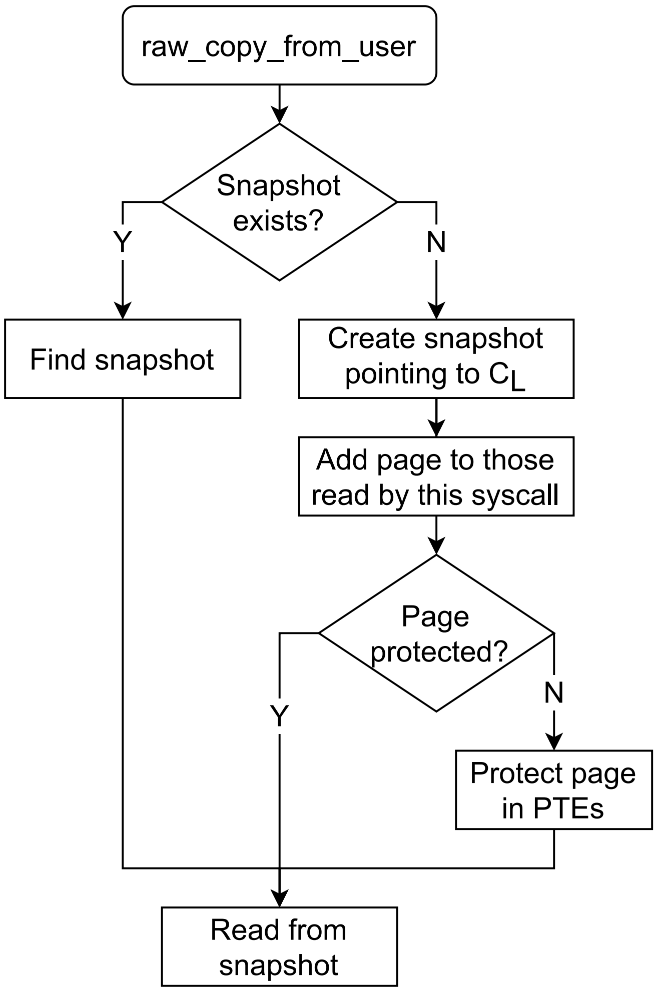


Figure 2.7: Flowchart for a syscall using the transfer function `raw_copy_from_user` for reading from userspace.

`raw_copy_from_user`.

Exemptions. Our prototype Midas kernel exempts a couple of functions from Midas' invariant (in addition to those described in Section 2.5.2), and these functions are therefore not instrumented to follow the aforementioned steps while accessing userspace memory. First, `raw_copy_from_user_inatomic` is a special transfer function used by the kernel to read user memory in special situations such as a kernel oops[3] where the kernel reads user memory to provide a call backtrace. In this severe situation, the kernel's goal is to collect debug information before its imminent termination and no TOCTTOU protection is needed. Second, we also exempt the `write` system call's reads from user memory from instrumentation. The `write` syscall takes three arguments: a file descriptor passed as a register, a pointer to a user buffer and a count of bytes to be written to the file. While the write to the file's pages is sensitive, and Midas takes care to ensure that it follows the page state machine, the read from the userspace buffer is not. The syscall reads from userspace only once, and its data is only used for copying into the file. An attacker who modifies the user buffer concurrently with the syscall only manages to change the contents written to file, which it could have done anyway since it has access to this buffer. A kernel developer can similarly exempt other syscall which they can prove to be secure from double-fetch bugs.

2.6.3 Handling Faults

The memory management unit generates a fault when kernel or user code accesses a page without having the correct permission in the corresponding PTE. Midas marks writable pages read-only to protect them in states 1 and 3, allowing the kernel to detect writes to these pages. A common OS mechanism, copy-on-write (COW) pages, also uses permissions in the PTE to detect when COW pages need to be copied. The PTE's present bit are used to store pointers to file-backed pages when they are swapped to disk. Figure 2.8 shows the flowchart implemented by `handle_pte_fault` to handle faults for userspace addresses.

The page-fault handler first checks if the PTE is NULL, and if so knows that it must allocate a page. If the required page is anonymous, the page can be allocated as usual. Otherwise, for file-backed pages, the handler has to check if the page is already in a protected state (states 1 and 3) by reading the SW3 bit of the PTE and if so, transitions to the required state and allocates a new copy. Pages in states 0 and 2 can be directly mapped, and subsequently accessed.

For non-NULL PTEs, the handler checks if the PTE indicates that the page is present. Non-present pages need to be swapped in. After finding the page, Midas then checks if the page was previously swapped in by any other task and is now in a protected state. For protected pages, Midas implements the required state change based on whether the faulting access was a read or a write.

[3]A kernel oops is triggered when the kernel detects a problem while running which can affect its proper functioning, such as corrupted data structures. A more severe version, a kernel panic, causes the kernel to stop executing, expecting data loss or damage if it does.

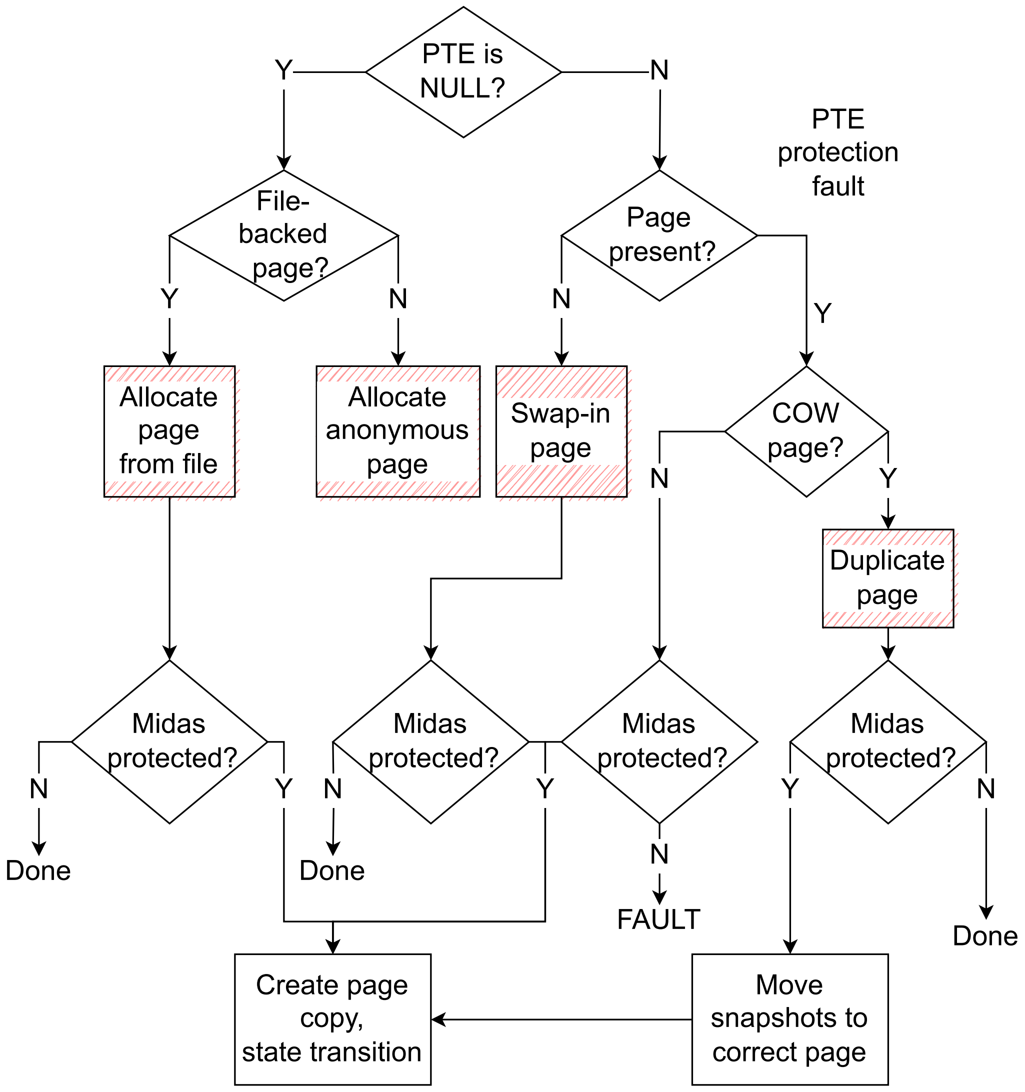


Figure 2.8: Flowchart for handling a page fault. Shaded operations are unmodified.

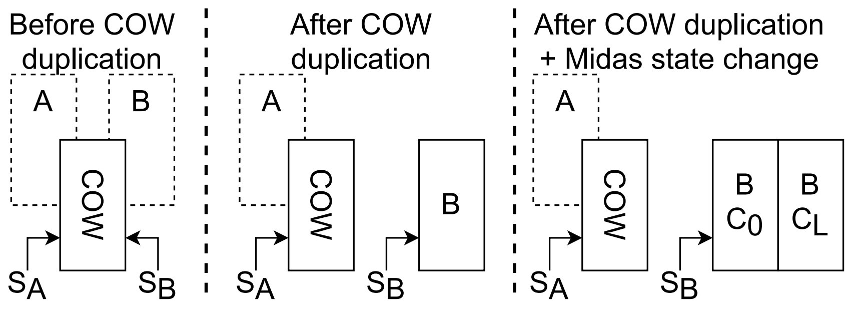


Figure 2.9: Flowchart for handling a page fault to a COW page.

In the remaining case, faults for a present page indicate a permission fault (for example, a write to a read-only page). If the page is not a COW page, the handler then checks if the page is in a protected state by checking the SW3 bit. If the page was protected, a new copy is allocated and the page transitions to the following state. For non-protected pages, however, the fault implies a real access violation, sending a signal to the process.

COW pages represent separate virtual pages from different address spaces mapped to the same physical page. An example of a COW page protected by Midas is shown in Figure 2.9, where logically-separate pages A and B are actually mapped to the COW page. COW pages cannot be in states 2 or 3, since they cannot have multiple Midas copies. COW pages in state 0 can be dealt with by the kernel's standard duplication method (not Midas' duplication). For a COW page in state 1, its list of snapshots can correspond to reads from syscalls for threads in different address spaces. In Figure 2.9, we show snapshots S_A and S_B corresponding to syscalls for threads in different address spaces (containing A and B respectively). These snapshots correspond to different logical pages, but are all squashed into the snapshots list of the single COW page. Therefore, after the kernel duplicates the COW page (new page B created, in Figure 2.9), Midas moves the snapshots for the faulting process (S_B) to the new page. Here, Midas also updates the protected page list in the affected syscalls' `task_structs` so that these structures correctly refer to the new page. Finally, the new page is transitioned to its next state to allow for the write to occur, creating a new copy (C_0) for the snapshot S_B to read from.

We ensure that Midas' modifications to the fault handler correctly handle concurrent faults and do not cause additional nested faults. During concurrent faults for the same page, only one thread changes the page's state whereas the other directly uses the new state. The kernel's split page-table lock is reused to serialize state changes. We also ensure that the only additional accesses to user memory within the handler (used for duplication) happen when the page is assured to be in memory and correctly mapped. All nested faults are therefore caused by existing kernel code and do not interact with Midas' modifications.

2.6.4 Syscall Completion

On syscall completion, Midas cleans up snapshots allocated for the syscall by instrumenting the end of `do_syscall_64`. Midas goes through the list of all the pages for which the executing syscall has a snapshot, and frees those snapshots. For snapshots which were the last to point to a copy, that copy is also freed.

2.6.5 File System Writes

Midas instruments file-system writes to protect the kernel from modifications via kernel mappings. When a `write` syscall writes to a file, it actually writes to copies of pages of the file stored in memory within a page cache. In the spirit of abstraction, the kernel does not directly write to these pages, but calls the relevant file-system (FS) driver instead. The FS driver will access the page using kernel mappings when writing to pages in the page cache. Since Midas only protects userspace mappings for protected pages, writes by FS drivers will not raise a fault. To comprehensively protect the page, any implementation needs to instrument FS drivers' write functions. Fortunately, FS drivers provided with the kernel follow a simple recipe: for pages not in the page cache, the driver executes FS-specific code to read the page into the page cache and then call a generic function (`generic_file_write_iter`) to actually write the data into the page. Instrumenting this generic function, therefore, protects the kernel for a wide range of common file-systems (including ext4, nfs and ntfs). [4] The added instrumentation checks whether the target page is protected, and if so, transitions it to the next state and creates a copy of the page before writing to the latest copy.

Our current prototype does not, however, protect out-of-tree drivers which are not distributed with the kernel if they do not use the `generic_file_write_iter` function. A user with superuser privileges can load a insecure module implementing a FS driver which does not implement Midas checks. A malicious superuser is, however, outside our threat model.

2.6.6 New Mappings to Protected Pages

Our Midas prototype preserves the state machine for user pages across operations which create new mappings to a page to prevent attacks which rely on mappings being created between double fetches. The `mmap` syscall is responsible for creating new virtual memory mappings for processes, and requires instrumentation. When `mmap` is called with the `MAP_POPULATE` flag, or on the first access to the page, the `mm_populate` function is responsible for actually mapping the correct page in the page table. In our prototype, we check if the page being mapped is protected, and if so, correctly protect the new mapping too. Another syscall, `clone`, duplicates a process' address space when called without the `CLONE_VM` flag, creating new mappings to pages. We instrument `clone` to ensure that new mappings for protected pages are also correctly protected.

A class of syscalls can be used to modify user memory using temporary mappings generated by the kernel. Connor et. al. [25] investigated the semantics of the Linux syscall interface in the context of bypassing access control for intra-address space compartmentalization. For example, `process_vm_writev` can be used to modify a process' virtual memory from a separate process

[4] A more comprehensive list of kernel-provided FS drivers protected via `generic_file_write_iter` includes v9fs, ADFS, AFFS, AFS, BFS, CIFS, eCryptfs, extFAT, ext2, F2FS, FAT, FUSE, HFS, HFS+, hostfs, HPFS, JFS, JFFS2, Minix, NILFS2, OMFS, OrangeFS, ramfs, ReiserFS, SystemV, UBIFS, UDF, UFS, VboxSF, shmem.

executed by the same user as the target process. Our Midas prototype currently lacks protection for modifications using these temporary mappings. We leave the manual inspection of each syscall, required to block this attack vector, for future work.

2.6.7 Discussion

Optimizations on capable hardware. To protect a page in an address space, a Midas implementation needs to change the permissions in the page table for that page. Modern CPUs cache virtual memory translations in per-core Translation Lookaside Buffers (TLBs) which need to be (partially) flushed on page-table updates (TLB shootdown). On most CPUs, the core updating permissions will perform a global shootdown to ensure that other TLBs for cores executing in the same address space are also updated. Implemented with inter-processor interrupts, global shootdowns are expensive. In our evaluation, 21% of the runtime of the load generator `bombardier` used for stressing the Nginx server was spent performing TLB shootdowns when running on the Midas kernel.

A more efficient solution would be to have special hardware support for invalidating TLB entries globally, not just on the executing core. The AMD64 architecture manual [6] lists such an instruction (`INVLPGB`), though it is only implemented on a limited selection of server processors only (e.g., 3rd Generation AMD EPYC). The ARM v8-A architecture manual [78] lists similar instructions `TLBI ASIDE1IS` and `TLBI ASIDE1OS` which invalidate all PTEs of a page within a cluster of cores but not for cores in other clusters (Inner Shareable Domain) and cores across clusters (Outer Shareable Domain) respectively. Academic proposals for hardware TLB coherence [108, 132, 150] would also benefit Midas by reducing the overheads for PTE modification.

Alternate architectures [22, 51] with a single, system-wide translation table would also benefit Midas by having a single page table to update instead of multiple page tables for each address space a page is mapped in.

Porting to other OSs. Midas can provide TOCTTOU protection on other operating systems by tracking the states of each page and implementing state transitions as required. OSs track page state in per-page state structures, such as `vm_state_t` for BSD-based OSs (*BSD) such as FreeBSD and XNU. An implementation on these OSs must instrument the read transfer function(`copyin` for *BSD) to transition to states 1 and 3. The OS' fault handler (`vm_fault` on *BSD) will trap on writes to protected pages, and needs to be modified to implement the required page duplication and state change.

The remaining OS modifications for Midas support depends on the OS' features. If an OS allows userspace to map file pages, filesystem code to write to these page needs to be modified. Other syscalls which create/modify mappings to userspace pages will also have to be instrumented to ensure that the new mapping respects the page's state. Such modifications are OS-specific,

```
 1    //First fetch
 2    if(get_user(count,&argp->dest_count))
 3    {...}
 4    //Using first fetch
 5    size = offsetof(..., info[count]);
 6    //Secong fetch
 7    same = memdup_user(argp, size);
 8  + //Added check for bug
 9  + if(same->dest_count != count)
10  +   printk("Bug triggered");
11  + // Fix: copy over original count
12  + same->dest_count = count;
13    //Using second fetch
14    ret = vfs_dedupe_file_range(file,same);
```

Listing 2.1: CVE-2016-6516: Vulnerable double fetch in `ioctl_file_dedupe_range`. Lines in green show the fix and testing code.

making it difficult to recommend a generic methodology.

2.7 Evaluation

In this section, we empirically verify Midas' ability to mitigate a known double fetch vulnerability, and quantify Midas' overhead on workloads with different characteristics including both compute-bound applications which rarely use syscalls and a mix of syscall-heavy applications which heavily rely on the kernel's performance.

2.7.1 Mitigation of CVE-2016-6516

CVE-2016-6516 is a known vulnerability in kernels prior to version 4.7 in a file-system `ioctl`. The vulnerable code is shown in Listing 2.1 and is triggered when the value of the `dest_count` object differs between the two fetches (in lines 2 and 7). `memdup_user` uses the value from the first fetch for allocating a buffer and copying in an array of descriptors from the user in line 7. `memdup_user` also contains the second fetch of `dest_count` which is later used in the function `vfs_dedupe_file_range`. An attacker who increases the size of `dest_count` between the two fetches will cause the kernel to access the copied array out-of-bounds, causing a heap buffer overflow.

For verifying Midas' defense, we introduce a non-faulting assertion check into the function (lines

9–10) and run a known exploit.[5] The condition checks whether the fetched value of the user object (`dest_count`) had changed, indicating a successful attack, and prints a message. Finally, we re-introduce the fix for the bug (line 12), fixing the value of `dest_count` in `same` to that from the first fetch. In this setup, we can detect when the conditions for triggering the bug are met, but also revert to a correct state allowing the kernel to safely continue. The exploit was able to successfully trigger the bug on the baseline kernel every time over 10 runs. With Midas enabled, the exploit was never triggered, i.e., both fetches returned the same value on every call.

2.7.2 Performance evaluation

We evaluate Midas on *i*) microbenchmarks targeting specific common syscalls, *ii*) workloads from two benchmark suites: the NAS Parallel Benchmark (NPB) [10] and select workloads from the Phoronix Test Suite (PTS) [99], and *iii*) the webserver Nginx. NPB includes compute-intensive multiprocessing workloads with a low, but non-negligible syscall rate. NPB therefore demonstrates the ability of Midas to scale to systems where pages are protected across numerous address spaces. PTS includes a variety of benchmarks, both compute bound and I/O bound representative of both desktop and server workloads. PTS includes syscall-heavy applications with varying degrees of parallelism. The Nginx webserver is capable of both high request service rates and scalability with multiple worker processes. We do not include the SPEC CPU2017 benchmarks as they are heavily compute bound and designed to isolate userspace performance without syscalls, and are impervious to kernel performance. SPEC benchmarks would unfairly bias performance in favor of Midas.

The testbench for the evaluation consists of a desktop machine with an 8-core Intel i7-9700 processor and 16GB DRAM running Ubuntu 20.04 LTS. This configuration and CPU is commonly used on desktop machines and workstations. To eliminate the effect of dynamic frequency and voltage scaling (DVFS), we set the processor to run at constant frequency of 3.0GHz which is this model's base frequency. In the *baseline* configuration, we run the testbench with the mainline kernel v5.11 available from Ubuntu's package repository. The *Midas* configuration runs our prototype Midas kernel also based on kernel v5.11. For particular benchmarks, we also run the *Midas+write* configuration which also runs our prototype Midas kernel but instruments all syscalls including `write`.

Microbenchmarks. We test Midas on microbenchmarks from OSBench [95]. The programs use `libc` interfaces such as `fopen`, `pthread_create`, `fork` and `malloc` for creating files, threads, processes and for memory allocation respectively. Table 2.3 lists the prominent syscall usage for these workloads. Figure 2.10 shows Midas' performance (time per operation) on OSBench relative to the baseline kernel, with overheads ranging from zero to 5.3%.

[5]`https://github.com/wpengfei/CVE-2016-6516-exploit/tree/master/Scott%20Bauer`

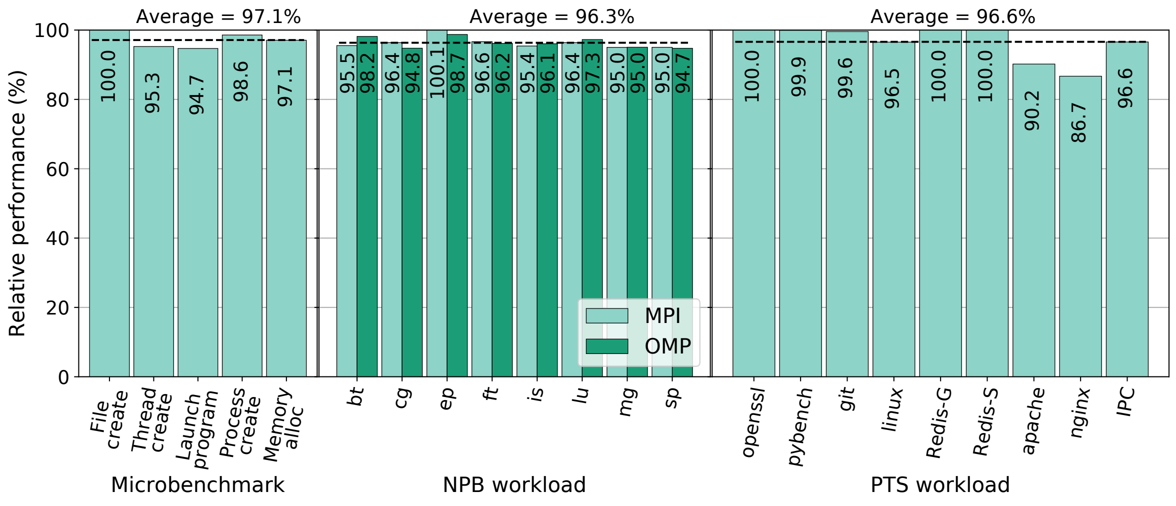


Figure 2.10: Midas' performance on microbenchmarks, NPB and PTS benchmarks relative to the baseline kernel.

Microbenchmark	**Top syscalls used**
File creation	`openat`, `fstat`, `write`, `close`
Thread/Process creation	`mmap`, `clone`, `exit`, `wait`
Program launch	`mmap`, `execvereadlink`, `openat`
Memory allocation	`brk`

Table 2.3: Prominent syscalls used by OSBench microbenchmarks.

NAS Parallel Benchmarks. NAS Parallel Benchmarks (NPB) [10] is a benchmark introduced by NASA. NPB consists of several parallel programs using different communication patterns and is available for two frameworks for parallel programming: OpenMP and MPI. OpenMP [37] is a compiler extension that splits a program's execution to multiple threads. All threads still use the same address space, keeping the overhead minimal. MPI [122] implements parallel execution by launching multiple processes which communicate by message passing. The two technology stacks have different frequency of syscalls due to different communication methods. Communication through kernel syscalls for either stack will incur overhead due to Midas' protection. Additional global TLB shootdowns (for snapshot synchronization) added by Midas will also affect the performance of such parallel benchmarks.

We run NPB benchmarks of class A on our testbench, executing 4 threads or processes in parallel. The benchmarks' runtime varies between 10 seconds and 8 minutes, and are all long enough for the kernel to reach equilibrium. Certain benchmarks require a parallelism number which is a perfect square. On our 8-core CPU, having 4 compute-bound threads/processes instead of 16 allows all threads to run without time sharing. Figure 2.10 shows Midas' performance for both MPI and OpenMP, normalized to the performance of the baseline system with the same parallelization framework. On average, Midas achieved 96.3% of the baseline system's performance on both frameworks. Midas' performance for the `ep` (Embarrassingly Parallel) benchmark is closest to that of the baseline, since it has low communication overheads. Midas shows low overhead (3.7%) for compute-intensive, parallel workloads.

Phoronix Test Suite. The Phoronix Test Suite (PTS) [99] includes a large set (> 500) of open-source benchmarks, of which we have chosen a range of benchmarks suitable for evaluating both desktop and server performance. We bias the selection to benchmarks that require (heavy) kernel activity to test the overhead of Midas' instrumentation. A sole benchmark, OpenSSL, is included to represent single-threaded, compute-bound workloads for which kernel performance is less relevant. The benchmarks are also varied, ranging from single-threaded (Pybench) to multi-threaded, multi-process workloads (Apache). At the extreme, we have an inter-process communication (IPC) benchmark transferring tiny, 128-byte buffers between processes which spends all of its time in syscalls and whose performance is entirely dependent on kernel IPC performance.

We plot Midas' performance relative to the baseline kernel on these benchmarks in Figure 2.10, roughly ordering workloads in increasing order of syscall dependence from left to right. For benchmarks for which PTS reports runtime, we compute the inverse of the runtime as performance. Benchmarks with low syscall frequency such as OpenSSL, Pybench and Git have correspondingly low dependence on kernel performance. Accordingly, these benchmarks see a negligible overhead when running on our prototype kernel. The benchmark titled "Linux" represents compilation of the Linux kernel. While compilation is mostly compute bound, compiling the Linux kernel requires accessing a large number of source files, resulting in the creation of many compiler processes each of which read and create files. Midas experiences a small, but non-negligible overhead of 3.5% on this workload. Redis requires syscalls for receiving and replying to requests, but processes its transaction entirely in-memory. Our evaluation prototype achieves practically identical results as the baseline, highlighting the final prototype's competitive performance. The webservers, Apache and Nginx require network and file-system I/O, and rely heavily on syscall performance. We see that Nginx, which is a higher-performance webserver, sees a larger overhead. IPC, which implements 128 byte transfers between two processes over a TCP connection, is almost entirely bound by kernel performance and sees a performance overhead of 3.4% on Midas.

Our prototype Midas kernel benefits significantly from exempting particular, proven-safe syscalls from instrumentation. While we exclude `write`-like syscalls from Midas because they are not vulnerable to double-fetch bugs, we also evaluated the performance cost of an unoptimized implementation (Midas+write) which also instruments these syscalls. To highlight the worst-case performance of the unoptimized implementation, we evaluate the performance of the IPC benchmark on Midas+write due to its high frequency of `write` syscalls. With Midas+write, the performance of the IPC benchmark falls to 12.6% of the baseline, a further degradation of 84% compared to Midas, showing that developer effort towards properly exempting frequently called *safe* syscalls

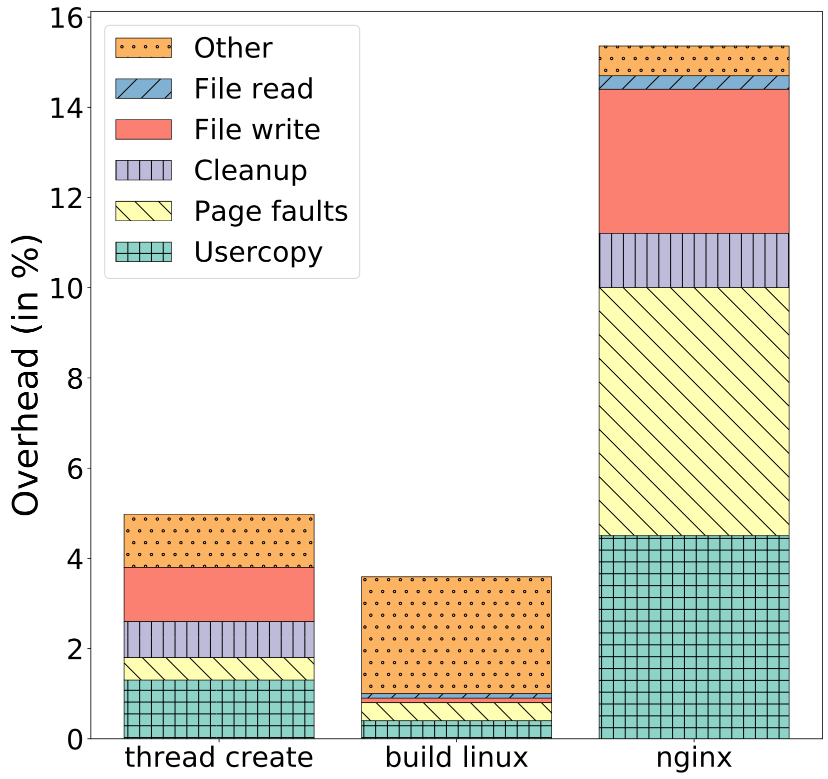


Figure 2.11: Classification of overheads for various benchmarks due to Midas.

from Midas protections is crucial towards for implementations to maintain competitive performance compared to the baseline.

Memory overhead. Our prototype incurs memory overhead due to metadata, tracking page snapshots and copies. At any instant, the memory overhead mainly depends on the number of executing syscalls (limited by the core count) and the number of page copies for these syscalls. On average, for every 1000 syscalls issued by the PTS benchmarks, our prototype created 236 snapshots (32B each) and 54 copies (4KB each). We can see that the occurrence of copies is low, resulting in negligible memory overhead.

2.7.3 Overhead breakdown

In this section, we explore the sources of Midas' overhead by analyzing `perf` traces for three workloads: thread creation from OSBench, linux compilation from PTS, and Nginx. We aim to classify overheads into the various kernel function we instrumented: *i*) user copy in transfer function, *ii*) page duplication on page fault, *iii*) metadata cleanup on syscall end, and *iv*) filesystem operations.

To estimate the time spent in each function, we create FlameGraphs for each workload[46] using samples of processor state, including the call stack, collected over 30 second periods by `perf record`. After identifying one binary for the workload from the FlameGraph, we estimate the overhead for a function as the difference in execution time attributed to that function between the baseline and Midas systems. The total overhead is estimated from the throughput figures obtained from Section 2.7.2.

Figure 2.11 shows the breakdown of overheads for three workloads. As expected, metadata tracking and duplication causes most overheads for the user copy and fault handling functions respectively. The results for the Linux build breakdown differs from the other workloads in the large portion of the unaccounted overhead (labelled "Other"). This anomaly stems for the fact that Linux's compilation runs a large number (1000s) of processes, of which the compiler accounts for less than 50% of the total execution time. The reported breakdown accounts for overheads on the compiler, but not all the other processes.

Both page faults and user copies cause state changes for a page, and thereby change the page's access permissions in the PTE. The resulting TLB flush accounts for 0.3%, 0% and 1.1% overhead for thread creation, compilation, and Nginx respectively. The load generator `bombardier` used for loading Nginx, however, sees a much larger overhead for TLB flushing, accounting for 21% of its execution time.

2.7.4 Case study: Nginx

To better understand Midas' behavior under varying syscall rates and different core configurations, we study Nginx's (version 1.18) throughput while varying payload sizes and different worker counts. Each worker is single threaded and uses one core. The server is loaded with requests from a separate machine running a load generator (`bombardier`) with 100 concurrent connections (chosen to maximize throughput) over a 1Gbps link. The clients send http requests for files ranging between 20B and 10000B.

In Figure 2.12, we plot the request rate and throughput for Nginx servers running with one and eight workers. For all configurations, we can see that the rate of requests served remains almost constant while increasing payload size until the network link reaches saturation. Under a saturated network, the request rates for Midas match that of the baseline system. With a single worker, Midas' overheads cause a consistent 13–14% overhead on the request rate for small packet sizes. However, we see that Midas has practically no overhead when serving requests with 8 workers even when packet sizes are too small to saturate the network link. In this case, both Midas and the baseline system are limited by the scalability of the Linux networking stack.

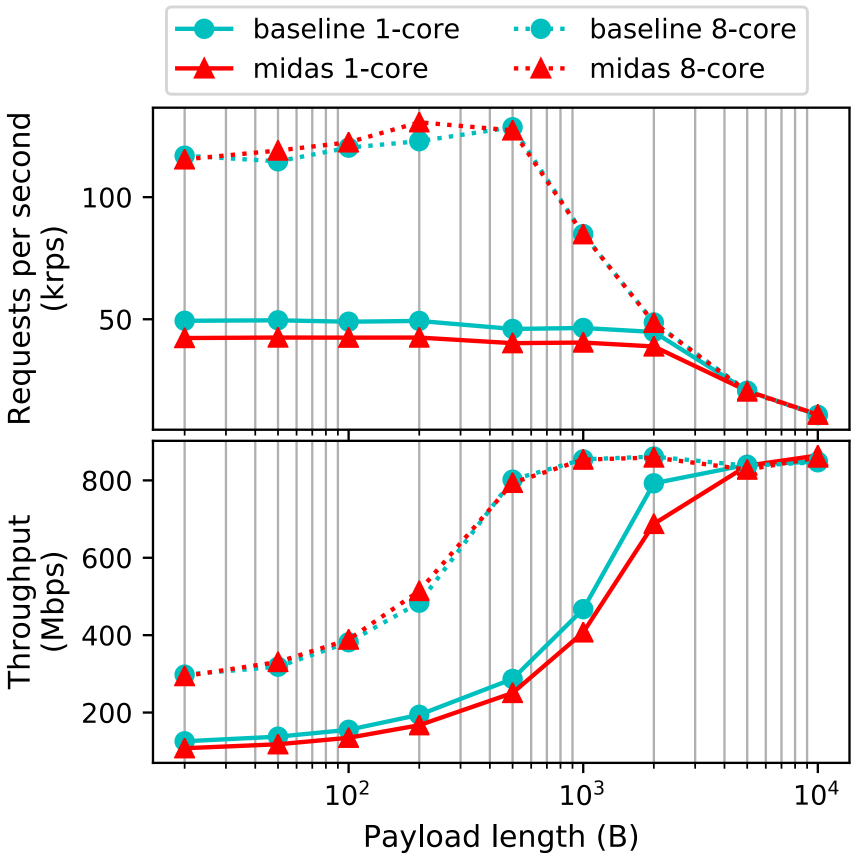


Figure 2.12: Request rates and throughput served by the Nginx server for static pages.

2.8 Related Work

Early work on double-fetch bugs relied primarily on manual code analysis to identify bugs in kernel code [128, 151] or in syscall wrappers [140]. Realizing the limited scalability of this approach, particularly when applied to large codebases such as the Linux kernel, subsequent work focussed on automated techniques based on static or dynamic analysis techniques, and on leveraging hardware features to mitigate such bugs.

Static analysis. Static analysis proposals use code analysis to find and fix double fetch vulnerabilities. DFTinker [80] improves the coverage of pattern matching rules for detecting double fetches in code as initially proposed by Wang et al. [136]. Deadline [147] and DFTracker [138] further generalize the analysis by leveraging symbolic execution.

However, static analysis is severely limited by its requirement for source code, which eliminates possibility of protecting of analyzing binary-only modules for which code is not available. In contrast, Midas can also protect such modules since well-behaving module use the kernel transfer functions to access user memory. Symbolic execution solves the generality problem of pattern-matching approaches but has its limitations (path explosion, function pointers, modelling numerous library functions, etc.). Deadline [147] specifically requires the additional assumption that pointer syscall argument do not alias, an assumption that can wilfully be violated by our adversarial model.

Additionally, the protection allowed by static analysis methods are limited: only the bugs which are detected can be fixed, and static analyses are necessarily incomplete. In contrast, Midas mitigates *all* TOCTTOU vulnerabilities. Further, specific cases of double fetches, such as in syscall wrappers cannot be fixed in code, and require a versioning system such as Midas in order to enable deep argument inspection.

Dynamic Analysis. Dynamic analysis techniques leverage runtime information and values to detect double fetches, and are notable in their ability to find bugs in binaries.

To enable the search for various classes of bugs, Google Project Zero's Bochspwn project [59] introduced a comprehensive emulator for `x86` with callbacks to allow tracing of kernel operations, including memory accesses. When paired with a syscall fuzzer, Bochspwn successfully detected and reported double fetches from these access logs, but suffered from a high rate of false positives. Another major shortcoming of Bochspwn was its low execution throughput of 40-80MIPS which limited its ability to explore code paths. Xenpwn [143] extended Bochspwn's trace-driven approach to fuzz hypervisors for double-fetch bugs. Xenpwn found three double fetches in the Xen hypervisor, but no critical vulnerabilities in KVM.

DECAF [114] inverts Midas' adversarial model, leveraging concurrent access to syscall parameters from userspace to detect kernel accesses via a cache side-channel. DECAF is strongly reliant on CPU-specific behavior, which is sensitive to CPU parameters, subject to changes from generation to generation (or even from core to core) and prone to noise and false sharing. Following the discovery of transient-execution attacks [65], proposals such as InvisiSpec [61, 148] have tried to prevent information leakage via cache side-channels. Future generations might entirely close this channel, or introduce constraints that limits this information flow.

Dynamic analysis techniques can only detect vulnerabilities on executed code paths, and therefore typically rely on a fuzzer to extensively cover kernel code. However, fuzzers are inherently incomplete, limiting the ability of dynamic analysis to find bugs.

Mitigations. Previous attempts [80, 114] to eliminate double fetch vulnerabilities rely on Intel TSX, a hardware transactional memory implementation, to detect malicious writes to data read by the kernel. A defense based on TSX improves upon Midas by reducing the scope for false sharing from a page size to a cache line size. However, TSX suffers from major limitations which restrict its useability for general kernel implementations. Of note, TSX requires that the data working set for the critical section experiences no L1 cache evictions, even due to contention from a simultaneously-multithreaded (hyperthreading) core. Further, TSX is deprecated and limited to processors from a specific manufacturer (Intel), leaving the vast majority of computing devices (mobile, IoT, AMD processors) unprotected.

2.9 Midas Summary

Midas mitigates double-fetch bugs in system calls and protects the operating system interface by enforcing the invariant: *through a syscall's lifetime, every read to a userspace object will return the same value*. Our Midas implementation creates on-demand snapshots and copies of pages that are read and merges any writes through the execution of the system call. Our mitigation protects the core kernel, as well as drivers by carefully instrumenting functions that interact with the process address space. While our implementation focuses on Linux for x86-64, our concept is generic and empowers other kernels to protect themselves against notoriously hard-to-find and easy-to-exploit double fetch bugs.

The performance evaluation of our prototype implementation is promising. Compute-bound benchmarks have negligible overhead and even syscall-intensive benchmarks exhibit low overhead. On one hand, Midas mitigates all double fetch bugs in the kernel and gives developers a tool to locate such bugs. On the other hand, Midas sets the foundation for efficient, stateful system call filtering and validation. We have released the source code of our prototype as open-source at `https://hexhive.epfl.ch/midas/`.

Chapter 3

SecureCells: A Secure Compartmentalized Architecture

Modern programs are monolithic, combining code of varied provenance without isolation, all the while running on network-connected devices. A vulnerability in any component may compromise code and data of all other components. Compartmentalization separates programs into fault domains with limited policy-defined permissions, following the principle of least privilege, preventing arbitrary interactions between components. Unfortunately, existing compartmentalization mechanisms target weak attacker models, incur high overheads, or overfit to specific use cases, precluding their general adoption. The need of the hour is a secure, performant, and flexible mechanism on which developers can reliably implement an arsenal of compartmentalized software.

We present SecureCells, a novel architecture for intra-address space compartmentalization. SecureCells enforces per-Virtual Memory Area (VMA) permissions for secure and scalable access control, and introduces new userspace instructions for secure and fast compartment switching with hardware-enforced call gates and zero-copy permission transfers. SecureCells enables novel software mechanisms for call stack maintenance and register context isolation. In microbenchmarks, SecureCells switches compartments in only 8 cycles on a 5-stage in-order processor, reducing the switching cost by an order of magnitude compared to the state-of-the-art. Consequently, SecureCells helps secure high-performance software such as an in-memory key-value store with negligible overhead of less than 3%.

3.1 Introduction

Modern software systems are complex but monolithic, comprising multiple interacting subsystems, incorporating third-party code like libraries, plugins, or interpreted code, while interacting over untrusted interfaces including networks, shared memory, file systems, or user input. The lack of isolation between the components of a monolithic program allows vulnerabilities to have far-reaching consequences. An attacker who exploits one component can corrupt other parts — for example, a buggy Linux driver can compromise core kernel data structures. The traditional process abstraction for running monolithic software violates the principle of least privilege [111] which requires components to only have access to the data necessary for their operation. Instead, all code running within a process' address space has equal permissions to all data and code regions allowing attackers to subvert pre-defined interfaces between components. For example, calls between components can jump to an arbitrary address bypassing checks on function call arguments.

Intra-address space compartmentalization allows developers to *isolate components* of a program within compartments, only granting each compartment permissions to access their own data. When compromised, a buggy component cannot access another component's data. Conversely, a component is guaranteed integrity of its private data against other corrupted compartments. Compartmentalization is a key defense mechanism that leverages the inherent modularity of code to fortify the cloud [20, 87, 120] and desktop [100] sandboxed environments, programs with third-party libraries [44]and underpins the design of security-focused microkernel operating systems [45, 53, 70, 109]. Compartmentalization constrains the negative effects of the myriad possible faults in software, including memory safety violations and logic errors, to compartment boundaries. For example, the Log4Shell exploit (CVE-2021-44228 [93]) which allowed attackers to exfiltrate secrets and inject arbitrary code in memory-safe programs can be mitigated by isolating the vulnerable Log4j framework in a separate compartment.

The compartmentalization mechanism enforcing the rules of access and communication between the program's components must be secure, performant and flexible. To be secure, the mechanism must enforce policy-dependent restrictions on memory accesses and inter-compartment calls in the face of powerful attackers. Particularly, the mechanism must prevent compromised compartments from escalating their memory access rights or from bypassing inter-compartment call gates. Developers for performance- and security-critical software such as operating systems constantly trade off the benefits of protection mechanisms against their overheads. The mechanism must implement low overhead checks and operations to support fine-grained compartmentalization for such programs. Faster compartment switching, for example, enables developers to refactor programs into smaller compartments with more frequent compartment switches, improving security while maintaining the same performance. Finally, a flexible mechanism which is able to support the wide variety of desired compartmentalization policies will bolster developer adoption.

Existing compartmentalization mechanisms lack one or more desirable features, often trading security for performance, or flexibility for backward compatibility or implementation simplicity. Traditional, process-based isolation [64, 76, 144] only permits costly, microsecond-scale compartment switches. On the other end of the spectrum, protection-key [50] based mechanisms [96, 113, 129] are performant, with nanosecond-scale switches, but fail to deter attackers with code-injection capabilities. Mechanisms co-locating permissions with page-based virtual memory [40, 52, 67, 69, 113, 129] improve compatibility with existing page-tables but inherit the limited reach of modern Translation Lookaside Buffers (TLBs), incurring overheads for programs with large working sets. Finally, other mechanisms [42, 67] target simpler policies, such as protecting a single trusted compartment from an untrusted compartment.

SecureCells achieves the trifecta of secure, flexible, and high-performance compartmentalization by embedding compartmentalization into the architectural virtual memory abstraction. SecureCells proposes *i*) *TCB-maintained VMA-granularity* access control, and *ii*) unprivileged (i.e., userspace) instructions implementing *securely-bounded* compartmentalization primitives, with *iii*) software implementing call gates, call stacks, and context isolation. Related efforts towards languages, compilers and libraries for compartmentalization can extend these benefits to developers by using SecureCells as the underlying isolation mechanism.

For the first pillar, access control, SecureCells introduces the first VMA-granular permissions table consolidating permissions for all compartments into a single data structure designed for efficient permission lookups. In contrast, previous mechanisms use per-compartment permission tables with either duplicate VMA bounds information [144], duplicate per-page permissions within a VMA [113, 129], or both [40, 76]. Deduplicating VMA bounds accelerates compartment switching, eliminating the need to re-load bounds for the target compartment. VMA-scale permission tracking requires smaller VMA-based permission lookaside buffers while also overcoming TLB-reach limits.

For the second pillar, SecureCells accelerates common compartmentalization operations with novel, low-cost unprivileged instructions. Particularly, SecureCells is the first mechanism to allow generic, unprivileged permission transfer from userspace. SecureCells maintains the integrity of permissions by bounding the semantics of untrusted userspace operations to known-safe parameters — the hardware checks the compartment switch instruction to enforce call gates, and permission transfer instructions to prevent privilege escalation.

SecureCells' final pillar leverages the flexibility of software for operations where possible without compromising security or performance (context isolation, call gates and call stack maintenance). SecureCells shows the first software mechanism for restoring register context following a compartment switch, necessary for isolating compartment contexts, without trusting any general-purpose registers.

In this chapter, we:

- define SecureCells' key security and performance objectives and survey how related mechanisms meet these goals,

- propose SecureCells, a novel, secure, flexible, and performant mechanism which introduces compartments into the architectural virtual memory abstraction,
- apply SecureCells to typical application scenarios,
- present a hardware implementation of SecureCells based on the 5-stage in-order RISC-V RocketChip, and
- characterize SecureCells' performance for compartmentalizing micro- and macro-benchmarks.

3.2 Background

Compartmentalization is built on a few basic principles: modularity, least privilege, and defense in depth. Compartmentalization mechanisms implement these principles to provide security to applications through two major properties, isolation of compartments and controlled communication between compartments. In these section, we will explain the principles supporting compartmentalization and see how these principles lead to security properties. The following sections list key objectives for a compartmentalization mechanism in order to guarantee these security properties.

3.2.1 Principles supporting compartmentalization

Abstraction and modularity. Both software and hardware engineering rely heavily on abstraction, and the related principle of modularity to manage the every exploding complexity of computer systems. Logical parts of systems, for example, are separated into modules, each of which hide internal complexity behind well defined interfaces (commonly called application programming interfaces, or APIs). One module can use abstract functionality implemented by a separate module using the module's API without being aware of much of the implementation complexity. Modules are typically structured around a specific functionality, and each module is distributed as a single package. Common examples of modules include libraries for implementing cryptographic processing, threading, mathematical operations, compression and standard language features. Modularity can also be used to separate instances of the same code running on logically separate data. For example different connections being handled by a server can be separated into individual modules, as is done by the Apache webserver. Modularity lends itself to extensibility, allowing the functionality of applications to be gradually extended beyond its original purpose as more and more code modules get added. Modularity has two consequences which greatly benefits compartmentalization efforts. First, developers need to develop well-defined interfaces for modules to enable inter-module communication. These interfaces can be reused as interfaces if modules are isolated within compartments. Second, modules often need to define dependencies or privileges required to function. A compartmentalized system should limit a compartmentalized module's privileges to those necessary for functioning.

Least privilege. Least privilege, defined by Saltzer [111], states that every part of a system should operate using the least set of privileges necessary to perform their function. This principle aims to control the consequences resulting from a malicious or buggy module by limiting the potential interactions each module is allowed to perform. For example, a media decoding module running on a smartphone application might be allowed to access the file system to read media files, but should be denied access to the device's location or to the internet. Further, an application which downloads media from the internet and plays it can be split to isolate the two privileges to separate modules. One module can be allowed to use the internet and save media to a file, and the second module can be allowed to play media from that file and write to the device's screen. A compromised media decoding module, therefore, will be unable to directly leak information to the internet. Least privileges also applies to the ability to call functions in other modules. For example, a media decoder should be allowed to call functions to open and read media files, but not to other file system operations like modifying a file's permissions or owner. The principle of least privilege should also restrict a module's permissions to the temporal period where it requires specific privileges. Privileged programs (like webservers) typically drop elevated privileges when no longer required, following this principle.

Least privileges require applications to be decomposed into modules implementing logically different functionality or working on logically different data. Implementing least privilege for modern applications can leverage the modularity on which systems are already built.

Defense in depth. Defense in depth poses that systems should not rely on a single layer of defense, instead using multiple protective measures for the same objective. Defense in depth mitigates possible shortcomings in a particular defense by relying on one of the other defenses to mitigate attacks. This principle essentially removes the single point of failure for defenses. For systems software, developers should rely on more than one defense mechanism to provide stronger security guarantees. While compartmentalization and memory safety, for example, can both defend against some memory corruption attacks, each of them can protect against attacks which the other cannot. For example, compartmentalization provides no defense against a function overflowing one of its stack objects into another, which is prevented with memory safety. However, memory safety mechanisms might not detect a malicious arbitrary write from one module to another module's data, but this attack should be mitigated through compartmentalization. In general, defense in depth posits that mitigations for memory safety and compartmentalization complement each other, rather than replacing each other.

3.2.2 Compartmentalization properties

A compartmentalized program consists of compartments, each of which is isolated from other compartments but can communicate through well-defined APIs. Isolation and controlled communication aim to limit unintended interactions between compartments, limiting the damage from such interactions.

Isolation. Compartmentalization provides isolation between compartments. Essentially, each compartment has access permitted to a subset of a system's resources, including memory, ability to call other compartments, and OS resources like files, execution threads or I/O. Compartments can have private resources, for which no other compartment has permissions, or resources shared between a subset of compartments. The compartmentalization mechanism then guarantees that no compartment without permission can use these resources. Essentially, a compartment's private resources are isolated from access or use by other compartments.

With isolation, a compartment is protected from data and control flow attacks from misbehaving compartments. With fault isolation or fault handling, an application can also ensure that faults in one compartment only crash that compartment and not the entire application. Hence, isolation is the first key feature of compartmentalization.

Controlled Communication. For functionality, compartments require processing by other compartments. Similar to remote procedure calls, a compartmentalized program requires support for inter-compartment calls. Compartments have interfaces, from which other compartments can request processing. A compartment's interface represents the main attack surface for the compartment, and requires protection. First, compartments must only be allowed to call authorized compartments. Second, inter compartment calls must enter the callee at fixed entry points, allowing call gates to be implemented. Call gates allow a compartment to ensure the validity of arguments and switch context, if required, before processing a request. Therefore, controlled communication is a second key feature of compartmentalized programs.

3.3 Objectives For Architectural Isolation

Compartmentalization mechanisms are characterized by a specific set of objectives, which we introduce in this section. We demonstrate how SecureCells' objectives benefit compartmentalization using two representative programs described below and discuss how alternate goals lead to the differing designs of related mechanisms.

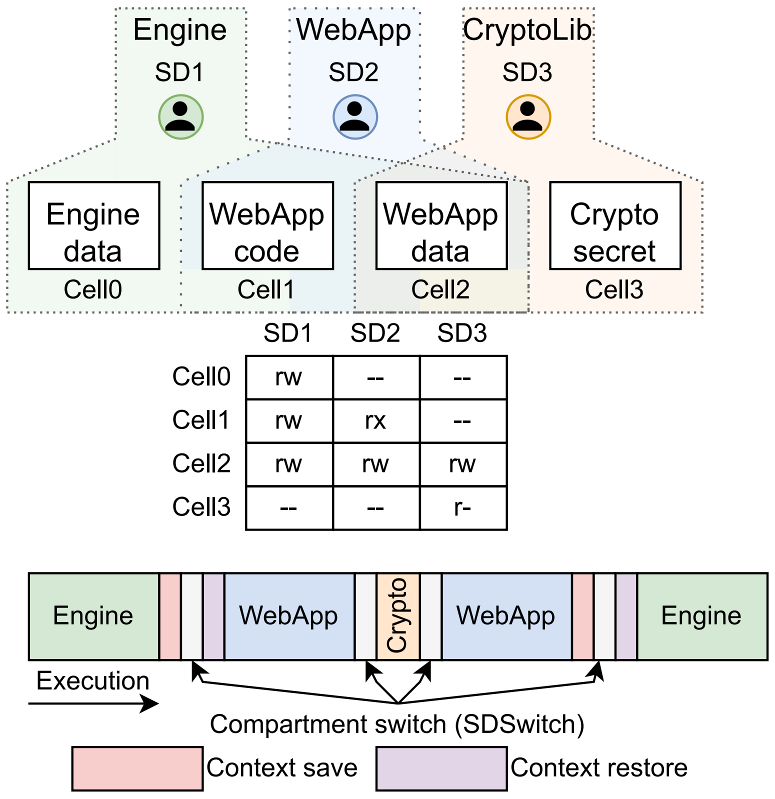
The characteristics of a compartmentalization mechanism determine its applicability. Primarily, the mechanism must be *secure* (**O1**) and enforce the restrictions on data access and communication despite arbitrarily compromised compartments. Second, the mechanism must be *performant* (**O2**), with low overhead for enforcing its checks and restrictions. A performant mechanism allows high-performance software to be compartmentalized without violating performance targets. Finally, a mechanism must be *flexible* (**O3**) in order to support the varying needs of software across security and performance criticality, and their corresponding isolation policies. A flexible mechanism does not make additional assumptions such as a hierarchical trust structure among compartments, and allows data to be shared in an arbitrary fashion. We concretize these objectives, based on insights from existing and candidate compartmentalized programs and related work, in the following subsections. We justify each objectives' importance using the two characteristic programs described below.

Use case: Browser. The first program, a browser (Figure 3.1), consists of a just-in-time (JIT) compilation engine (Engine), a sandboxed web application (WebApp) compiled and executed by the Engine, and a cryptographic library (CryptoLib) storing a secret key for encryption. The compartmentalization policy aims to isolate the Engine's data and CryptoLib's secret from the possibly malicious WebApp. Borrowing the threat model for browsers, we assume that the WebApp can exploit bugs in the Engine's compiler to generate and execute arbitrary code as the WebApp compartment, ultimately aiming to leak the browser's data or the cryptographic secret. The developer aims to prevent unauthorized inter-compartment data accesses by enforcing the per-compartment permissions shown in the figure. To maintain similar performance to the monolithic version, the developer desires minimal overhead from operations added due to compartmentalization: context switching and compartment switching.

Use case: Network Function Virtualization. The second program is a virtual network function pipeline (Figure 3.2) consisting of three stages progressively performing processing steps on a stream of packets. In particular, this pipeline has three compartmentalized stages, implementing the network card driver (Driver) which generates packets, a network address translation (NAT) stage which translates IP addresses in the header based on a translation table, and a firewall stage that implements checks on the packet headers based on a rule table. In this example, we omit further stages for simplicity. Middleboxes in datacenters and the internet [83] commonly contain virtual network functions sharing a buffer pool in uncompartmentalized dataflow pipelines. Translation and rule tables in the NAT and Firewall compartments must be isolated in private regions, protecting them from potential bugs in the Driver compartment that processes input from untrusted traffic from external sources. The programmer requires isolation of network stages for high reliability of the middlebox and low cost for passing packets between stages for line-rate packet processing, enabled by zero-copy packet flow through permission transfer.

3.3.1 Threat Model

Our threat model assumes an attacker who wants to compromise a compartmentalized program with multiple communicating compartments. We assume that the attacker has compromised one or more compartments, and gained the ability to both generate arbitrary code and execute it, but



	SD1	SD2	SD3
Cell0	rw	--	--
Cell1	rw	rx	--
Cell2	rw	rw	rw
Cell3	--	--	r-

Figure 3.1: Browser compartmentalization with three compartments.

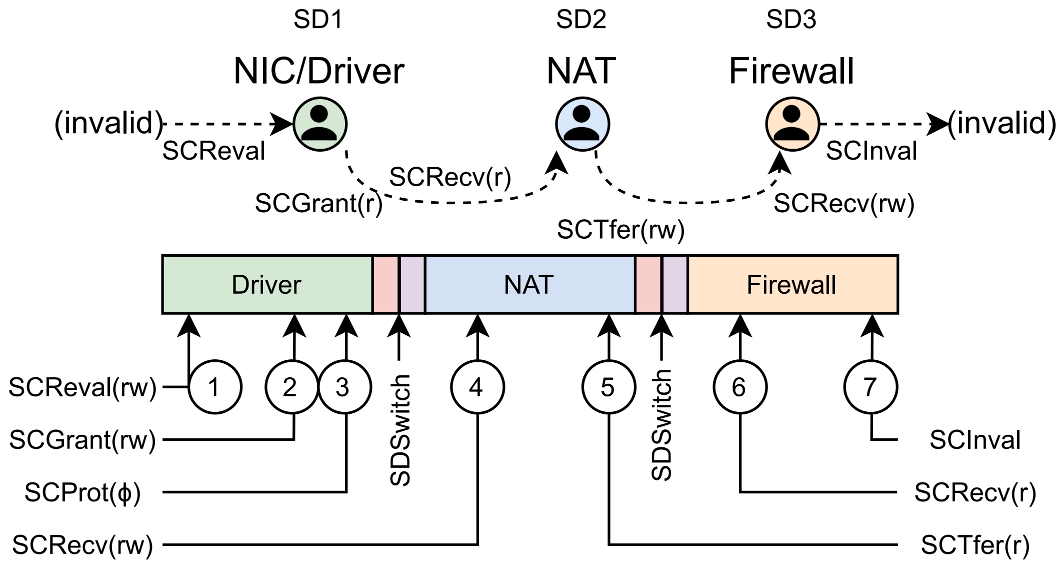


Figure 3.2: Permission transfers for a packet between SecureCells compartments. The figure shows the compartment executing the relevant SecureCells instructions on one core.

is restricted to the compromised compartments. The attacker wishes to compromise confidentiality, integrity, or gain code execution in other compartments. For example, the attacker might try to:

- gain permissions and directly access (load/store) another compartment's private memory,
- inject unsolicited code/data regions in another compartment's memory,
- execute unintended code in another compartment,
- create new compartments, or
- achieve any combination of the above.

The policy used for compartmentalization is assumed to be sound, and the software implementations of the modules comprising compartments are assumed to be free of bugs that can be exploited via only their exposed communication interfaces. For example, we assume that each compartment validates arguments for incoming cross-compartment calls. SecureCells' trusted computing base (TCB) consists of the hardware implementation and the supervisor. Exploitable bugs in the policy or TCB can lead to a compromise irrespective of the compartmentalization mechanism. While speculative side-channel attacks are outside the scope of our threat model, we discuss SecureCells' speculative resiliency in Section 3.7.

3.3.2 Security Objectives

A secure mechanism must enforce restrictions on a compartmentalized program, as described below.

Obj. O1a. Mechanisms must implement *access control*, validating every memory access against the policy. For the browser in Figure 3.1, the table holds policy-defined permissions for each compartment and memory region. Mechanisms must, for example, prevent all accesses by the compromised WebApp from reading the Engine or CryptoLib's private regions as per the policy. Mechanisms must also prevent corruption of policy-defined permissions stored in memory or registers. Intel MPK-based protection [96], for example, loads permissions from a user-controlled register when executing a `wrpkru` instruction, allowing a compartment to corrupt its own permissions.

Obj. O1b. Inter-compartment communication consisting of cross-compartment calls demand *validity checks*. Relevant validity checks include checking that *i*) the entry point is valid, *ii*) the calling compartment is allowed to call the target compartment, *iii*) the return respects the call stack, and *iv*) the passed arguments are valid. Compartment switches from the WebApp to the Engine must use valid entry points which are followed by argument-validating code. Failure to enforce this constraint enables control-flow attacks such as return-oriented programming (ROP). Vanilla Intel MPK-based protection also lacks such entry-point checks to accompany `wrpkru` instructions.

Obj. O1c. *Context isolation* accompanying a cross-compartment call is essential for protecting mutually distrusting compartments. After a cross-compartment call, the callee compartment (for example, the Engine) must be able to fetch its context without trusting the registers which are controlled by the caller (correspondingly, the WebApp), representing an attack vector. The WebApp, for example, could try to switch to the Engine with a malicious stack pointer register, attempting to corrupt the Engine by reading from the wrong stack. CODOMs [131], for example, uses the migrating thread model, and is consequently vulnerable to an attack involving an invalid register state on compartment transitions.

Obj. O1d. Mechanisms that allow untrusted compartments to modify or transfer their permissions must prevent privilege escalation through *TCB-imposed limitations* on these operations. Specifically, compartments should only be allowed to surrender access permissions or transfer existing permissions to other compartments. A stage in the network function pipeline, for example, should not be allowed to grant write permissions for a packet to the next stage if it has read-only permissions. Transferring permissions between compartments must also be mutual, requiring explicit actions from both compartments. One-directional permission transfers studied by Lipton *et. al.* [75] allow compartments to either steal other compartments' permissions (violating confidentiality) or inject illegal data or code into other compartments (violating integrity). Linux, which allows processes to specify their own permissions when `mmap`ing shared regions, violates this objective without syscall mechanisms like SECCOMP.

Obj. O1e. *Temporary exclusive access* to otherwise shared data regions enables compartments to use data regions safely, preventing exploitation of double-fetches. With exclusive access to a packet, the Firewall stage of the network function pipeline can safely validate and use addresses in the packet header in-place (without copying), with the assurance that another corrupt stage cannot concurrently modify the packet. XPC [40] recognizes this objective, allowing exclusive access to a single region tracked by the Relay Segment register.

Obj. O1f. *Auditability*, the ability to easily determine the global access permissions, facilitates auditing compliance to a compartmentalization policy by checking which compartments have access to which memory region. A browser might regularly audit its permissions to ensure that the WebApp has not escalated its privileges. An audit for a mechanism with a centralized permissions store, such as page tables, must only check this store simplifying audits. In contrast, an audit for CHERI [139] requires an expensive, full-memory scan since the set of memory regions accessible to a compartment is the transitive closure of capabilities held in its registers, along with capabilities held in any memory region accessible through these registers.

3.3.3 Performance Objectives

Low-overhead checks and operations allow performance-critical programs to be compartmentalized.

Obj. O2a. *Single-cycle access verification* in the common case is essential for core throughput. While most mechanisms meet this objective in the best (not common) case, page-table based isolation mechanisms suffer from the limited scalability of Translation Lookaside Buffers (TLBs) used to cache permissions. Programs with large datasets can incur high TLB miss rates, with correspondingly high verification latency in the common case due to page-table walks. UNIX process-based protection particularly suffers from this limitation since modern Address Space ID (ASID)-tagged TLBs will effectively contain duplicate entries for a shared page with separate permission for each compartment, effectively dividing an already capacity-limited structure among compartments [55]. This objective implicitly requires the mechanism to support a sufficiently large number of compartments and data regions. A mechanism with small limits, like Intel MPK which is restricted to 16 colors for data regions, will incur overheads from software workarounds required to virtualize the corresponding resource [96].

Obj. O2b. Cross-compartment calls are essential and frequent for communication between fine-grained compartments necessitating *fast compartment switches*. Fine-grained library isolation [44] requires compartment switches accompanying every function call to an untrusted library. A program isolating short-running functions, such as AES encryption using hardware AES-NI extensions, can incur a compartment switch every tens or hundreds of cycles [3]. Specialized hardware instructions accessible from userspace are essential for cheap compartment switches in tens of cycles. Even the fastest supervisor-mediated compartment switch still costs hundreds of cycles [139].

Obj. O2c. *Fast, zero-copy permission transfer* enables programs to efficiently move data between compartments. Data copying for passing large buffers during compartment calls can overwhelm high-performance programs, such as our example network function pipeline. Such applications typically pass packets by reference between unisolated stages profiting from zero-copy. Cheap permission transfers, within ten to hundred cycles, enable such applications to be compartmentalized with performance comparable to the monolithic versions. UNIX process-based permission transfers instead involve microsecond-scale system calls, precluding their use for practical compartmentalization.

3.3.4 Flexibility

A mechanism demands flexibility to be suitable to compartmentalization across a variety of application domains.

Obj. O3a. For flexibility, a mechanism must support *arbitrary sharing of data regions*, requiring independent per-compartment per-region permissions. A private region, for example, should be accessible by only a single compartment. Another shared region might allow read access to one compartment, write access to another, and execute permissions to a third. Mechanisms that target hierarchical security, for example, limit flexibility — the trusted compartment implicitly has permission to access an untrusted compartment's data — and exclude wide applicability. In contrast, even if the WebApp in the browser trusts the Engine, the Engine is denied execute permissions to the WebApp's code. A mechanism must support, but not be exclusive to, specific trust models such as nested compartments.

Obj. O3b. To scale performance overheads with security objectives, we introduce a desirable property, *security-proportionality*. A security-proportional mechanism allows policies to trade-off overheads for security when unnecessary. Despite not trusting the WebApp, transitions from the WebApp to CryptoLib can elide context switching required for register isolation under a specific condition. Verification approaches [23, 66] can be used to prove that a small function in CryptoLib does not leak the key under the assumption that entry points are enforced, and that the function's code overwrites registers used to store the key before returning to the WebApp. By using the cheaper migrating thread model [41], a security-proportional mechanism can reduce overheads where acceptable. Process-based isolation, for example, is not security-proportional since every compartment switch incurs the same non-negotiable overheads (including context switching, page-table switching, or scheduling).

3.3.5 Alternate Visions for Compartmentalization

SecureCells envisions a future where the mechanism supports widespread application compartmentalization efforts, with consequently differing goals and designs compared to related mechanisms. First, some mechanisms only support *custom-tailored use cases* such as differentiating between single trusted-untrusted compartments [52, 62, 67], or a binary classification of data as (in)sensitive [42]. CODOMs [131] link code addresses to compartments, restricting code sharing that is abundant in modern programming. Specialization allows simpler hardware mechanisms, but do not support a broad spectrum of applications. Second, SecureCells does not aim to compartmentalize existing software with zero-modifications. While automated isolation techniques provide a crucial first step towards compartmentalized programs [63, 107, 130], security-critical software requires refactoring to fully realize the benefits of proper compartmentalization. Finally, related works target *compatibility with legacy hardware* or existing or upcoming software/hardware mechanisms and abstractions for isolation. Numerous mechanisms try to compartmentalize using process-based isolation implemented by the OS [40, 64, 76], retrofitting compartmentalization onto an abstraction originally designed for multiprogramming on unicore processors. Others leverage Intel MPK [52, 67, 129] or similar protection-key based mechanisms [113], synergizing with traditional page table-based virtual memory. Targeting immediate adoption, Hodor [52] and LOTRx86 [69] (ab)used existing processor features intended for other purposes to isolate compartments. HAKC [85] leverages state-of-the-art ARM extensions, PAC and MTE, to compartmentalize the Linux kernel, but requires a two-level clustering of closely-connected compartments to overcome MTE's compartment scaling limitations and still incurs a significant performance hit. With the sole exception of Mondrian [144], proposals assume current page-based virtual memory. Meanwhile, trends in applications and memory architectures

	O1						O2			O3	
	a	b	c	d	e	f	a	b	c	a	b
UNIX	✓	✓	✓	N/A		✓				✓	
Mondrian	✓	✓	✓	N/A		✓	✓			✓	
IwC	✓	✓	✓	N/A		✓				✓	
CODOMs	✓	✓		N/A		✓		✓			✓
XPC	✓	✓	✓	✓	✓	✓		✓	~	✓	
MPK						~		✓	✓	✓	✓
ERIM	~	~				~		✓	✓	✓	✓
Donky	~	✓		N/A		✓		✓	✓	✓	
CHERI	✓	✓	✓						✓	✓	
SecureCells	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table 3.1: Comparison of compartmentalization mechanisms based on compliance with the objectives described in Section 3.3. Limited compliance is marked with "~".

have led to a resurgence in range-based translations and protections among academic proposals [12, 51, 60, 98, 149] and commercial processors including AMD's Zen lineup [14]. Table 3.1 summarizes the objectives satisfied by related mechanisms (justification in Appendix B.2). We discuss related mechanisms further in Section 3.6.

Complementary requirements. To satisfy application requirements, programs compartmentalized with SecureCells' mechanisms require complementary properties from other parts of the system including secure compartmentalization policies, a secure and performant supervisor interface, and formal verification of application-level properties aided by programming conventions. For example, supervisors might include a syscall for microsecond-scale compartment creation [76]. Safe calling conventions can provide formal guarantees against inadvertent information leakage from the stack [121]. These investigations are outside the scope of this thesis.

SecureCells overview. SecureCells is a compartmentalization mechanism designed to satisfy the above objectives across a wide array of programs, providing flexibility and performance without compromising on security. SecureCells stores permissions in a centralized permissions table accessible only by the supervisor and hardware. A novel, range-based memory management unit (MMU) and lookaside buffer design (Section 3.4.1) allows single-cycle access control on the fast path satisfying objectives **O1a**, **O1f**, **O2a**, and **O3a**. SecureCells introduces fast, userspace instructions for common compartmentalization operations (see Table 3.2): switching compartments, transferring permissions and validating exclusive access for data regions (Section 3.4.2). These instructions satisfy requirements **O1b**, **O1e**, **O2b**, **O2c**. SecureCells delegates context isolation, call-stack maintenance, and argument validation to software. Section 3.4.3 outlines how software can securely and efficiently implement context isolation and call-stack maintenance. Software implementing these functions satisfy security (**O1b**, **O1c**) and flexibility (**O3a**, **O3b**) objectives.

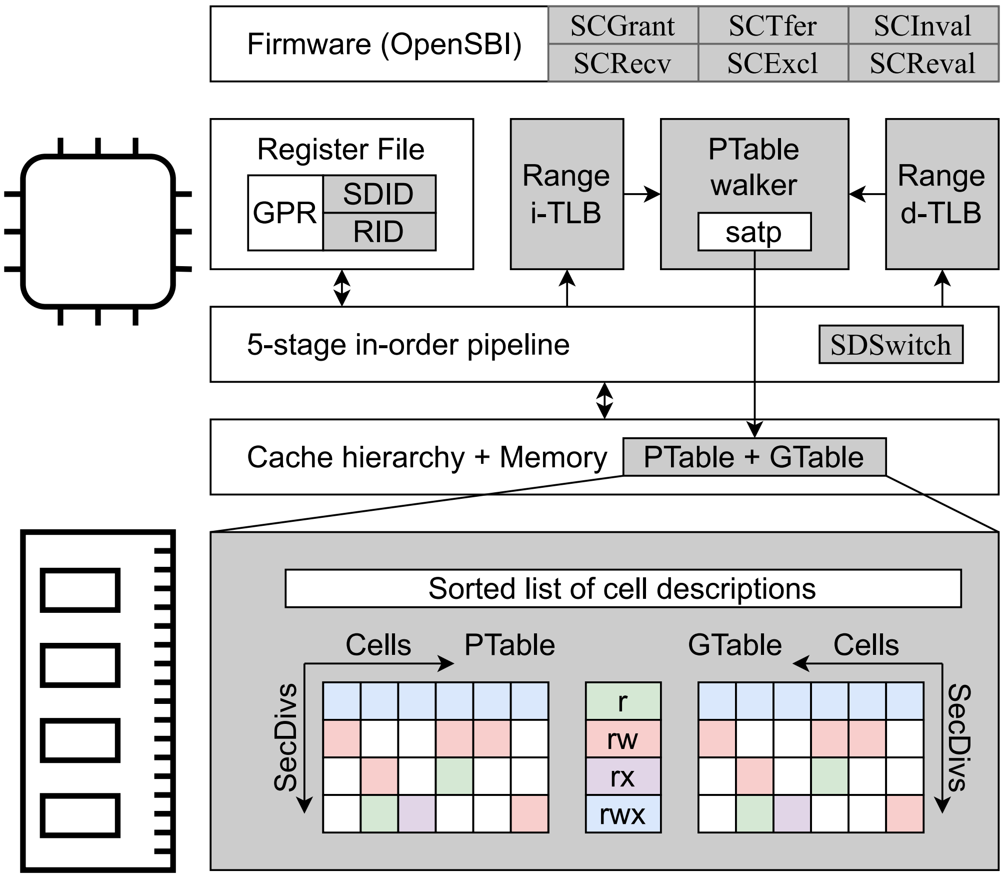


Figure 3.3: SecureCells' architecture, highlighting modified hw/sw in gray.

3.4 SecureCells

SecureCells proposes hardware-software co-design to satisfy the manifold objectives for efficient and secure compartmentalization. The key insight that *compartmentalization operations from untrusted userspace are secure with TCB-maintained permission checks* allows SecureCells to implement compartment switch and permission transfer through trusted hardware-checked userspace instructions which are hundred to thousand times faster compared to traditional supervisor calls. Pragmatically, SecureCells retains software for operations such as context switching which, while common, would not benefit significantly from hardware support. Software implementations of such operations achieve higher flexibility and resilience to implementation errors at negligible or low additional performance cost compared to a hardware implementation. For example, both hardware and software context switching can saturate the L1 data cache bandwidth, achieving similar performance. The second insight is that VMA-based permission tracking eliminates permission duplication inherent in page-table entries for pages within a VMA. SecureCells leverages this insight, eliminating overheads for permission storage (compared to equivalent page tables) and allowing hundred-times smaller core-side lookaside buffers.

SecureCells protects compartments, application-defined mutually untrusting parts of a program, by controlling their access to memory regions. Each compartment is allocated a Security Division (SD) with individual permissions to each VMA-granularity data region (cell). The Browser (Figure 3.1), for example, has three compartments (Engine, WebApp and CryptoLib) allocated SD_1-SD_3, and four cells. SecureCells augments each core with a read-only register ($SDID$) tracking the currently executing compartment. Along with a table for storing permissions (PTable) and a modified MMU for enforcing the permissions, SecureCells implements single-cycle access control. The WebApp SD has executable permissions to one code cell and read-write permission to one data cell. Userspace instructions (see Table 3.2) enable secure compartment switching and permission surrender/transfer. Another per-core read-only register (RID) tracks the caller after a compartment switch, allowing the callee to securely identify its caller. During permissions transfer, a granted permissions table (GTable) tracks outstanding permissions. Further, the design implements context isolation and secure call stacks in software leveraging the above hardware primitives. Figure 3.3 summarizes SecureCells's architecture. The detailed layout of cell descriptors, the PTable, and the GTable are shown in Appendix B.1.

3.4.1 Access control

The Permissions Table (PTable) stores per-cell, per-SD permissions for the compartmentalized program. For each cell, the permissions for each SD are independent and define the degree of sharing for that cell. Compartment-private data stores allow only one SD non-null permissions in this table. Shared cells can be readable, writable, or executable by more than one SD. For a JIT compiler, the cell holding generated data will be writable by the compiler SD, while being executable by the sandboxed code's SD. The PTable is stored in privileged supervisor memory, restricting accesses (including stores) from userspace. Figure 3.1 shows the permissions for the three browser compartments, assigned to separate SDs, to the four data regions, similarly assigned to cells. SecureCells' current PTable design supports a large number of SDs and cells (2^{29} and 2^{32} respectively), vastly exceeding application requirements. A finely-compartmentalized modern browser, for example, will only require a few hundred compartments, isolating each loaded shared library (around 100 on the author's Firefox installation) and per-tab rendering compartments [11].

SecureCells replaces the core's MMU with a PTable walker and range-based lookaside buffers for permissions and translations. The MMU checks access permissions based on the accessed address and the executing SD identified by the core's $SDID$ register. The lookaside buffers track a small number of frequently accessed cells, along with $SDID$-tagged permissions. In the common case, access control verifies permissions from entries in this buffer. When accesses miss in this buffer, the PTable walker reads the required permission from the PTable. The walker first performs a (fast) binary search in a sorted list of permissions to find the correct cell containing the accessed address,

Instruction	Purpose
`SDSwitch`	Switch to another SD
`SCProt`	Change current SD's permission to cell
`SCInval`	Mark a cell invalid
`SCReval`	Revalidate an invalid cell
`SCGrant`	Grant cell permissions to another SD
`SCRecv`	Accept granted cell permissions
`SCTfer`	Grant and drop cell permissions
`SCExcl`	Check for exclusive access to a cell

Table 3.2: Overview of SecureCells' userspace instructions.

then reads the correct permission from the PTable for that cell. The location of the permission in the PTable is found through very simple arithmetic.

SecureCells' PTable layout and MMU design has three key advantages: fast PTable walks, scalability to large data working sets, and low silicon cost. The PTable layout aids fast permission lookups by sorting the cell descriptors, allowing a binary search for the cell descriptor containing an address, and the contiguous layout of the permissions for a particular SD, which improves spatial locality for PTable walks. Range-based lookaside buffers also enable scalability for programs with large datasets, since permissions should be verified against TLB entries in the common case. With growing dataset sizes, traditional processors require larger TLBs in order to track additional page translations and permissions. Importantly, all permissions for pages within a VMA are the same, leading to duplication in TLB entries' permissions. However, the growth of program datasets has exceeded the TLB reach of modern processors, leading to attempts at range-based translations (explicitly managed by the supervisor [149], or implicitly through coalescing [98]). In contrast, as dataset sizes grow, the cell count remains constant and the size of cells increases. Previous work in range-based translation caching [51, 149] have also demonstrated that processors require hundred-times smaller range-based lookaside buffers than in traditional systems, drastically reducing silicon cost. Research proposals [51, 154] have also tackled external fragmentation from range-based translations by introducing a system-wide page table after the last-level cache.

3.4.2 Userspace Instructions

SecureCells introduces 8 new serializing userspace instructions for accelerating common compartmentalization operations (Table 3.2). These instructions, formally defined in Figure 3.4, implement speculation-free compartment switching with checked entry points, permission surrender and transfer for zero-copy dataflow.

The `SDSwitch` instruction targets secure, low-overhead compartment switching within userspace.

SecureCells Program State

```
1: $S$ = Set of $M$ SDs incl. supervisor $SD_{sup}$
2: $C$ = Set of $N$ cells, each valid or invalid
3: Per-core register $SDID$
4: Per-core register $RID$
5: PTable $PT : S \times C \mapsto \mathcal{P}(\{r, w, x\})$
6: GTable $GT : S \times C \mapsto S \times \mathcal{P}(\{r, w, x\})$
```

Instruction 1 SDSwitch($addr$, SD_{tgt})
Switch to SD_{tgt} at instruction pointer $addr$

```
1: $c_i \leftarrow cell(addr)$
2: assert $valid(c_i)$
3: assert instruction at $addr$ is $SDEntry$
4: assert $x \in PT(SD_{tgt}, c_i)$
5: instruction pointer $\leftarrow addr$
6: $RID \leftarrow SDID$
7: $SDID \leftarrow SD_{tgt}$
```

Instruction 2 SCProt($addr$, $perm$)
Restrict rights to $addr$ to $perm$

```
1: $c_i \leftarrow cell(addr)$
2: assert $valid(c_i)$
3: $p_{i,cur} \leftarrow PT(SD_{cur}, c_i)$
4: assert $perm \subseteq p_{i,cur}$
5: $PT(SD_{cur}, c_i) \leftarrow perm$
```

Instruction 3 SCGrant($addr$, SD_{tgt}, $perm$)
Grant SD_{tgt} $perm$ rights to $addr$

```
1: $c_i \leftarrow cell(addr)$
2: assert $valid(c_i) \wedge perm \neq \phi$
3: $p_{i,cur} \leftarrow PT(SD_{cur}, c_i)$
4: assert $perm \subseteq p_{i,cur}$
5: $GT(SD_{cur}, c_i) \leftarrow (SD_{tgt}, p_{tgt})$
```

Instruction 4 SCRecv($addr$, SD_{src}, $perm$)
Accept $perm$ rights to $addr$ from SD_{src}

```
1: $c_i \leftarrow cell(addr)$
2: assert $valid(c_i) \wedge perm \neq \phi$
3: $(SD_{tgt}, gp_{tgt}) \leftarrow GT(SD_{src}, c_i)$
4: $p_{i,cur} \leftarrow PT(SD_{cur}, c_i)$
5: assert $SD_{cur} = SD_{tgt} \wedge perm \subseteq gp_{tgt}$
6: if $perm = gp_{tgt}$ then
7:     $GT(SD_{src}, c_i) \leftarrow (SD_{inv}, \phi)$
8: else
9:     $GT(SD_{src}, c_i) \leftarrow (SD_{tgt}, gp_{tgt} - perm)$
10: end if
11: $PT(SD_{cur}, c_i) \leftarrow perm \cup p_{i,cur}$
```

Instruction 5 SCTfer ($addr$, SD_{tgt}, $perm$)
Transfer all $perm$ rights for $addr$ to SD_{tgt}

```
1: SCGrant($addr$, $SD_{tgt}$, $perm$)
2: SCProtect ($addr$, $\phi$)
```

Instruction 6 SCReval($addr$, $perm$)
Re-validate address $addr$ with $perm$ rights

```
1: $c_i \leftarrow cell(addr)$
2: assert $invalid(c_i) \wedge perm \neq \phi$
3: Validate $c_i$
4: $PT(SD_{cur}, c_i) \leftarrow perm$
```

Instruction 7 SCInval($addr$)
Invalidate $addr$ cell

```
1: $c_i \leftarrow cell(addr)$
2: assert $valid(c_i)$
3: for all $SD_j \in S - \{SD_{sup}, SD_{cur}\}$ do
4:     $p_{i,j} \leftarrow PT(SD_j, c_i)$
5:     $(SD_{tgt}, gp_{tgt}) \leftarrow GT(SD_j, c_i)$
6:     assert $(p_{i,j} = \phi) \wedge (gp_{tgt} = \phi) \wedge (SD_{tgt} = SD_{inv})$
7: end for
8: $PT(SD_{src}, c_i) \leftarrow \phi$
9: $GT(SD_{cur}, c_i) \leftarrow (SD_{inv}, \phi)$
10: Invalidate $c_i$
```

Instruction 8 SCExcl($addr$, $perm$)
Verify exclusive $perm$ rights to $addr$

```
1: $c_i \leftarrow cell(addr)$
2: assert $valid(c_i) \wedge perm \neq \phi$
3: $p_{i,cur} \leftarrow PT(SD_{cur}, c_i)$
4: assert $perm \subseteq p_{i,cur}$
5: $(SD_{tgt}, gp_{tgt}) \leftarrow GT(SD_{cur}, c_i)$
6: if $perm \cap gp_{tgt} \neq \phi$ then
7:     return $False$
8: end if
9: $excl \leftarrow True$
10: for all $SD_j \in S - \{SD_{sup}, SD_{cur}\}$ do
11:     $p_{i,j} \leftarrow PT(SD_j, c_i)$
12:     $(SD_{tgt}, gp_{tgt}) \leftarrow GT(SD_j, c_i)$
13:     if $perm \cap p_{i,j} \neq \phi \vee perm \cap gp_{tgt} \neq \phi$ then
14:         $excl \leftarrow False$
15:     end if
16: end for
17: return $excl$
```

Figure 3.4: SecureCells' state and userspace instructions.

`SDSwitch` resembles function call instructions, with direct and indirect variants, additionally switching the core's $SDID$ register and saving the caller's $SDID$ to RID. Since the $SDID$ register is not writable from userspace, inter-compartment calls must use `SDSwitch`. The cost of executing an `SDSwitch` is essentially the cost of pipeline serialization, plus the negligible cost of updating core registers, making it extremely cheap. For an in-order 5-stage pipeline, an `SDSwitch` instruction completes in around 8 cycles. For an out-of-order processor, pipeline serialization is an essential cost incurred by all related mechanisms to prevent Spectre-like [65] speculative execution attacks, typically requiring 50-100 cycles. For example, serialization dominates ERIM's MPK-based 99-cycle switch latency. Compared to supervisor-controlled compartment switching, `SDSwitch` eliminates the cost of serialization on supervisor entry, context switches on entry and exit, syscall dispatch, scheduling, and accounting costs.

SecureCells requires `SDSwitch` instructions for both forward and backward edges on cross-compartment calls. We show how software can implement cheap, secure call stacks in Section 3.4.3. Programs are also allowed more flexibility, and can implement both remote procedure call (RPC)-like call-and-return (as in Figure 3.1) and circular function call graphs with one-way switches (as in Figure 3.2).

`SDSwitch` instructions impose an additional restriction over function calls in order to enforce call gates — the instruction at the target address must be an executable `SDEntry` instruction for the target SD. This requirement limits the valid entry points for a compartment to the executable `SDEntry` instructions in its code, and is conceptually similar to Intel's CET [56]. A compartment can mark valid entry points with `SDEntry` instructions, and implement call gates directly afterwards. Note that while our attacker can inject arbitrary code into a compromised SD, it cannot write code into any other SD, protecting uncompromised SDs from attack via code injection containing unintended entry points. The only remaining way for an attacker to propagate between compartments is by using valid interfaces. Proper input validation, which is always crucial for compartmentalized programs, protects against this attack vector. In contrast, Intel MPK-based methods allow an attacker to inject and execute a `wrpkru` instruction into a compromised compartment to elevate its privileges to access all memory. Further, the core executing `SDSwitch` updates the RID register with the caller's $SDID$, allowing the callee to identify and validate the caller.

The `SCProt` instruction allows a SD to update its permissions to a cell, with the restriction that the new permissions are a strict subset of existing permissions. Essentially, `SCProt` allows a SD to surrender permissions when no longer needed. This instruction supports a common paradigm in secure software where a program drops permissions as soon as possible.

An `SCGrant`-`SCRecv` instruction pair, executed by separate compartments, allows permissions for a cell to be transmitted between them. When the granting SD executes `SCGrant`, the targeted $SDID$ and permissions are stored in the GTable. A SD is only allowed to grant permissions it already has. Only the targeted SD can later accept these permissions by executing an `SCRecv`, specifying the SD it expects to receive permissions from. Mutual involvement in permission transfers prevents

SDs from "stealing" from or "injecting" into other SDs' permissions, ensuring confidentiality and integrity respectively. Note how this prevents malicious code injection in particular, including where an attacker might try to inject new entry points (`SDEntry` instructions). Recognizing a common software pattern where a SD hands over its permissions to the next stage and drops its own permissions, SecureCells introduces the `SCTfer` instruction. Unlike `SCGrant`, a SD executing `SCTfer` also drops its own permissions to the cell involved. The semantics of `SCTfer` are identical to consecutive `SCGrant` and `SCProt` instructions, but `SCTfer` deduplicates permission checks. All data transfer instructions (`SCProt`, `SCGrant`, `SCTfer`, `SCRecv`) also flush the relevant entry from the MMU's lookaside buffer. The network function is dependent on these instructions to progressively transfer permissions to a packet between stages, as illustrated in Figure 3.2. However, a SD can only have a single outstanding grant for a particular cell. If a SD grants a second set of permissions to a cell before the first set of permissions to the same cell is accepted, the first grant will be overwritten in the GTable.

The `SCInval`-`SCReval` instruction pair allows dataflow pipelines to optimize the end and beginning of dataflow pipelines such as the aforementioned network function pipeline. The pipeline stages progressively drop permissions to the cell holding a packet, and finally wish to drop all permissions after the final stage. However, dataflow pipelines reuse the cells to hold packets, implying that the Driver SD must find a way to regain write permission to the cell to write a new packet's contents to it. While this use case seems to require an illegal privilege escalation *prima facie*, the fact that the end of the pipeline "discarded" the cell holding the packet implies that its contents are trash, and allowing another SD escalated permissions to the cell is secure. To support such usage, SecureCells introduces the concept of validity for a cell. The `SCInval` allows a compartment to explicitly state that a cell holds trash and is available for reuse. On executing this instruction, this cell becomes unavailable for memory accesses and cannot be used by any instruction apart from `SCReval`. The `SCReval` allows any compartment to re-validate and use an invalid cell with arbitrary permissions. SecureCells imposes a key restriction in order to secure cell reuses. A SD can only invalidate a cell when it has exclusive access to it, requiring all other sharers to explicitly drop their permissions to this cell. This restriction ensures that a malicious SD cannot indirectly elevate its privilege to a shared cell by using an `SCInval`-`SCReval` sequence.

Exclusive access to a data region is critical to security and performance, and SecureCells introduces the `SCExcl` instruction for this purpose. Apart from enabling invalidation of a cell, exclusive access is also important for safety in concurrent programming. Concurrent access to data regions enables double fetch vulnerabilities (such as time-of-check-to-time-of-use or TOCTTOU). The `SCExcl` instruction allows a SD to check whether it has exclusive access to a cell. With exclusive access, a SD can skip making private copies of data for double fetches, improving performance. Conversely, when the policy dictates that a SD should have exclusive access to a cell, that SD can verify compliance with the policy using `SCExcl`.

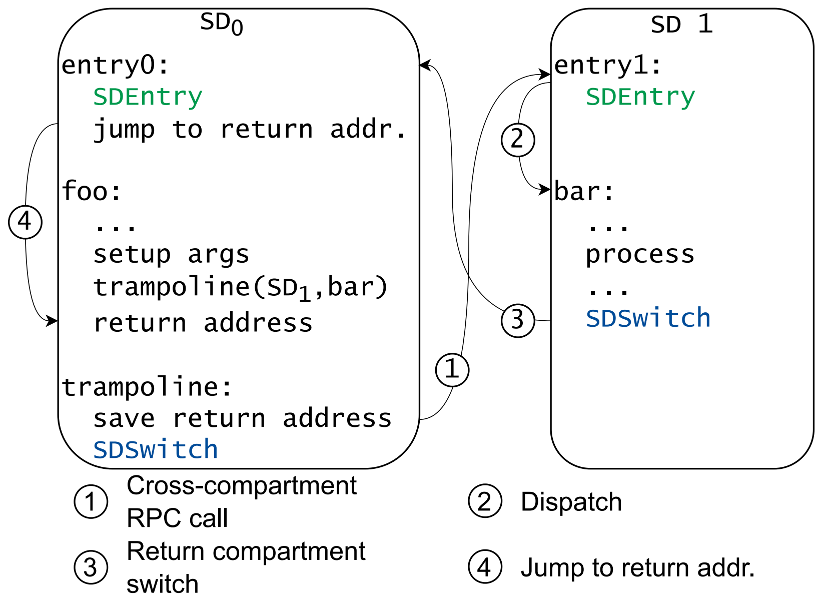


Figure 3.5: Cross-compartment procedure call in SecureCells.

3.4.3 Software Mechanisms

SecureCells delegates certain operations to software: argument validation for call gates, maintaining call stacks, and context switching for register context isolation. Of these, argument validation is arbitrarily variable based on the compartmentalization policy and best left for software checks in hardware-enforced call gates. SDs can determine their caller by reading the RID register, and find arguments in register or memory, and implement software checks as necessary. Software maintained call stacks for inter-compartment calls allow flexibility of calling models, simplifies hardware and remains secure. The software can securely restore with the same performance as hardware, making a hardware implementation unnecessary. In-software operations also improve SecureCells' security-proportionality as these operations can be skipped for lower overheads when safe to do so.

Both forward and return edges on RPC-like cross-compartment calls use `SDSwitch` instructions, as illustrated in Figure 3.5 where function `foo` makes a cross-compartment call to `bar`. Arguments are passed in registers. In this example, the caller uses a trampoline to hide its return address before switching to SD_1 (Step 1) and uses this address on the return path (Step 4). SD_0 is able to hide its calling address from SD_1, just leaking the address of the generic trampoline. Further, following the return switch to its entry point, SD_0 can read RID to verify that the return is indeed from the called SD, not any other. On the other side, the callee (SD_1) can store its caller and switch back to the caller's entry point on the backward edge. If SD_1 contains nested calls to other compartments, it merely needs to remember its caller somewhere in its memory. The dispatch

(Step 2) secures the forward edge to `bar` with call gates. While this example is secure, the flexibility of software allows other calling patterns.

Context isolation requires the caller to save non-argument/return registers to a state store before a `SDSwitch` and restore the same state on the return edge. The second step (context restore) is challenging since it requires the SD to find its state store without trusting any register state, since the register state prior to the `SDSwitch` persists. We propose an array of per-SD private cells as state stores, indexed by $SDID$. The base of this array is easily constructed with instruction pointer-relative instructions following an entry point. Simple arithmetic involving the readable $SDID$ register allows a SD to locate its state store, and consequently restore the register state. The latency of in-software context saving to memory is limited by the core's bandwidth to the L1 cache, the same as for any potential hardware implementation. Therefore, delegating this operation to software has no performance impact. Context switching also switches the stack pointer between per-compartment private stacks.

3.4.4 Implementation

Our implementation of SecureCells augments and modifies the RocketChip [8] core and firmware. An overview of the implementation is shown in Figure 3.3, with additions to the existing processor highlighted in grey. RISC-V provides the ideal, open platform for implementing fully-functional prototypes of experimental architectures. SecureCells permits a range of implementations for single and multi-core processors containing in-order and/or out-of-order cores depending on the application's requirements: from firmware implementations on low-power embedded processors through hardware or microcode implementations on mobile, desktop, and server processors. We discuss the trade-offs in detail in Appendix B.4. To match the RocketChip's simple, in-order pipeline, we implement access control and compartment switching in hardware within the pipeline and emulate the remaining instructions in firmware.

SecureCells provides an alternate virtual memory mode, replacing SV-39 and SV-48. We replace the core's MMU with a range-based TLB and a PTable walker (replacing the traditional page-table walker). We design the layout of the two-dimensional tables (PTable and GTable) in memory to accelerate cell lookups and maximize spatial locality within the cache hierarchy when accessing permissions. We add $SDID$ and RID to the core's Control-Status Registers (CSRs), and implement `SDSwitch` in the core pipeline. The remaining instructions are implemented through hardware-assisted firmware by modifying OpenSBI [123].

The unified PTable-GTable in memory starts with a sorted list of cell descriptions, followed by the permissions held in the PTable, and then the mappings for the GTable. Appendix B.1 shows the layout in detail. Each cell is described by the base and bound virtual addresses, the corresponding physical address base, and a single bit denoting validity. The sorted list of cell descriptions allows the PTable walker to perform a binary search when looking for the cell which contains a particular address, greatly accelerating lookups. The row-major layout of the PTable groups permissions for the same SD in contiguous cache lines, resulting in intra-cache line spatial locality for permission lookups, and synergizing well with next-line prefetchers. As a result, most MMU permission lookups are likely to be served by the L1 cache. The unified PTable-GTable together occupies $\sim 160kB$ to track permissions to 1024 cells with 64 SDs, equal to the memory used by leaf page-table entries to map 80MB of data.

The range-based lookaside buffer holds a few cell descriptions and the corresponding permissions tagged by $SDID$. The implementation of these structures is inspired by recent forays into range-based translation caches [12, 51, 149], primarily aimed at tackling the limited reach of modern page-based translation lookaside buffers (TLBs). Midgard [51] has shown that such lookaside buffers can sufficiently cover the working set of large applications with a few (~ 16) entries.

SecureCells' userspace instructions are implemented through hardware-software co-design. The `SDSwitch` instruction is implemented purely in hardware, and the remaining permission-modifying instructions are emulated through firmware. Additional hardware helpers, designed to aid operations trivially achieved in hardware but costly in software, simplify and accelerate the emulation. We describe these hardware modifications in detail in Section B.6. One notable operation is the lookup of the cell's index in the PTable, which is common for all added instructions. While a binary search in software is expensive, the MMU already holds this information. We add an instruction, only accessible in RISC-V's machine mode and similar to the `AT` instruction in ARMv8-A ISA [5], to directly query the MMU. We envision that higher performance processors with microcode sequencers can implement these instructions in microcode, and leave the investigation of the requirements of such an implementation to future work.

3.5 Evaluation

In this section, we evaluate key metrics for SecureCells' security and performance. First, we show how SecureCells provides security for the Browser described in Section 3.3. Second, we measure the latency of the SecureCells' userspace instructions in microbenchmarks, particularly comparing compartment switching latency to related work. We finally measure SecureCells' performance for two representative workloads highlighting the effect of range-based access control and using userspace instructions for compartment switching and permissions transfer.

Testbench. We ran the security evaluation on a QEMU implementation of SecureCells, which faithfully models its functional behavior, and the performance experiments on our hardware implementation of SecureCells, which uses cycle-accurate Register-Transfer Level (RTL) simulation to accurately measure its timing behavior. The core configuration, described in Table 3.3, resembles ARM's Cortex-A75. Our baseline is an identical core using a traditional page-based MMU and

Component	Configuration	
	Baseline	**SecureCells**
Core	1 × Rocket, 6-stage, in-order	
L1 - D/I TLB	32-entry, fully-assoc.	16(D)/8(I)-entry, fully-assoc.
L2 TLB	1024-entry, 4-way assoc.	32-entry, fully-assoc.
L1 D/I-cache	32KB, 8-way associative	
L2 cache	16MB, 16-way associative	
Main memory	DDR3, 800MHz, 1GB	

Table 3.3: HW configuration of the SecureCells prototype.

	Traditional			**SecureCells**		
	LUTs	**FFs**	**SRAM**	**LUTs**	**FFs**	**SRAM**
L1 ITLB	1915	1886	0	1529	869	0
L1 DTLB	2613	2048	0	1272	1903	0
L2 TLB + PTW	5000	3428	18KiB	3826	4596	0
MMU Total	9528	7362	18KiB	6627	6200	0

Table 3.4: FPGA resource utilization for SecureCells' MMU

TLBs instead of SecureCells. Table 3.4 shows the FPGA resource utilization for both the baseline and SecureCells MMUs. SecureCells' PTable walker contains simpler logic than the baseline, as evidenced by the fewer LUTs required in the design. Additionally, the much smaller range TLB eliminates the 18KiB block SRAM required to store 1,024 entries in the baseline L2 TLB. We run our benchmarks on a seL4 kernel ported to use SecureCells' memory protections. To evaluate realistic workloads on the seL4 kernel, we faithfully ported core functionality of benchmarks, carefully limiting system calls.

3.5.1 Security Evaluation

To evaluate SecureCells' security claims, we test that a properly compartmentalized SecureCells program prevents common attack vectors for monolithic software. We also include an in-depth analysis of SecureCells' instruction semantics afterwards.

Access Control. We evaluate SecureCells' access control on a mock Browser, modeling the example described in Section 3.3. The Browser contains a simple compiler Engine that generates code for sandboxed WebApp applications. The WebApp can allocate arrays, and read/write elements in the array through get/set instructions. We emulate a buggy Engine that generates vulnerable WebApp code lacking bounds checks on array accesses, allowing the WebApp arbitrary reads and writes. With the monolithic Browser, an attacker WebApp could leak/modify the Engine's data as well as that of a second sandboxed WebApp. When compartmentalized with SecureCells with the permissions shown in Figure 3.1, illegal accesses by the attacker WebApp outside its data cell instead raise load/store access faults. SecureCells' access control also prevents arbitrary code injection by the WebApp by preventing the WebApp from writing to either its or the Engine's code regions.

Context Isolation and Call Gates. When uncompartmentalized, the WebApp can modify the Engine's stack enabling control- and data-flow attacks like ROP [118]. Using SecureCells for separation, inter-compartment calls between the Engine and the WebApp are protected through call gates implementing context isolation (Section 3.4.3) including stack switching. SecureCells successfully prevents the WebApp from accessing the Engine's stack.

Formal Description and Security Analysis of SecureCells' Userspace Instructions

We define the semantics of SecureCells' unprivileged instructions in Figure 3.4 and discuss their corresponding security checks below.

`SDSwitch`. This instruction checks that the jump target is valid, and holds an `SDEntry` instruction executable by the target SD. With the precondition that the caller SD does not have writable permission to any cell executable by the target SD, `SDSwitch` guarantees compartment entry at previously defined entry points (helping implement call gates).

`SCProt`. This instruction checks that the target cell is accessible by the SD, and the new permissions are a subset of the existing permissions. After this instruction, the SD is assured to have no more permissions than before.

`SCGrant`, `SCRecv` and `SCTfer`. `SCGrant` checks that the granting SD has permissions to the cell, and that the granted permissions are a subset of its existing permissions. `SCRecv`, in turn, checks that the SD is receiving permissions for a valid cell, that the permissions were previously granted by the specific SD that the receiving SD expects, and that the received permissions are a subset of the permissions granted. `SCTfer` includes the checks of both `SCGrant` and `SCProt`. The granting and receiving SDs must cooperate in order to transfer permissions, and together finish with the same or fewer permissions than they began with.

A correct compartment is defined to not grant or receive any permissions or invalidate cells that it is not required to grant as per a correct compartmentalization policy. Considering a set of compromised attacker SDs and their permissions to cells and assuming that uncompromised compartments are correct, SecureCells guarantees that the attackers can neither gain any new permissions through any sequence of permission transfer instructions nor elevate the permissions of any uncompromised compartment. Using `SCGrant` and `SCRecv` instructions, the compromised compartments can transfer permissions between themselves but those grants cannot include

	Round-trip Cycles			CPU	
	Switch	Context Saving	Total	OoO[1]	Model
lwC	2×6000		12000	✓	SkyLake
seL4	2×514		1027		RocketChip
CHERI	254[2]	129[3]	425		CHERI
ERIM	2×99	Opt[5]	198	✓	Xeon
XPC	82[4]	Opt[5]	82		RocketChip
SecureCells	2×8	Opt[5]	16		RocketChip

[1] Out-of-order CPUs incur higher pipeline serialization costs
[2] In-kernel time
[3] Userspace time (caller, libcheri)
[4] XPC call + return + TLB miss
[5] Optional, software-implemented context switch

Table 3.5: Compartment switching cost of various compartmentalization mechanisms.

permissions which none of the attackers had initially. The only way for the attackers to gain permissions is from an uncompromised SD either granting permissions to a cell or from invalidating a private cell which one of the attackers can validate with `SCReval`. The only way for the attackers to inject permissions is to have an uncompromised SD receive them. By definition, uncompromised compartments will do neither of the above. Once again, we stress on the importance of a correct compartmentalization policy. No mechanism, including SecureCells, can protect against an insecure policy where compartments transfer permissions from/to untrusted compartments without proper validation.

`SCInval`. This instruction allows a SD to invalidate a cell to which it has exclusive access, and to which no outstanding permission grants exist. The first condition can be true for a private region, or for one which other SDs have willingly dropped permissions. Consequently, no other SD will unwittingly lose permissions to the invalidated cell as a consequence of `SCInval`. The second condition provides the assurance that no compartment can regain permissions to the cell without executing `SCReval`.

`SCReval`. This instruction checks that the address corresponds to an existing cell and that it is currently invalid. Due to the initial invalidity of the cell, no SDs could have access to the cell to be revalidated.

`SCExcl`. This instruction does not modify any permissions, only allowing a SD to check if it has exclusive access to a cell to which it already has access to.

Instruction	Trap entry	Dispatch	Emulation	Total Cycles
`SCProt`	79	32	33	144
`SCInval`	79	35	68	182
`SCReval`	79	39	44	162
`SCRecv`	79	54	69	202
`SCGrant`	79	52	63	194
`SCTfer`	79	61	62	202
`SCExcl`	79	57	67	203

Table 3.6: Cycles for emulating SecureCells instructions.

3.5.2 Performance Microbenchmarks

First, we create microbenchmarks to measure the latency of each userspace instruction introduced by SecureCells, of which `SDSwitch` is directly implemented in hardware, and the other instructions are emulated in firmware.

In Table 3.5, we compare SecureCells' compartment switching cost with that of related mechanisms, particularly for a round-trip cross-compartment call. SecureCells' userspace `SDSwitch` enables 8-cycle compartment switches, with optional software context saving costs, which is more than $5\times$ faster than XPC's switch. `SDSwitch`'s latency consists of pipeline serialization (5 cycles), an instruction permission check (2 cycles) and a single cycle for the targeted `SDEntry` instruction. Of course, both XPC and SecureCells would incur higher pipeline serialization costs on an out-of-order core, putting SecureCells on par with, or better than, the MPK-based ERIM. Note that ERIM requires stringent code integrity and control-flow integrity guarantees while SecureCells does not impose any code requirements for its compartmentalization guarantees.

All instructions other than `SDSwitch` and `SDEntry` are emulated by the firmware, and therefore incur the costs of context saving, firmware entry and exit handlers, and dispatch to the correct emulation function. Table 3.6 shows the latency of each instruction, breaking down the cycles spent on each of the above overheads. A microcode implementation of SecureCells would allow the core to use internal registers for storage, eliminating the context switch, and directly lookup the microcode ROM to find the emulation microcode, eliminating dispatch. Consequently, a microcode-based implementation would reduce SecureCells' cost to that of the core emulation code only.

3.5.3 Compartment Switching and Access Control

To evaluate SecureCells' practical performance, we create a simplified benchmark representative of a popular server workload, `memcached`, accurately modelling the workload's memory access patterns across varying dataset sizes. Our benchmark implements the core hashtable-based storage

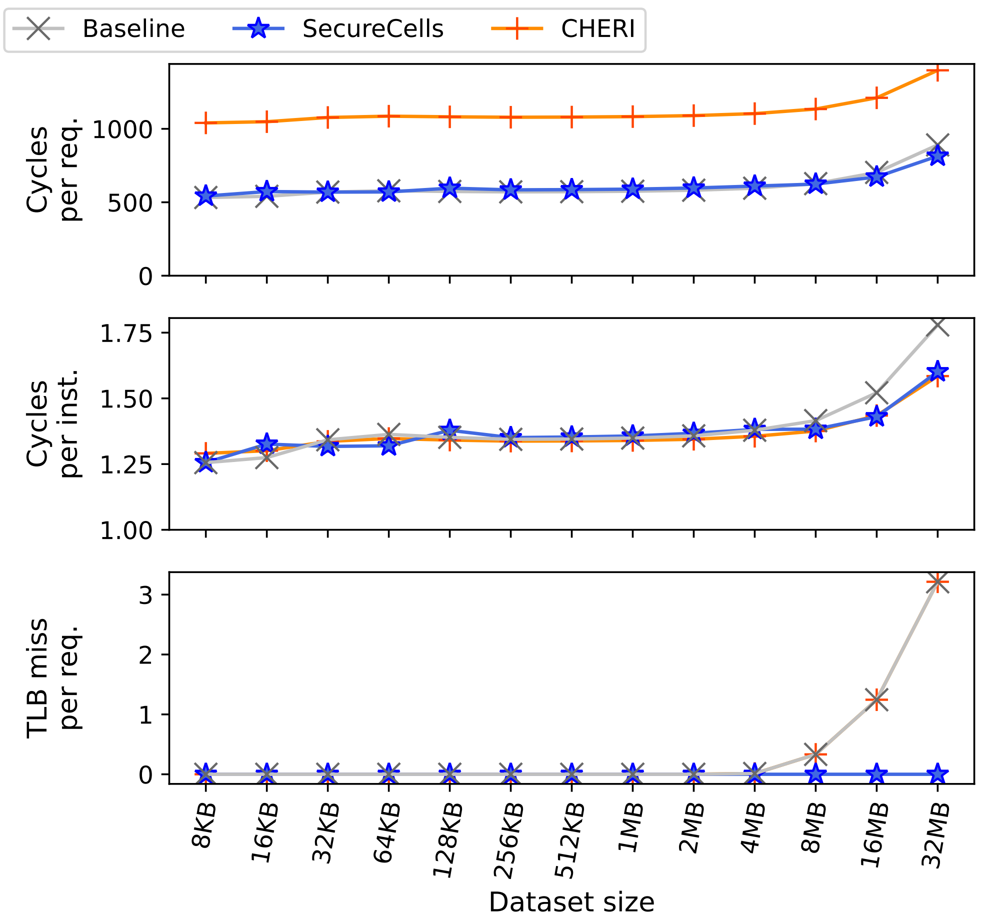


Figure 3.6: Comparison of cycles-per-request, cycles-per-instruction (CPI), and TLB miss rate while executing compartmentalized `memcached` benchmark on SecureCells, compared to the uncompartmentalized version on RocketChip (lower is better).

and the common query path loaded by an in-process load generator function and omits system call-dependent features (networking, dynamic resizing), and the global LRU list. The benchmark isolates the data store from the vulnerable external interface — attackers might send malformed requests to trick the interface into directly accessing the data store — by assigning them to separate SDs. The interface deserializes incoming requests, queries the data store by switching compartments using `SDSwitch`, and serializes the outgoing response. For simplicity, this benchmark utilizes the migrating thread model.

Compartmentalizing the server allows us to measure the overheads of frequent compartment switches, while varying the program's dataset size allows us to compare SecureCells' scalability. We scale the dataset size by sweeping the number of fixed-size (64B) entries stored in the data store, all of which are accessed randomly by the load generator. We compare the compartmentalized server running on SecureCells' implementation to an uncompartmentalized server running on an unmodified RocketChip core by measuring the average count of instructions retired and cycles used to process each request. To compare against another emerging compartmentalization architecture, we also conservatively model CHERI's performance on this benchmark, adding the costs of supervisor-mediated compartment switches with hardware support, as reported in the CHERI paper [139]. We model each compartment switch as 191 instructions requiring 254 cycles, excluding the costs of context switching and ignoring other microarchitectural overheads. In Figure 3.6, we plot the average per-request cycle count and the cycles-per-instruction (CPI) for the server.

SecureCells implements fast compartment switching, and the cost of switching to and from the data store compartment for each request (16 cycles) is minuscule ($<3\%$) compared to the request processing time (minimum 532 cycles). Consequently, SecureCells' performance closely tracks that of the baseline even for small dataset sizes. In contrast, CHERI's compartment switching overwhelms the request processing time, only approaching the baseline's performance for large dataset sizes. While CHERI's performance for compartmentalization compares favorably to that of traditional OS-based isolation techniques, it offers unacceptable overheads for finer, function-granularity compartmentalization (up to 95.5%).

The CPI graph highlights the baseline system's limited TLB reach. As the dataset exceeds the TLB reach of 4MB, the baseline starts to encounter TLB misses on accesses to the data store. Consequently, the baseline CPI starts to degrade compared to SecureCells, and only worsens as the dataset increases past the CPU's last-level cache capacity. In contrast, SecureCells' range-based lookaside buffer comfortably scales to large datasets, allowing the `memcached` server to serve requests 9.3% faster for a 32MB dataset.

3.5.4 Compartmentalized pipeline

To illustrate SecureCells' zero-copy permission transfer performance, we implement the virtual network function pipeline presented in Figure 3.2. The Driver stage generates a "packet" by writing a UDP/IP packet of varying length into a packet buffer, whereas the Firewall and NAT read and modify the IP and UDP headers respectively, but ignore the packet's payload.

Representing the ideal performance target, we include the "uncompartmentalized" configuration that passes the packet by reference, incurring no overheads for data transfer. The second configuration, "compartmentalized-copy", compartmentalizes pipeline stages and uses shared buffers to transfer packets by copy. The third, zero-copy "SecureCells ZC" configuration isolates stages, and uses userspace instructions to transfer access permissions to packets, each of which occupies a different cell. Finally, the "SecureCells ZC-μcode" configuration models the possible performance of a microcode implementation of SecureCells' dataflow instructions by mitigating trapping overheads to the firmware and dispatch. This model is conservative, ignoring possible optimizations from

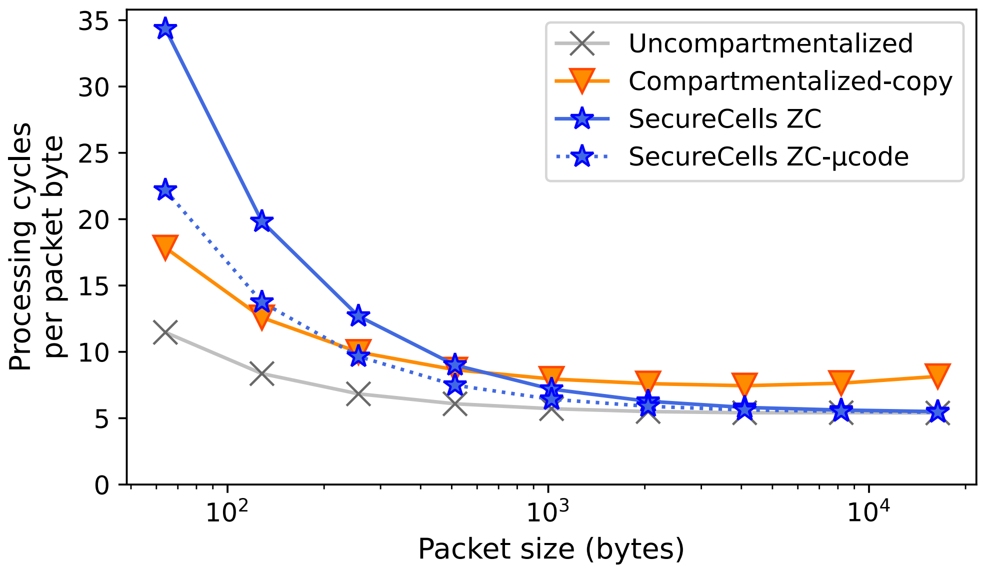


Figure 3.7: Packet processing cycles-per-byte comparison.

parallelizing checks in microcode.

In Figure 3.7, we plot the average number of cycles required by the benchmark to process a byte of a packet as the packet size grows. Fixed costs, such as a function call, compartment switch or permission transfer, have diminishing impacts as the packet size grows. The costs for generating and copying the packet, however, grows linearly with packet size, and add a constant vertical offset in the graph. The "compartmentalized-copy" configuration incurs additional costs over the uncompartmentalized baseline due to compartment switches (4.4% for small packets) and packet copy (51.1%). The "SecureCells ZC" configuration trades-off linearly-growing packet copying costs with fixed-cost permission transfers and (in)validations. While the ~ 250-cycle average latency of SecureCells' permission-modifying instructions causes a massive 199% overhead for the smallest packets, this fixed cost quickly gets amortized for larger packets. Indeed, this configuration overtakes the "compartmentalized-copy" configuration for 600B packets and above, and approaches the performance of the uncompartmentalized configuration (2.0% overhead) for 16kB packets. Finally, the "SecureCells ZC-μcode" configuration highlights SecureCells' performance potential, with (average) 69-cycle operations for transferring permissions which lowers the break-even threshold to 200B packets.

3.6 Related Work

A variety of compartmentalization techniques exist, both in software and leveraging hardware, targeting differing goals and with consequently different designs.

Attacks often target specific, sensitive data for leakage or corruption (e.g., keys or flags). Consequently, various proposals such as IMIX [42], ERIM [129], and MemSentry [67] introduced mechanisms to specifically protect such data from untrusted or unsafe code. However, these mechanisms fail to apply to more generic scenarios, with more than two compartments, per-compartment sensitive or private data, and non-hierarchical trust models. COde-centric memory DOMains [131] proposed an architecture where the instruction pointer identifies the running compartment, in a bid to isolate untrusted libraries. However, this proposal is unable to support the extensive code sharing in modern programs, including shared libraries like `libc`.

Compatibility with existing systems brings immediate security benefits. By mapping the same physical pages across separate per-compartment page tables with different permissions, the existing virtual memory implementation can mimic intra-address space compartmentalization. Typically, such mechanisms require costly supervisor intervention to switch compartments limiting the temporal granularity of compartmentalization. SMV [55] introduced an API for creating intra-address space memory views, but relied on the supervisor for compartment transitions. Light-Weight Contexts (lwC) [76] proposed a new OS abstraction enabling a fast-path in the supervisor for compartment switching, essentially eliminating overheads from unnecessary tasks such as scheduling. lwC successfully reduces the cost of a compartment switch from 4 to 2μs, but remains an order of magnitude away from nanosecond-scale switching. Hodor [52] uses the VMFUNC instruction, introduced for virtual machines, to instead switch page tables in a few hundred cycles, eliminating supervisor overheads but consequently inherits the additional costs of two-dimensional page table walks. LOTRx86 [69] repurposed unused x86 rings to introduce a privileged userspace for storing sensitive data. XPC [40] prioritized software compatibility, choosing to accelerate the remote-procedure call (RPC) interface used for process-based compartmentalization with new userspace instructions. To achieve this goal, XPC cores track a complicated system of metadata across the cores and memory, storing a list of compartments, entry points, valid caller-callee pairs, and a caller stack. XPC is secure, performant, and can allow exclusive access to a single data memory range at almost zero cost. However, XPC requires additional caches for dedicated storage of its metadata, does not allow permissions to be transferred, and requires hardware to implement features cheaply implementable in software (e.g., call stacks), and cannot support non-RPC like compartment switches. With page table-based virtual memory, such proposals all inevitably suffer from the scalability limitations of modern TLBs [12, 98, 149].

Existing architectures have introduced features for intra-address space isolation, e.g., Intel's MPK and ARM's MTE extensions, with fast compartment switching (< 100 cycles) in the common case. These extensions enforce additional permissions, but are insecure under stronger threat models due to designs which prioritize compatibility with existing processors. MPK, for example, is defeated by arbitrary code injection. ERIM [129] requires complicated code scanning to prevent code injection, and Donky [113] requires hardware modifications to introduce an additional trusted privilege level within userspace. Since neither ERIM nor Donky validates code accesses, an attacker targeting cross-compartment code injection need not make the malicious code executable for the target before tricking the target into executing this code. Memory keys also architecturally limit the number of memory regions for which permissions can be efficiently tracked, leaving no room for future microarchitectural advances to improve code performance. These systems also inherit the TLB-reach issues of modern TLBs.

Range-based permission tracking tackling the TLB-reach issue appeared in Mondrian Memory Protection (MMP) [144]. MMP proposed a virtual memory architecture tracking segment-based permissions for compartments within an address space, simulating zero-copy for networking through redundant mappings for packet buffers with different, static permissions. MMP only implements access permission checks in hardware, delegating other operations, including compartment switches, to the supervisor, precluding high-performance applications. MMP also uses different permissions tables for each compartment, reading duplicated range boundaries on each switch.

CHERI refers to hardware-enforced memory capabilities [145], and an eponymous compartmentalization mechanism reusing the same capabilities [139]. The original proposal for memory capabilities offers a practical mechanism to mitigate spatial safety bugs, restricting the ability of pointers to access memory beyond bounds. We recognize that CHERI's capabilities can prevent memory corruption within a compartment, motivating integration with SecureCells to together improve security. CHERI compartmentalization encapsulates capabilities to a compartment's code and data, relying on costly supervisor-mediated compartment switches. CHERI lacks auditability since capabilities are spread throughout memory, and a bug resulting in a capability being leaked cannot be cheaply detected and fixed. CHERI's switching costs are not security-proportional, lacking the ability to skip context switching costs when acceptable. Finally, CHERI's permissions are built on traditional page-based translations, and inherit TLB limitations. Nonetheless, CHERI allows more granular per-object capabilities as compared to SecureCells' per-VMA permissions.

Along with mechanisms, policy research is equally important. Researchers have attempted to formalize a compartment program's guarantees [58], determine the scope of access following permission transfers under the take-grant model [75], automatically infer isolation policies from programs [63, 107, 130], provide hints to programmers on isolation boundaries based on automated analysis [49], and reason about what guarantees remain when one or more compartments are compromised [2].

3.7 Discussion

Legacy program/OS support. SecureCells is compatible with existing pre-emptive operating systems which already separate architecture-specific memory management code. SecureCells also supports page-based memory management (demand paging, swapping) when integrated with upcoming intermediate-address space memory architectures [51, 154]. Since SecureCells preserves the VMA-based view of virtual memory, an OS can present a legacy userspace environment for existing monolithic applications by allocating a single compartment in the PTable. Legacy applications will also benefit from SecureCells' improved TLB-reach with range-based address translations.

Adopting SecureCells. SecureCells faces the daunting task of changes across the software and hardware stack. Nonetheless, library and compiler support for software development can greatly aid developer adoption. We developed a prototype library (`scthreads`) to support compartments with isolated contexts, and envision that most software can be ported through compilation with a SecureCells-compatible C/C++ library. We compartmentalized the example Browser (~1kLoC), initially developed and tested on an `x86` machine, in approximately two additional days. Software such as browsers desiring the full benefits of compartmentalization will still require rewriting (to refactor monolithic code into compartments). SecureCells' userspace instructions map to common compartmentalized applications' operations, evidenced by strong parallels between SecureCells' instructions and APIs in related mechanisms or language-level operations in compartmentalization frameworks (Table 3.7). This mapping will simplify porting existing compartmentalized applications, such as Nginx-lwC [76], to run on SecureCells by replacing existing operations with the SecureCells equivalent (e.g., substitute `SDSwitch` in place of `lwSwitch`). Existing software compartmentalization libraries and compilers [55] can also use SecureCells as a backing mechanism. For example, consider a SecureCells backend for the LitterBox sandbox, used by the compartmentalizing compiler Enclosures [44] to isolate untrusted Go libraries, improving performance and security over the existing Intel VT-x and MPK backends respectively. Enclosure switching (`Prolog` and `Epilog`) map to `SDSwitch` instructions whereas data movement (`transfer`) maps to a `SCTfer-SCRecv` pair.

System call semantics with SecureCells. Recent work [25] has demonstrated that the Linux system call interface can be used to compromise userspace compartmentalization. Modifications of the syscall interface, such as those proposed in Jenny [112], are orthogonal to the compartmentalization mechanism and can be applied to SecureCells. We leave a systematic evaluation of kernel performance and system call semantics with SecureCells to future work.

Advantages for microkernels and system calls. Fast compartmentalization is the key objective for practical microkernel operating systems. By running the OS kernel and drivers in SDs, SecureCells improves over a modern microkernel's switching time by two orders of magnitude. Similarly, userspace programs can benefit from significantly faster system calls if the kernel is assigned a compartment within each program's address space. Essentially, the costly system call

Instruction	Analogous API					
	Linux	MPK	Donky	CHERI	Enclosures	lwC
SDSwitch			dcall	CCall CReturn	Prolog/Epilog	lwSwitch
SCProt	mprotect	mpk_mprotect	dk_mprotect	CAndPerm		
SCInval	munmap	mpk_free	dk_munmap			
SCReval	mmap	mpk_mmap	dk_mmap			
SCGrant SCRecv SCTfer	mmap		dk_domain _assign_key		Transfer	lwOverlay
SCExcl						

Table 3.7: Mapping SecureCells instructions to related mechanisms, libraries and language features.

entrances can be replaced by cheaper `SDSwitch` instructions into the kernel.

Speculative-execution attacks (SEA). We consider the threat of speculative side-channel attacks like Spectre [65] in SecureCells' design, despite omitting such attacks from our attacker model. SecureCells introduces additional mechanisms for changing an executing thread's permissions, through userspace compartment switching and permission transfers. Fault-based attacks like Meltdown [74] must be prevented in implementations by preventing faulting loads from accessing memory or forwarding their data to subsequent instructions [142].

SecureCells does not mitigate existing SEA, but takes care not to introduce vulnerabilities. SecureCells specifies that userspace instructions are serializing, precluding speculative permission changes. An attacker cannot, for example, speculatively switch to a victim SD using an `SDSwitch` following a long-latency branch and read the victim's private data using the victim's permissions. SecureCells' permission transfer instructions are atomic, preventing visibility or exploitation of any intermediate permission state. An attacker SD cannot, for example, drop permissions for a cell using `SCProt` while transferring the same permissions using `SCTfer` in parallel. Our firmware (and future microcode) implementation use load-linked store-conditional atomic operations commonly available across architectures to ensure atomicity.

SecureCells' access control limits the leakage scope of Spectre attacks to a compartment's accessible cells, weakening SEA. SecureCells allows the pipeline to speculate as usual within a compartment's execution, and speculative accesses are also subject to access control by the MMU and cannot illegally access any cell. Access control, therefore, also limits the leakage potential of existing Spectre gadgets. Whereas a Spectre gadget on a traditional processor can address and access any user memory in the process' address space, the same Spectre gadget can only access memory within the compartment's cells. SecureCells also limits the code (speculatively) executable within a compartment, further restricting the availability of Spectre gadgets.

3.8 SecureCells Summary

Compartmentalization requires labor-intensive code restructuring, deterring developers from adopting piecemeal solutions which provide partial protection or which cripple performance. This chapter introduces SecureCells, a secure, flexible and performance-focused compartmentalization architecture to underpin future software compartmentalization efforts. Further work is required, for scaling our FPGA prototype to an out-of-order, multicore processor, investigating implementations of higher-level abstractions on SecureCells' mechanisms, developing software conventions to develop correctly compartmentalized programs for SecureCells, and to improve OS support for the architecture.

Nevertheless, SecureCells enables practical, effective, and efficient compartmentalization by tackling the core architectural requirements for a mechanism. SecureCells strictly enforces access controls and protects permissions from corruption, while supporting secure 8-cycle compartment switching. SecureCells constrains inter-compartment control flow to respect call gates, protecting these interfaces from fault propagation. SecureCells is also an enabler for data processing pipelines with userspace zero-copy data transfers. SecureCells remains flexible, eschewing policy-specific specializations. We have published the SecureCells prototype, benchmarks and supporting infrastructure at `https://hexhive.epfl.ch/securecells`.

Chapter 4

SoK: Classifying Compartmentalization Mechanisms

Compartmentalization is the next major frontier of defense for modern applications, providing security based on the principles of least privilege and defense in depth in the modern era of extensive code sharing and internet-connected devices. Compartmentalized applications rely on isolation between compartments, enforced by mechanisms through checks restricting a compartment's operations to those allowed by a policy. These checks aim to limit each compartment's privileges and ensure that interactions between compartments conform to the developer's intentions. Practically, the assorted commercial and research offerings for compartmentalization mechanisms vary greatly in their design goals and offer support for restricting different classes of privileges, and to different degrees. This fragmentation hinders the development of hardware based on a common mechanism for greater adoption within the application ecosystem.

This chapter classifies the checks mitigating various attack vectors for inter-compartment attacks into three major heads: restrictions on executable operations, their operands, and the resources accessed by these operations. We systematically score 12 mechanisms, chosen from commercially available and state-of-the-art proposals, on axes measuring how comprehensively they implement checks for each category of restrictions listed above, how fast these checks are, and the limitations of these checks. From this comparison, we highlight the commonalities and difference between the surveyed mechanisms, and underscore common weaknesses to highlight opportunities for future mechanisms to improve upon.

4.1 Introduction

Modern software architecture, where a majority of applications are monolithic, stands in stark contrast to the fragmented nature of software development. Applications run code from diverse authors and varied sources, trusted to different degrees, with little to no expression of these trust relations. Within a process, all parts of an application execute with the same privileges, without isolation between parts. Applications often comprise dependencies and libraries automatically pulled from software repositories, inheriting the bugs in this software. Security-critical software can be dynamically extended with third-party modules, perhaps written by developers lacking the same security consciousness as the original application's developers. Further, the code churn over time due to updates of the main application as well as all of its dependencies makes a comprehensive analysis to eliminate all bugs or vulnerabilities infeasible [68, 156]. Attackers can exploit a bug or malicious code in any component to compromise the entire application. Working under the realistic assumption that all code contains exploitable vulnerabilities, compartmentalization is a principled approach to mitigating the propagation of faults or exploits between components of applications.

Compartmentalization relies on isolating components of applications within separate compartments, each with access to the minimal set of privileges to function, preventing malicious or otherwise unintended interactions between components. When an attacker compromises any component, exploiting a bug in its code and bypassing existing mitigations (for memory safety, for example), additional restrictions on the compartment's privileges hinders the attacker by limiting the set of malicious operations they can execute to propagate and compromise other compartments. Compartmentalization is based on principles tracing their heritage to security research in the 1970s: least privilege [110] and defense in depth.

Compartmentalization fundamentally changes software architecture, requiring developers to write applications with well-defined components each of which will be isolated within a separate compartment with restrictions enforced on each compartment's privileges. The definition of compartments and allocation of application functionality to compartments is specified by a compartmentalization policy. A compartmentalization mechanism is tasked with upholding the isolation properties and restrictions defined by the policy.

Identifying the need for compartmentalization, academic and industrial projects have tried to introduce mechanisms enforcing compartment-wise checks on operations and resource access. Proposed mechanisms involve changes to the software [76], hardware [15, 40, 96, 113, 129, 131] or both [139]. While these mechanisms are built for compartmentalization, each one targets a different set of design requirements leading to designs with a significant range of security and performance characteristics and practicality. More backward-compatible designs [76, 96, 129] either suffer from high overheads or provide limited security at low overheads. With clean-slate designs [15, 139, 153], proposals have been able to mitigate these shortcomings by introducing fundamental changes to hardware/software interfaces. However, there is a significant adoption cost for new interfaces, particularly for hardware, and these proposals are often more exploratory than practical. Consequently, there is no accepted standard mechanism to support compartmentalization, hindering the rate of application adoption. A comprehensive systematization of compartmentalization mechanisms is essential to understand the strengths and weaknesses of existing mechanisms, and to identify the key remaining common shortcomings.

In this chapter, we first describe a generic compartmentalized application and list the attacker model mechanisms attempt to defend against (Section 4.2). Next, we systematically list the avenues for cross-compartment attacks, and categorize the privileges for operations involved in these attacks (see Figure 4.2). We then propose a categorization to classify the restrictions required to mitigate these attacks at different points, and define security metrics for mechanisms based on how many restrictions mechanisms implement and how strong these restrictions are (Section 4.3). We also define a categorization for the performance properties of mechanisms for implementing the checks required to enforce these restrictions (Section 4.4). Finally, we compare 12 mechanisms to find which attacker models they can protect against, and which restrictions they support to isolate compartments (Section 4.5). To compare mechanisms quantitatively, we create a scale for scoring mechanisms' security and practicality, based on their security properties and their practical limitations including performance overheads. The scores for each mechanism are illustrated in Figure 4.3 (security) and Figure 4.4 (practicality). We find that

- supervisor abstractions (processes and newer proposals) are secure but have higher performance overheads,
- newer mechanisms are pushing the performance limits of mechanisms,
- clean-slate designs improve the performance-security pareto frontier leveraging new interfaces and improved hardware capabilities, and
- there are significant areas of missing protections across all surveyed mechanisms.

Our systematization allows us to gain insights on the range of mechanisms, and to highlight improvement opportunities for future mechanisms in Section 4.6.1.

4.2 Compartmentalization: A Deeper Dive

To understand the security properties of compartmentalization mechanisms, we need to first understand how compartmentalization expresses trust relationships within applications, and how privilege restrictions help enforce isolation as demanded by trust relationships. Therefore, this section illustrates a practical application as an example to walk readers through the steps of compartmentalization.

Compartmentalization involves composing applications from communicating components with well-defined roles, where each component is restricted to exclusively the privileges required to implement their functionality. To build a compartmentalized application, a developer must identify separate components in their application, define their privileges, and specify interfaces allowing communication between compartments respecting the trust relationships between compartments. The modern Chromium browser, for example, is compartmentalized into a browser kernel and a rendering engine that communicate over inter-process communication (IPC) calls [11]. Only the kernel compartment can interact with local system resources like files. To protect the local system, the kernel compartment distrusts the rendering engine compartment which handles sandboxing untrusted, and possibly malicious, JavaScript code from websites. Practically, building a compartmentalized application may rely on a varying mix of manual programming effort and automated toolchains based on program analysis or developer annotations.

The security of compartmentalized applications relies on limiting how compartments interact, reducing the surface area available for attacks between application components. Whereas components of a typical uncompartmentalized, or monolithic, application can directly interact and affect each other, compartments are restricted to only interact via defined and secured interfaces. Monolithic browsers suffer from severe security issues since code compiled from untrusted JavaScript downloaded from a website shares the process' address space with key trusted compiler components with the same access permissions. Consequently, if a bug in the browser resulted in an arbitrary read/write primitive for a malicious JavaScript application, that application could directly leak/-modify *any* browser data, or execute any code including system calls [101, 102]. Modern browsers therefore compartmentalize the JavaScript engine and all JavaScript code. Compartmentalized Chromium removes this interaction by using separate processes, and hence separate memory address spaces, for the kernel and renderer compartments. An attacker must instead use the more limited IPC interface to exploit separate bugs in the kernel component to corrupt its memory or execute malicious system calls, making similar exploits harder.

Compartmentalization mechanisms empower the idea of least privilege by enforcing restrictions on which operations each compartment can execute. Commonly, existing and proposed mechanisms restrict what memory compartments may access, what system calls they may use, and what supervisor resources compartments may access through those system calls. For memory access restrictions, mechanisms support either separate per-compartment memory address spaces or per-compartment permissions for intra-address space compartmentalization. Mechanisms also rely on incarnations of system call filtering to restrict system calls and access to supervisor resources. The limitations listed above are not comprehensive: applications may also require other restrictions such as the ability to freeze privileges (essentially limiting temporal permission changes), limits on inter-compartment control flow, the ability for compartments to influence other compartments' register contexts, and more. We will later explore useful restrictions comprehensively in Section 4.3.

In reality, mechanisms offer varying support for restrictions and their security properties differ

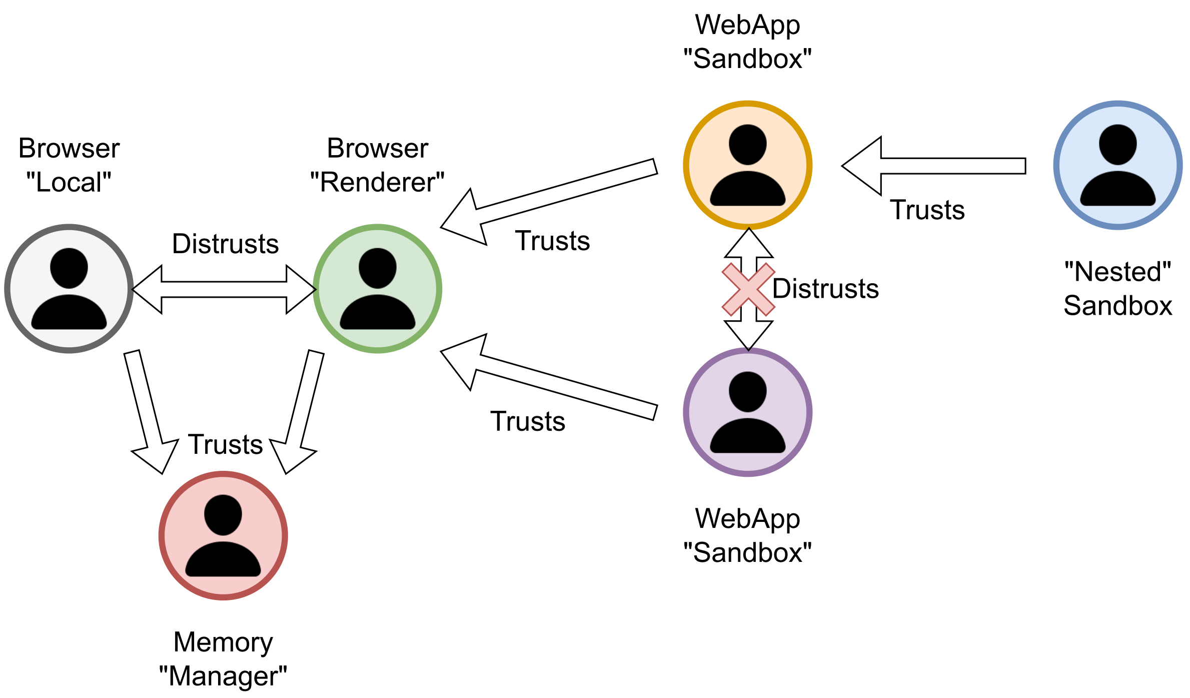


Figure 4.1: Architecture of a compartmentalized *Browser*.

as a consequence. For example, some mechanisms integrate support for system call filtering (e.g., Donky [113], ERIM [129]), while others provide the supervisor the necessary support (SecureCells [15], CHERI [139]), and some mechanisms are inherently unable to support this restriction (MPK [96], CAPSTONE [153]). The differences arise from different design goals, like prioritizing backward compatibility with minimal hardware/software changes over clean-slate designs, and trading-off security checks for greater performance.

4.2.1 Example: *Browser* Compartmentalization

We shall use a fictional architecture for a browser (Figure 4.1) to illustrate how the properties of compartmentalization mechanisms affect realistic applications. Our browser roughly resembles a modern Chromium [11] or Firefox [88, 100] browser, with separate compartments facing the internet and the local system, and also includes novel aspects designed to illustrate desirable features for future applications.

Our *Browser* is built from compartments implementing different functionality and executing with the requisite privileges. The *Local* compartment is responsible for handling interactions with the local system, like accessing files and creating threads. The *Renderer* compartment handles the management of sandboxes for running untrusted code (like JavaScript apps) from websites, including possible communication between mutually distrustful sandboxes. The browser isolates web application code from separate sources (e.g., separate websites) into per-*Sandbox* compartments. Finally, we allow a *Sandbox* to further sandbox a part of itself, effectively creating a *Nested* sandbox. This models the situation where web application developers themselves rely on untrusted code, for example, JavaScript code imported by a package manager. A trusted memory *Manager* compartment handles memory management for the application and can directly update the system/hardware configuration to reflect (de)allocations or permission changes. The *Manager* helps model applications with trusted and privileged user components directly managing system-level configurations traditionally restricted to the supervisor. Future supervisors, with different compartments responsible for different subsystems and having the privilege to modify the relevant system configurations, may also require support for *Manager*-like compartments.

Functionality. Our browser maintains functionality similar to modern browsers. The *Local* compartment handles I/O including user inputs and the networking stack, and passes inputs for a website over to the *Renderer*. The *Renderer* renders the site, including running the site's dynamic code. The *Renderer* either interprets the site's code or creates sandboxes for running compiled/generated versions of the same code. Each *Sandbox* generates website elements while running, which are communicated to the *Renderer*. The *Renderer* generates bitmaps visualizing each website and communicates these bitmaps to the *Local* compartment which finally sends the image to the display device. The *Manager* handles memory management, ensuring separate regions for each compartment when required, and can use privileged operations to modify hardware configuration as required. During this process, compartments within the browser communicate by calling other compartments and sharing data as arguments.

Threats. A browser runs code controlled by many parties and manages different threats. The attacker might control a malicious website and send code when the victim uses the browser to access the website crafted to exploit bugs in the *Renderer*'s code generation engine. The *Browser* attempts to protect the local system from such malicious code by having *Renderer* and *Local* components. In another attack scenario, a web application generates code attempting to access a different website *Sandbox*'s data. Our browser uses per-website Sandboxes to address this threat. A third attack scenario considers the maintainer of a popular JavaScript package used by many websites. The rogue maintainer can make a malicious update to the package which gets pulled by many unsuspecting website developers, and spread to visitors of many websites. The malicious package runs code which tries to access data from the website's application (perhaps the website user's data) and leaks it as a request to another server. Web developers attempt to mitigate this threat by using *Nested* sandboxes for code from untrusted packages.

Trust Relations. The browser described above requires various trust relationships. The *Local* compartment distrusts the *Renderer* since it faces the internet. For simplicity, we assume that the distrust is mutual, and the *Renderer* likewise distrusts the *Local* compartment. In this non-hierarchical setup, two compartments must interact without trusting the other. The sandboxes, however, create a hierarchy of unidirectional trust where the *Renderer* is trusted by each *Sandbox*, which in turn are trusted by their *Nested* sandboxes. The *Renderer* distrusts each *Sandbox*. Likewise, each *Sandbox* distrusts *Nested* compartments. The *Manager* is trusted but distrusts its callers: the *Renderer* and *Local* compartments.

Privileges Expressing Trust Relationships and Functionality. The browser's compartmentalization policy defines privileges for each compartment to express the trust relations described above. A mechanism must implement the restrictions defined below to support the *Browser*'s requirements. Each *Sandbox* must execute in a highly restricted environment, maintaining access to defined code and data memory regions and the privilege to call into the *Renderer* to request services like allocating more memory or managing *Nested* compartments. Each *Sandbox* must be allowed to only call the *Renderer* and its *Nested* sandboxes. The *Renderer* and *Sandbox*es (including *Nested* sandboxes) are all prohibited from using arbitrary system calls. The *Renderer* can implement its functionality, like drawing to the screen, by requesting services from the *Local* compartment. However, the *Renderer* is trusted by each *Sandbox*, and is allowed to directly access their memory regions including modifying their code. The *Local* compartment has greater privileges to use generic system calls. However, the *Local* compartment must be restricted to also filter out system calls which enable direct access to the *Renderer*'s resources (like memory). An untrusted caller must be prevented from controlling the callee's register context after a cross-compartment call. Therefore, a secure context switch is required when calling from the *Renderer* into the *Local* compartment, but is unnecessary when the *Renderer* calls into a *Sandbox*. To prevent side-channel attacks, specific compartments, like the *Sandbox*es, must be restricted from executing microarchitecture-management instructions (like x86's `clflush`) or reading specific (high-precision) timers. Since the *Manager* compartment can modify memory configuration, only that compartment should be allowed to execute instructions that change privileged registers, access permission tables, and/or maintain hardware permission cache (e.g., translation-lookaside buffer or TLB) coherence.

Performance. Our *Browser* must be able to provide a responsive browsing experience to its users, while incurring overheads from the supporting mechanism associated with checks implementing the restrictions described above. Since applications have performance targets, developers seeking to introduce security through compartmentalization must architect their applications considering the overheads of the underlying mechanism. The *Sandbox*es and *Nested* libraries have short execution periods, and cross-compartment calls can execute in sub-microsecond timescales. The *Manager* compartment must manage (de)allocations equally quickly, to match web application performance. Consequently, our *Browser* demands a proportionally fast mechanism. To run on slower mechanisms, we must redesign our browser with larger compartment which execute longer. The coarse-grained compartmentalization employed by the Chromium browser is a result of the significant microsecond-scale overheads of compartmentalization [76] built on traditional processes (besides factors like code complexity). Chromium must limit switches between its compartments, since each inter-process call (IPC) typically costs around ten microseconds.

Higher-performance use-cases for compartmentalization also exist, pushing the need for mechanisms supporting crucial operations like checks on memory accesses, compartment switching, memory (de)allocation, and permission changes/transfers/revocation within hundreds, tens, or even single nanoseconds/cycles. Server applications are prime candidates for improved security from compartmentalization, and are are composed of smaller (often micro-) services, each of which executes under strict microsecond-scale latency/quality-of-service targets. At datacenter scales, even microsecond-scale delays in a few components can balloon into perceptible changes for the end user [39, 71]. The performance characteristics of mechanisms are of paramount importance to developers considering compartmentalization for datacenter applications.

4.2.2 Execution Model and Threat Model

We assume that the system being protected is a modern general-purpose CPU or system-on-chip. Particularly, the system contains one or more general-purpose processors (called cores) and specialized processors (called accelerators). Processors may be time-shared between different applications and between different compartments of the same application. Processors have private hardware resources (like caches) and shared hardware resources (shared caches, network-on-chip, main memory). The use of or access to special resources (like timing units, devices, or shared registers) requires special instructions or system calls to privileged supervisor software. The trusted computing base (TCB) includes the hardware, the supervisor software (unless specified otherwise), and (occasionally) privileged userspace libraries.

The system executes a compartmentalized application, consisting of communicating compartments executing as per a secure and well-defined compartmentalization policy which defines privileges for each compartment. The policy must be secure, as insecure policies allow applications to be compromised irrespective of the underlying mechanism, and would not allow us to compare mechanisms' properties. We also assume that the application has been initialized properly, i.e., that the software-hardware configurations express the correct compartments, and correct per-compartment privileges to the extent permitted by the mechanism. We assume that compartments secure their communication interfaces, the primary remaining attack surface, with necessary checks to prevent confused-deputy attacks. The above assumptions mean that compartments cannot be compromised using any sequence of legitimate calls into the compartments (indicating a shortcoming in the compartmentalization policy, not the mechanism). We assume that the developer tries to reasonably implement the same checks across mechanisms. For example, consider two mechanisms which check memory permissions for 4KB pages and byte-granularity objects, respectively. A developer can reasonably isolate data from two compartments by assigning them to separate 4KB pages on the first mechanism. However, developers will consider the memory overhead of assigning each object on a separate page unreasonable, preventing access control between two objects on the same page.

We consider an attacker who controls the code and/or data accessible to a "compromised" compartment of the target application. The attacker aims to compromise the other compartments of the application, to either corrupt their execution (integrity) or leak information (confidentiality). The attacker must exploit a weakness of the compartmentalization mechanism, not the policy. The attacker can also try to maliciously influence the victim's execution environment, including the core's microarchitectural state. The attacker is allowed to execute any operation permitted by the mechanism, as per the policy. Executing ISA instructions, special register accesses, and system calls are examples of possible operations. The attacker might attempt to: *i)* execute specific instructions including special instructions like system calls or permission modification instructions, *ii)* access/modify specific resources including virtual memory addresses, physical memory addresses, supervisor resources, and/or physical resources, *iii)* execute instructions or operations with specific operands.

4.3 Mechanisms Implement Restrictions

Compartmentalization mechanisms support the least privilege principle by restricting the abilities of individual compartments. Attackers rely on an arsenal of different abilities which can allow them to maliciously leak or corrupt a victim compartment. In Figure 4.2, we show how an attacker executing within a compromised compartment can try to compromise other compartments or the executing system. In the figure, we highlight the classes of restrictions that mechanisms can implement to prevent cross-compartment attacks. Mechanisms can either limit the instructions or operations that a compartment can execute, limit the resources accessible by using operations, or limit the range of arguments that compartments can use for operations.

In this section, we describe abilities available for compartments, whose use might be limited by compartmentalization mechanisms. These abilities are summarized in Table 4.1. Generally, abilities are classified into restrictions on *i)* resources accessible through instructions or operations, *ii)* arguments to instructions, and *iii)* instructions/operations compartments can execute. We discuss each of these abilities below.

4.3.1 Resource Access

Applications require restrictions to limit resources accessible to compartments, aimed at preventing cross-compartment corruption where one compartment can directly access to leak or modify other compartments' resources. For example, each *Sandbox* in our *Browser* is only allowed access to its own private state, holding the values of the objects being used by the website code running in that *Sandbox*. An attacker in a different compromised *Sandbox* with an arbitrary read/write primitive can attempt to access a target *Sandbox*'s data directly. One website running code

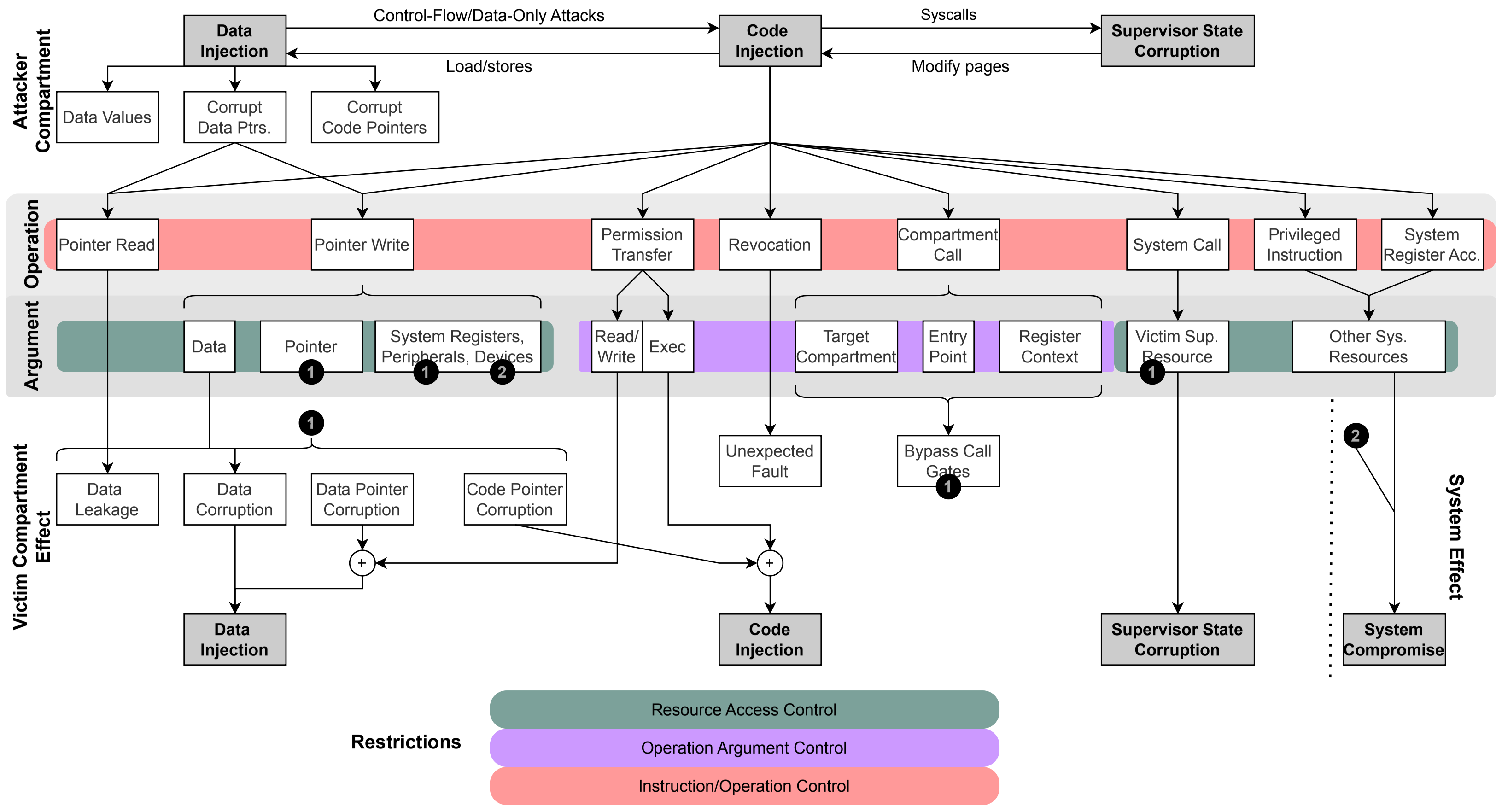


Figure 4.2: Classification of operations leading to a cross-compartment attack, and illustration of classes of mitigating restrictions.

Privilege	Type and Restriction Summary
Resource	Virtual memory Physical memory Supervisor resources, e.g., file handles System resources, e.g., RNG, timers Register contexts
Arguments	Mem. access type (read/write/execute) Inter-compartment call target Compartment entry point, for call gates Perm. transfers: perms., target, granularity Permission revocation type
Operations	Microarch. configuration (e.g., `clflush`) System calls/Traps to the supervisor Permission mod./transfers/revocation Cross-compartment calls Memory (de)allocation

Table 4.1: Summary of privileges restricted by compartmentalization mechanisms

from an attacker-controlled website might attempt to read the login password used for a different website running in a separate tab temporarily stored in memory, for example. Alternatively, the attacker might try to access files on the host system's disk, either to read private files used by other compartments or to compromise the host system itself. The mechanism should prevent both of the above attacks. In Figure 4.2, we show how attackers can attempt to dereference corrupted pointers to read/write across compartments and use system calls to access prohibited supervisor resources.

Common resources requiring access control include virtual memory (VM), supervisor resources, access to input/output interfaces, and special registers. Sandboxes, for example, must be limited to only have access to their own data and code regions. Therefore, virtual memory permissions are a key aspect of intra-address space compartmentalization mechanisms. The alignment and granularity requirements for virtual memory permissions greatly determine a mechanism's security. Kernel resources, including files, networking, or devices, are also crucial for isolating compartments. Essentially, the two aforementioned restrictions stem from restrictions on the resources accessible through load/store instructions and system calls, respectively. Mechanisms might also restrict resources accessible through input/output instructions. However, the ubiquity of memory-mapped I/O allows mechanisms to also restrict I/O by restricting memory access. Alongside virtual memory, mechanisms might also rely on restricting access to physical memory regions to prevent concurrent access to data due to aliasing in virtual memory translations. Since the execution of compartments is often temporally multiplexed on the same physical thread running on a core, compartments might also ensure isolation of register contexts available to compartments. Finally, compartmentalization might restrict access to system-wide or privileged resources or registers such as random number generators or those controlling permissions (e.g., RISC-V's Physical Memory Protection registers).

4.3.2 Operation Arguments

Applications also require limits on how compartments are allowed to use available operations. Figure 4.2 shows how operation arguments can help cross-compartment attackers bypass call gates or inject code and data. For example, system call filters can implement checks on which calls compartments can execute. Compartments might have permission to use the `write` system call in Linux for writing to the terminal window but are only allowed to write to the `stdout` file descriptor and denied access to other open files. The *Sandbox* compartment in our example *Browser* must communicate with the *Renderer* compartment, and hence has the privilege to use the compartment switch/cross-compartment call operation. The mechanism might limit the target compartment, ensuring that only transitions to the *Renderer* are allowed while denying attempts to directly call into other Sandboxes. Further, when calling into the *Renderer*, the execution must enter at predetermined entry points allowing the *Renderer* to implement call gates. The mechanism must prevent cross-compartment switches to arbitrary code points in the target. A mechanism might restrict permission modification operations so that the new permissions are lesser than the original permissions, to prevent privilege escalation. For mechanisms supporting the transfer of memory permissions between compartments, similar limits can be applied to prevent arbitrary permission transfers. Mechanisms can allow permissions to be transferred unilaterally or require bilateral consensus between the granting and receiving compartments. Systems supporting permission revocation may limit when granted permissions can be revoked, and in which order. Finally, mechanisms could limit permitted operands for instructions (e.g., x86's `CPUID`, RISC-V's `csrr/w`) for accessing system resources (e.g., control-status registers in RISC-V) as per an allowlist.

4.3.3 Instructions and Operations

Mechanisms might altogether restrict access to ISA instructions or other operations, to limit non-general purpose computation (e.g., arithmetic) possible by compartments. In Figure 4.2, we show how different classes of instructions enables different cross-compartment attack vectors. These restrictions can prevent select compartments from using instructions/operations with dangerous effects on the environment. For example, a sandboxing engine might rely on a mechanism to restrict sandboxes from executing system calls altogether. Mechanisms might restrict instructions which control microarchitectural configurations, such as a cache-line flush instruction (`clflush`) or processor halt. Instructions controlling microarchitectural state are crucial for some microarchitectural side-channel attacks [47, 152]. Generic compartmentalization can also benefit from mechanisms restricting the ability to use instructions/operations for modifying, transferring or revoking permissions to specific compartments to only those compartments requiring dynamic permission changes, and to instructions triggering cross-core interrupts. Compartments requiring deterministic execution might also be barred from special randomness-generating instructions like x86's `rdrand`, which might enable attackers to access special shared resources [104] or reverse-engineer the generator seed [7, 24]. While traditional architectural privilege levels already restrict which instructions are available at different levels, compartmentalization can bring a similar security benefit between userspace compartments. When offloading traditionally supervisor functions to userspace, mechanisms must also enable limiting the use of traditionally privileged instructions (like flushing TLB entries) to authorized compartments only. Our *Browser* would benefit from such a restriction, ensuring that a compromised *Sandbox* with injected code can never modify memory configurations like the *Manager* is permitted to.

4.4 Practical Considerations

Using a mechanism to compartmentalize an application leads to the application inheriting the mechanism's performance overheads and other practical limitations. These limitations may arise from the architectural design of the mechanism or from an implementation's microarchitectural limits. Limitations can lead to either situations where an application cannot be supported by a mechanism, or where the application suffers additional overheads beyond a limit. In this section, we will discuss various limitations of mechanisms, and what overhead applications will incur as a result.

Limitations. Mechanisms targeting a wide range of applications offers the best path towards adoption, and must support varying use cases, policies and performance requirements. Our *Browser*, for example, spawns a varying number of *Sandbox*es, and requires mechanisms which can allow for many compartments and many memory regions for these compartments. Our *Browser* also requires different models of trust, including mutual distrust and hierarchical trust between compartments. Mechanisms built for hierarchical trust only are unsuitable. Other possible limitations are dependence on a specific architecture, limiting common software practices like code sharing between compartments, or preventing integration with other orthogonal security mechanisms.

Memory Access Control. The most frequent check implemented by compartmentalization mechanisms is access control to memory, for loads/stores and for fetching instructions. Our example *Browser*'s memory footprint can scale from megabytes to gigabytes and beyond, and mechanisms must limit access control overheads in the face of scaling data set sizes. Modern high-performance desktop and server cores are highly sensitive to delays in accessing memory, and computer architects strive to enable fast, single-cycle permission checks. This criterion has only gained importance since the discovery of the Meltdown [74] attack, and puts permission checking on the critical path of the memory access to prevent illegal memory accesses from speculatively leaking data through side channels. All modern protection mechanisms depend heavily on metadata caching in microarchitectural structures near the core to enable single-cycle checks in the common case, and the performance of a mechanism is determined by how much data can be accessed through cached permissions. The performance of permission caching depends on the number of permissions the mechanism needs to track, which is determined by the granularity of permission enforcement and the count of individual permissions. Where mechanisms rely on permissions stored in in-memory data structures (page table entries, for example, store permissions in a radix tree) permissions are often cached in lookaside buffers near the core. The size of these buffers and the size of memory represented by each permission entry determines the buffer's reach, i.e., the size of memory for which permissions can be effectively cached. Permission tables for fixed-size memory regions like pages (traditional virtual memory), hugepages or words require more permission entries and incur more costs/overheads than permissions for variable memory ranges, which can scale to cover arbitrarily large regions. The choice of granularity of permission also affects performance. Mechanisms with permission to variable-sized memory ranges can target storing permissions for each object or for each virtual memory area (VMA). Applications can require a million times more objects (100's of millions) vs a few hundred to thousand VMAs. An alternate class of designs uses hardware capabilities to memory, stored in memory, and used for validating accesses. These mechanisms must also be able to validate the access against the presented capability within a single cycle. However, capability-based mechanisms typically target per-object capability storage leading to significant storage overheads and pressure on the small core-private data caches shared between data and cached capabilities. Fortuitously, capability systems benefit from the better scaling trends of L1 data caches compared to TLBs, allowing more permissions to be cached close to the core. Alternate capability-based designs storing per-page permissions can scale to efficiently protect even larger regions.

Compartment Switching. Compartment switching using inter-compartment calls is expected to also be a frequent operation, and its latency can significantly affect application performance. A *Sandbox* might use the services of a library in a *Nested* sandbox at sub-microsecond timescales. Assuming fine-grained compartmentalization where small snippets of code, corresponding to library calls or sandboxes, are isolated in compartments, efficiently servicing a sub-microsecond cross-compartment call requires the mechanism to support fast compartment switching within tens or up to a hundred nanoseconds. Conversely, a mechanism requiring tens of microseconds to switch between compartments will only efficiently support applications where compartments execute for longer timescales (perhaps milliseconds). Inter-process communication system calls require microseconds on modern commercial operating systems like Linux or seL4, and this cost limits compartmentalization for applications like browsers, microservices or virtualized network functions in the cloud. The need for faster switching times is a key motivator for proposals for alternate OS isolation abstractions and even for hardware-accelerated switches.

Instruction Execution Control. Another conceptually frequent check would be to limit which instructions a compartment can execute. Currently, few mechanisms implement such functionality for general instruction classes. Only restrictions on instructions based on an executing thread's privilege level are widespread. Restricting instructions might require changes to a core's decode stage, which is responsible for reading and decoding multiple instructions every cycle on desktop/server processors. Given the complexity of the decode stage, modifications might introduce delays leading to significant performance overheads. However, systems with specific use cases might reap sufficient benefit from delegating common privileged tasks to trusted userspace compartments to justify this cost. The *Manager* compartment, for example, would enable secure memory management while securely manipulating memory-management unit (MMU) configuration registers.

Exclusive Access. Certain mechanisms offer additional features, like the ability to maintain exclusive access to memory regions, transfer permissions to memory regions between compartments and revoke such grants. Each of these operations also has a performance cost which will impact how often applications will use them. Mechanisms with centralized permission tables and permission lookaside buffers will suffer the cost of not only modifying permissions, but propagating these changes between cores to maintain coherence with their private lookaside buffers. In contrast, compartments can transfer permissions by passing capabilities in registers on supported systems. These mechanisms can offer significantly faster permission transfer operations. In contrast, revoking granted permissions can be simpler on mechanisms with a centralized permissions table, which only needs to modify one set of permissions when compared to capability-based systems which might need to scan all memory regions for copies of the revoked capability. Memory scanning significantly hurts application performance due to large and unpredictable delays when a scan is triggered. Limiting the spread of permission storage within CPU registers or limiting duplication of capabilities also helps revocation performance.

4.5 Evaluating Mechanisms

In this SoK, we compare the security and performance characteristics of the mechanisms listed below in Section 4.5.1. The comparison aims to illustrate common strengths and weaknesses between mechanisms. Further, we aim to expose the main missing features that we consider useful for future proposals for comprehensive compartmentalization. We score mechanisms on features introduced in Section 4.3 and Section 4.4. The basis for our scores is presented in detail in Section C.1 and summarized in Figure 4.3 and Figure 4.4.

4.5.1 Mechanisms Compared

In our SoK, we have tried to include a variety of mechanisms across a variety of design points. These mechanisms are listed in Table 4.2. As our baseline, we include process-based compartmentalization on traditional operating systems, which is commercially used for software compartmentalization. Labeled "TRAD", our baseline includes traditional monolithic operating systems like Linux, FreeBSD, and Darwin as well as commercial microkernels like seL4. In the "HW/Sup", we classify what

Mechanism	HW/ Sup	Perm. Storage	Runtime TCB	Perm. Grain
TRAD	NA	PT	Sup	Fixed
lwC [76]	Sup	PT	Sup	Fixed
CODOMs [131]	HW	PT+Cap	HW	Fixed
XPC [40]	HW	PT	HW	Both
ARMlock [155]	Sup	PT+Reg	HW-Sup	Fixed
MPK [96]	NA	PT+Reg	HW	Fixed
ERIM [129]	NA	PT+Reg	HW	Fixed
Donky [113]	Both	PT+Reg	HW-User	Fixed
MMP [144]	Both	PT	Sup	Range
SecureCells [15]	Both	PT	HW	Range
CHERI [139]	Both	Cap	HW	Range
CAPSTONE [153]	Both	Cap	HW	Range

Table 4.2: Summary classifying surveyed compartmentalization mechanisms characteristics. Columns are described in Section 4.5.1.

changes are required for each mechanism. Among research proposals, we include mechanisms which can run on commercial-off-the-shelf hardware with no changes (NA), mechanisms requiring supervisor code changes ("Sup"), hardware modifications ("HW") or both. We have chosen mechanisms to span the range from designs targeting full backward compatibility to designs aiming for minimal changes to clean slate designs. Surveyed mechanisms rely on a trusted computing base at runtime dependent on the supervisor ("Sup"), hardware ("HW") or even privileged userspace libraries ("User"). The variation continues with implementations of permission storage, including mechanisms relying on variations of permissions tables ("PT"), special permissions registers ("Reg"), capabilities ("Cap"), or a mix of the above. Finally, we find mechanisms using permissions for fixed-size memory regions (pages, words), variable-sized regions (objects, segments and virtual memory areas), and even both.

In our comparison, we have omitted other compartmentalization mechanisms solely based on software-based checks [135] which are incapable of protecting against strong attacks as defined in Section 4.2.2 as they do not add any protection beyond that provided by traditional processes. We have also excluded mechanisms based on virtualization [13, 67, 77, 84, 119] as they provide similar protections as processes, but at a higher cost and protect against potentially rogue supervisors. We have also omitted other mechanisms based on novel OS abstractions due to a lack of space [19, 23, 38, 55]. We also exclude mechanisms abusing non-security hardware features for non-systematic point protections [48, 52]. Finally, we exclude mechanisms with very limited protections [42].

Process-based compartmentalization (TRAD)

The traditional (TRAD) abstraction of isolation is based on OS processes. We will describe process-based compartmentalization in UNIX-like OSs (particularly Linux), though the concepts generalize to other OS kernels.

The process abstraction was introduced to isolate different applications running on multi-user machines. Each process has an isolated memory space (architecturally-defined virtual memory) and individual OS resources, such as file descriptors, I/O handles or capabilities. The OS kernel time-multiplexes processes onto one or more processing cores. A process can have one or more kernel threads, corresponding to independent threads of execution, but sharing the same address space and OS resources. Everything within a process shares the same permissions to access its memory and its OS resources, including the application's executable, libraries, and loaded modules. The OS relies on hardware enforced permissions for virtual memory, typically implemented by per-core memory-management units (MMUs) reading permissions from a per-process page table structure.

A developer can compartmentalize an application by isolating each compartment in a separate process. Each process will have its own private memory address space, and its own OS resources. One process cannot name a separate process' resources since they each have separate namespaces, and hence cannot directly access another process' resource. OS resources require an open-like step, ensuring that each process in the application only accesses allowed resources. Processes cannot arbitrarily gain capabilities. Processes can, additionally, share memory and specific OS resources for communication. Shared memory must be set up with explicit system calls, and are subject to syscall filtering. The generated code region can be shared between the JIT and sandboxes, along with code sections for shared libraries. Processes can also communicate using inter-process communication (IPC) system calls (like `sendmsg`/`recvmsg`). A remote (cross-process) procedure call typically consists of serialization of arguments into a buffer, sending the buffer across using a system call, and then deserialization of the arguments on the receiving side followed by the requisite processing based on the arguments. The cost of inter-process switching contributes a great deal to the expensive nature of compartmentalization with processes.

A *Browser* can be compartmentalized using processes by isolating sensitive components in separate compartments, each of which runs in a separate process. For example, the *Renderer* can occupy one process and each *Sandbox* can be assigned a separate process. Microsoft's ChakraCore JavaScript engine [86], for example, used such an architecture to isolate itself from untrusted sandboxes. A shared memory region between the *Renderer* and each *Sandbox* holds the *Sandbox*'s code. This region is mapped as executable in the *Sandbox*, and writable in the *Renderer*. The *Renderer* and sandboxes can communicate using the kernel's inter-process message passing system calls. The Chromium web browser further implements a *Local* compartment, isolating parts of the *Browser* interacting with the local system [11].

Mondrian Memory Protection (MMP)

MMP [144] tackles the challenge of flexible, fine-grained intra-address space isolation. Intra-address space compartmentalization differs from previous mechanisms, all of which have separate memory address spaces for each compartment. In contrast, compartments in MMP (and the following mechanisms) all share the same address space. Isolation between compartments relies on different memory views, i.e., different permissions to the same address for different compartments. Intra-address space compartmentalization can simplify the process of porting monolithic applications to compartmentalize them, and allow smaller compartments with fine-grained memory permissions. Data transfer for intra-address space compartmentalization can be simpler than with IPC, since addresses remain valid across compartments.

Each compartment in a application compartmentalized under MMP maps to a protection domain, with an unique *Domain ID*. The virtual address space is split into a number of segments. Each compartment has its own permission table which holds per-segment permissions, essentially allowing the domain to have its own view of permissions to memory. The permission tables can be configured to allow segments to be private (only one domain has permissions), or shared. MMP also describes different structures for the permissions table in memory, and the hardware structures to read and cache permissions. MMP's permissions tables separate permissions from translation, and are independent of page tables. Crucially, MMPs permission tables allow segments starting and ending at word boundaries instead of page boundaries, allowing spatially fine-grained permissions. Compartments in MMP can call each other through system calls or traps, which also modify the permissions table base pointer register in hardware. The paper also proposes that hardware can be used to accelerate this switch, though the mechanism is not clearly explained. MMP requires call gates to prevent control flow attacks at inter-compartment boundaries. Data can also be passed between domains during switching, by marshalling data into copied buffers. However, MMP also introduces the concept of zero-copy data passing between compartments by modifying the permissions to the segments holding passed data. However, few details on maintaining coherence of on-chip permission buffers are discussed.

In MMP, the *Browser* fits in a single address space with separate protection domains assigned to the *Renderer*, *Local* compartment, the *Manager* and for each of the *Sandbox*es. For each *Sandbox*, the generated code can be assigned a separate segment which has different permissions for the *Renderer* (writable), and for the *Sandbox* domain (executable). Further, transitions between compartments use protected calls with call gates which ensure proper checks on these transitions.

Light-weight contexts: An OS abstraction for safety and performance

This proposal [76] introduces a new eponymous OS abstraction, Light-weight contexts (lwCs), which are independent units of isolated execution. Contexts, like processes, have separate memory address spaces, OS resources, and capabilities. However, contexts remain part of a single process and share metadata in the kernel, resembling kernel threads. Contexts offer the primary advantage that switching contexts within a process is faster than switching processes, or even kernel threads within the same process. lwC achieves faster switching between contexts by eliminating unnecessary kernel processing due to the kernel scheduler and resource accounting.

Contexts diverge from the point of calling `lwCreate` which acts like the `clone` system call, where each resulting context has an independent memory address space and OS resource handles. Like processes, OS resource handles can persist across a `lwCreate`, or be invalidated in the child. Further shared resources can be generated using the `lwOverlay` system call. During an application's setup phase, numerous contexts may be created, each of which can perform private setup steps or further restrict their resource rights using `lwRestrict`. Execution of contexts resembles processes, merely replacing inter-process system calls with the faster inter-context switches.

A *Browser* compartmentalized using lwC looks essentially the same as using UNIX processes. The *Renderer* and each *Sandbox* occupy separate contexts, instead of processes. With lwC, message passing system calls will be replaced by `lwSwitch` system calls.

Code-centric Memory Domains (CODOMs)

CODOMs [131] enabled fine-grained intra-address space compartmentalization with a novel architecture where domains are identified by the executing instruction pointer (hence the "code-centric" name), and with permissions to data regions determined by the executing domain. Essentially, bits of page table entries identify the domain owning that page. Executing an instruction from a page tagged with a domain ID equates to executing as that domain.

Domains in CODOMs each have permission to specific pages of memory: those tagged with that domain's tag in the corresponding page-table entry. A domain cannot access data belonging to other domains, except if explicitly allowed during access protection lists (APLs). Compartments can communicate with function calls, which implicitly cause compartment switches when the domain of the target address page differs from that of the source. APLs specify the ability of some domains to call other compartment, as well as for the caller to share their data with the callee. While APLs allow domains to permanently share data with other domains, CODOMs also proposes the use of capabilities to temporarily share permissions to data during cross-compartment calls.

Under CODOMs, a *Browser* must have separate pages corresponding to the *Renderer* and for each *Sandbox*. Each domain's pages must be correctly tagged in the page table to allow the domain to be identified by the hardware. Each *Sandbox* must own the pages holding their code regions, and have executable permissions in the page tables. The *Renderer* can generate new code using additional permissions specified in the APLs, where the *Renderer* should be granted write permission to each *Sandbox*'s code pages.

XPC: Architectural support for secure and efficient cross process call

XPC [40] shares much of the abstractions from the process-based isolation, but attempts to accelerate inter-process calls, in particular, while maintaining the same interface. Backward compatibility is a major design directive. Particularly, XPC replaces the system calls used for IPC with hardware instructions supported by state machines implementing the same functionality. XPC aims to reduce the cost of IPC compared to kernel software, also eliminating software dispatch and scheduling overheads in the common case. XPC also focusses on cheap zero-copy data movement between processes, dedicating a single relay segment for the purpose.

A compartmentalized application running under XPC looks essentially identical to that using processes. Just like previously, each process has their own address space and OS resources and capabilities. The XPC hardware engine tracks the page-table pointer and capability pointer for each process of a compartmentalized application in an *x-entry* held in an in-memory *X-Entry Table*. The OS sets up the X-Entry Table during an application launch. While a process is running, the XPC engine ensures that the hardware uses the correct page table pointer and capabilities. XPC accelerates remote procedure calls, introducing the `xcall` and `xret` instructions to replace `sendmsg`. On executing `xcall`, the hardware fetches and installs the relevant page table pointer and capabilities for the target process, and put an entry for the caller on a *Link Stack*. Executing `xret` allows the callee to return to the caller, and the hardware engine pops the caller's information and installs it in the corresponding system registers (including the return address). XPC eliminates the OS kernel from inter-compartment calls, relying on the hardware to implement traditional kernel functionality. Additionally, data passed between processes can use the relay segment, which is a single dedicated segment mapping memory separately from the page tables.

A *Browser* compartmentalized using XPC looks essentially the same as using UNIX processes. The *Renderer* and each *Sandbox* occupy separate processes. The major difference is that IPC system calls are replaced by faster `xcall`/`xret` pairs.

ARMlock

ARMlock is a mechanism for hardware-supported intra-process sandboxing, based on ARM's support for memory domains. ARMlock leverages four bits in the page table entries to partition the process' virtual address space into 16 memory domains. At runtime, a core's permissions to each of these domains is dictated by the privileged domain access control register (DACR). The value in DACR allows each domain to be in one of three states: inaccessible, accessible as per page table permissions (Client), or accessible without restrictions (Manager). A system call is used to change permissions to memory domains, by writing to the DACR register.

ARMlock supports intra-process compartmentalization by dividing the memory space into memory domains, and by setting the DACR register to properly limit accessible memory during each compartment's execution. A memory domain can be private to a compartment, where the DACR register bits for that domain are set to inaccessible whenever any other compartment is executing. Similarly, compartments can share a memory domain as equals if the DACR permissions are set to Client during each of their executions. ARMlock, however, assigns and restricts each unprivileged compartment to a single domain, and restricts sharing. The Manager permission setting allows ARMlock to implement support for a privileged compartment which has unrestricted access to domains. Finally, ARMlock implements a specialized stub including context switching during the system call used to switch compartments.

A *Browser* can implement isolation between the *Renderer* and each *Sandbox* with ARMlock, by assigning each *Sandbox* a separate domain. During a *Sandbox*'s execution, the DACR register permits accesses to the *Sandbox*'s domain, with read-write data permissions or read-execute code permissions as per the page table permissions. The *Renderer*, however, must generate code and write to the *Sandbox*'s code region, and must set the DACR register to Manager mode for the corresponding *Sandbox*'s memory domain. However, ARMlock cannot prohibit a buggy *Renderer* from executing a *Sandbox*'s code since the domain is accessed with Manager permissions. ARMlock can also isolate separate privileged compartments, such as the *Manager* and *Local*, from the *Renderer*. The *Renderer* has Manager permissions for *Sandbox*'s domains, but no access to *Manager* and *Local* memory domains.

Intel Memory Protection Keys (MPK)

Memory Protection Keys assign keys to regions of memory, and maintain a separate set of access permissions for each key. While previous implementations in ARM, PA-RISC, Itanium and POWER-6 rely on supervisor managed protection key permissions, Intel's MPK extension makes permission changes cheap by allowing userspace modification to permissions. MPK uses bits of the page table entry to assign each page a 4-bit color key, and uses permission bits in the per-core PKRU register to control permissions to each page color. PKRU permissions can be arbitrarily changed by userspace, using a special instruction (`wrpkru`). A software library (libmpk [96]), can be used to virtualize page colors, allowing for more than 16 page colors.

When executing a compartment, MPK uses permissions in the PKRU register to enforce additional restrictions on memory access. Each core's PKRU register should only have permissions for page colors as per the compartment executing on that core. Compartments in MPK can implicitly switch between each other using the `wrpkru` instruction to change the core's accessible page colors. Along with the permission changes, the application can also use function calls to implement cross-compartment procedure calls. Data can be transferred between compartments using shared page colors. Changing page colors requires page table entry modifications, implying a system call, and cannot be done as fast as PKRU writes. MPK lacks call gates, and software must ensure control flow integrity between compartments.

A *Browser* can be compartmentalized with MPK, with one page color for the *Renderer* and different colors for each of the *Sandbox* regions. The PTEs for all generated code regions must have writable and executable permissions, since PTE permissions are also enforced. During the *Renderer*'s execution, the PKRU register holds writable permissions for the *Sandbox* code regions. During *Sandbox* execution, the PKRU register holds executable permissions for that *Sandbox*'s code region, and no permissions for any non-*Sandbox* regions.

ERIM: Secure, efficient in-process isolation with protection keys

ERIM [129] builds on top of Intel MPK to implement strict call gates for compartment switching, mitigating a major shortcoming with Intel's MPK technology. Essentially, ERIM limits the existence of instructions which can modify the PKRU value (`wrpkru` and `xrstor`) to within software call gates. However, ERIM must inspect all newly loaded code, including shared library or module loading, to ensure that new instructions which modify the PKRU register are not injected.

ERIM relies on the same set of MPK permissions described above to isolate compartments. ERIM's main differentiating feature are its call gates, which should be the only parts in the application with executable `wrpkru` or `xrstor` instructions. The key feature of ERIM's call gates are that `wrpkru` instructions are immediately followed by a call to trusted code, or by a condition which checks the value in the PKRU register. The latter check allows control flow attacks which try to load invalid permissions to be immediately detected.

A *Browser* compartmentalized with ERIM essentially looks the same as with MPK, with the *Renderer* occupying a trusted compartment, and each *Sandbox* occupying one untrusted compartment. Data can be passed through pointers directly, just as with MPK, though compartment transitions use ERIM's aforementioned call gates.

Donky: Domain keys - Efficient in-process isolation for RISC-V and x86

Donky [113] aims to retain MPK's primary performance advantage, by allowing changing memory views in userspace, while mitigating its main weakness, where an attacker with arbitrary code execution can immediately bypass MPK's protections. Donky replaces Intel's PKRU register register with a new DKRU register which cannot be directly modified by userspace. Donky effectively introduces a new privilege level within userspace, running a special software monitor called the Donky monitor solely capable of modifying the DKRU register. The Donky monitor can be trapped-into by userspace software, enabling the monitor to serve inter-compartment call requests, among other requests which modify memory keys. While Donky's monitor does not run within the supervisor privilege level, its design of entry-exit through hardware traps and ability to modify registers not accessible to normal userspace code makes the Donky monitor similar to previous works of supervisor-mediated compartmentalization.

Compartments in Donky use separate regions of memory, with permission enforced as per the DKRU register, similar to compartmentalization with MPK. These permissions allow memory isolation for compartmentalization. Compartments can call each other using `dcalls`, essentially trapping into the monitor which modifies permissions in the DKRU register and sanitizes register values before dropping into the callee.

A *Browser* compartmentalized with Donky uses separate Donky domains for the *Renderer* and for each *Sandbox*, created by calls to the Donky monitor. The *Browser* can also install `dcalls` between the *Renderer* and each *Sandbox*, but prohibit `dcalls` between sandboxed domains directly. Each transition between domains using a `dcall` is interposed by the monitor. Inter-compartment `dcalls` ensure that compartments are entered at valid entry points. Arguments must be passed through either registers, or through shared memory between domains.

CHERI: A hybrid capability-system architecture for scalable software compartmentalization

CHERI [145] introduces architectural support for memory capabilities. Architecturally, pointers are replaced by capabilities, which track spatial bounds of memory accessible using that capability. CHERI compartmentalization [139] repurposes CHERI capabilities, with a customized CheriBSD kernel to provide intra address-space compartmentalization. CHERI provides compartmentalization based on the object-capability model, where each compartment is represented as an object encapsulating capabilities for the compartment's code and data regions. CHERI is drastically different from the previously mentioned mechanisms, as access permissions for a compartment are not stored in a centralized permissions table or permissions register. Instead, CHERI relies on capabilities distributed within a compartment's registers and data regions.

CHERI uses the capabilities to memory to spatially limit the memory regions accessible to a compartment. When executing, CHERI requires one or more code and data capabilities. For compartmentalization, each compartment encapsulates its code and data capability within an object, and sealed by an object type capability field (`otype`). When running as that compartment, those capabilities are installed in CPU capability registers, allowing the compartment to only access its code and data. Compartments can call each other using a system call to the CheriBSD kernel, which securely saves the caller's state to its object, and unseals the callee's code and data capabilities and installs them in the core's registers before dropping into the callee. Further, CHERI's capabilities greatly simplify passing permission to arguments during an inter-compartment call. A CHERI caller can simply pass a capability to the argument in memory during the call, allowing the callee to use the capability to access the corresponding memory. One consequence, which the caller must be careful about, is that the callee can use any capabilities stored in this argument memory region to further access other regions of memory. CHERI compartmentalization, though, lacks a mechanism for revocation, instead suggesting the use of garbage collectors for eventual revocation. A key concern for developers using CHERI is the possibility of capabilities being leaked during inter-compartment calls.

With CHERI, each of a *Browser*'s compartments must be allocated a separate object encapsulating that compartment's code and data. The *Renderer* would be an object, and each *Sandbox* will be implemented as a separate object. Both the *Renderer* and *Sandbox*es must each hold capabilities to the *Sandbox*'s code region. The *Renderer* can use its own capability to a *Sandbox*'s code region, permitting write operations to generate new code for the *Sandbox*. The *Sandbox*'s own capability for this region must only allow executable operations. Finally, the *Renderer* can pass temporary capabilities to *Sandbox*es when they are required to process certain data, including new data packets, with zero-copy.

SecureCells: A Secure Compartmentalized Architecture

SecureCells [15] introduces a compartmentalization mechanism based on hardware access control to variable-sized regions of memory, hardware tracked compartment identifiers, a unified permissions table for all compartments, and unprivileged instructions for securely accelerating common operations. SecureCells' permissions table stores permissions for each compartment to each data region. Further, SecureCells' userspace instructions enable control flow and zero-copy data movement between compartments with unprivileged instructions. SecureCells' unprivileged instructions implement hardware checks in order to prevent privilege escalation.

Each compartment in SecureCells is assigned a SD, whose accesses to memory regions are checked with an in-memory permissions table. Properly configured permissions allow compartments to have private regions to which on that compartment has permission, and selectively shared regions with specific other compartments having permission. When executing code for a compartment, a core tracks the executing compartment identifier in a system register, and accordingly presents a view of memory. Compartments can also interact with unprivileged `SDSwitch` instructions which atomically switch to a different compartment and jump to the callee's entry point. To aid zero-copy data transfer, SecureCells also includes unprivileged instructions which move permissions for regions between compartments.

A *Browser* compartmentalized with SecureCells will require separate SDs for each *Sandbox* and for the *Renderer*. The permissions table is set up with per-*Sandbox* private regions, to which only each *Sandbox* and the *Renderer* have permission. The permissions can allow the *Renderer* write access and a *Sandbox* execute access to that *Sandbox*'s code region. The *Renderer* can use `SDSwitch` to enter and exit *Sandbox*es. Compartments must use the `SDEntry` instruction to mark valid entry points leading to call gates to switch context and check inter-compartment call arguments. If arguments are isolated within a memory region, compartments with SecureCells can also move these arguments between compartments without copying by transferring permissions.

CAPSTONE

CAPSTONE is a proposal introducing linear hardware capabilities for memory. Hardware capabilities in CAPSTONE limit the memory range accessible using that capability. The CAPSTONE instruction set and hardware additionally provides a uniqueness property for each linear capability, guaranteeing the absence of any other capability in the system allowing access to the same region of physical memory. Linearity of capabilities solves a major shortcoming in older capability systems like CHERI: the potential of capability leakage. Further, linear capabilities allow a core to assume exclusive access to memory regions, removing the chance of data races and implementing hardware guarantees similar to Rust's ownership model. Finally, CAPSTONE introduces a revocation tree to enable compartments to generate revocation capabilities corresponding to linear capabilities, and later use a revocation operation to revoke capabilities transferred to other compartments. However, each memory access under CAPSTONE using a linear capability is accompanied by a second corresponding access to the revocation tree to verify the temporal validity of the capability.

Like other capability systems, CAPSTONE can implement compartmentalization by limiting the memory regions accessible during a compartment's execution to the regions accessible using capabilities held in processor registers and memory accessible through those registers. Compartments can also call other compartments using a special `call` operation on sealed capabilities representing the capabilities of the target compartment. The hardware is responsible for securely and atomically switching the caller and callee's register contexts (including capabilities) between memory and the core's register file during an inter-compartment `call`. A calling convention preserves specific registers over the call, allowing capabilities to arguments to be passed between compartments.

A *Browser* compartmentalized with CAPSTONE will resemble a *Browser* compartmentalized with CHERI, with a few key differences mandated by the use of linear capabilities. For example, linear capabilities prohibit the *Renderer* and a *Sandbox* to simultaneously hold capabilities for the *Sandbox*'s code region. Consequently, the *Browser* must pass the corresponding capability between the two compartments — the *Renderer* passses the capability to the *Sandbox* when launching the *Sandbox*, and the *Sandbox* returns the capability to the *Renderer* when new code needs to be generated. A naïve *Browser* implementation requires that a read-write-execute capability to allow the *Renderer* to write code to the same region executed by the *Sandbox*.

4.5.2 Methodology and Rationale

We classify mechanisms on the axes described in Section 4.3, essentially trying to measure how well each mechanism supports per-compartment restrictions to executable operations, resources accessible by those operations, and arguments to those operations. Each mechanism is individually scored on each axis, and the results are shown as a radar plot (see Figure 4.3 and Figure 4.4). To measure a mechanism's score on an axis, we essentially add scores corresponding to various restrictions that could be implemented on that axis. For each restriction, mechanisms are scored by whether that mechanism supports that restriction, and how well it supports the restriction compared to the other mechanisms. For example, when scoring a mechanism on resource access restrictions, we add scores corresponding to limits on memory accesses and supervisor resource accesses. When scoring mechanisms on virtual memory controls, we consider whether mechanisms restrict instruction accesses (execute) as well as data fetches (load/stores). For data access, mechanisms that support finer-grained access control checks, such as per-object permissions, score more than mechanisms that enforce page-granular access control.

4.5.3 Comparison

The scores for mechanisms on two aspects, security and practicality, are illustrated in Figure 4.3 and Figure 4.4, respectively. In this section, we focus on insights gained from broadly comparing mechanisms. Overall, we find that the most secure mechanisms are designed for strong attacker models, including using operating-system abstraction (TRAD, lwC), capabilities (CHERI, CAPSTONE) and hardware permission checks (MMP, XPC, SecureCells). Further, hardware changes designed for compartmentalization are essential for high-performance mechanisms (XPC, CHERI, SecureCells and CAPSTONE).

We postulate that a mechanism's design greatly influences its security characteristics. Given that mechanisms are designed for different threat models, their security scores for our powerful attacker model can seem lacking. Mechanisms targeting strong attacker models (TRAD, lwC, XPC, CHERI, SecureCells, CAPSTONE) present strong scores on restricting access to resources and operations. Other mechanisms (CODOMs, MPK, ERIM, and Donky) have simpler designs trading off security for simplicity and performance. For example, MPK only checks the data access path. A more secure version of MPK, also checking code fetches, would require changes to the core's (complicated) fetch stage, and require additional bits in the permissions key register (PKRU) for permission storage.

Access control for memory varies on two factors, the granularity of targeted permissions and alignment requirements for regions. Finer, object-level permissions are supported with permission tables (MMP, SecureCells) but are more suitable for capability systems. Most permission table-based mechanisms target access control for larger fixed-size (2MiB, 4KiB or 16KiB) pages or memory

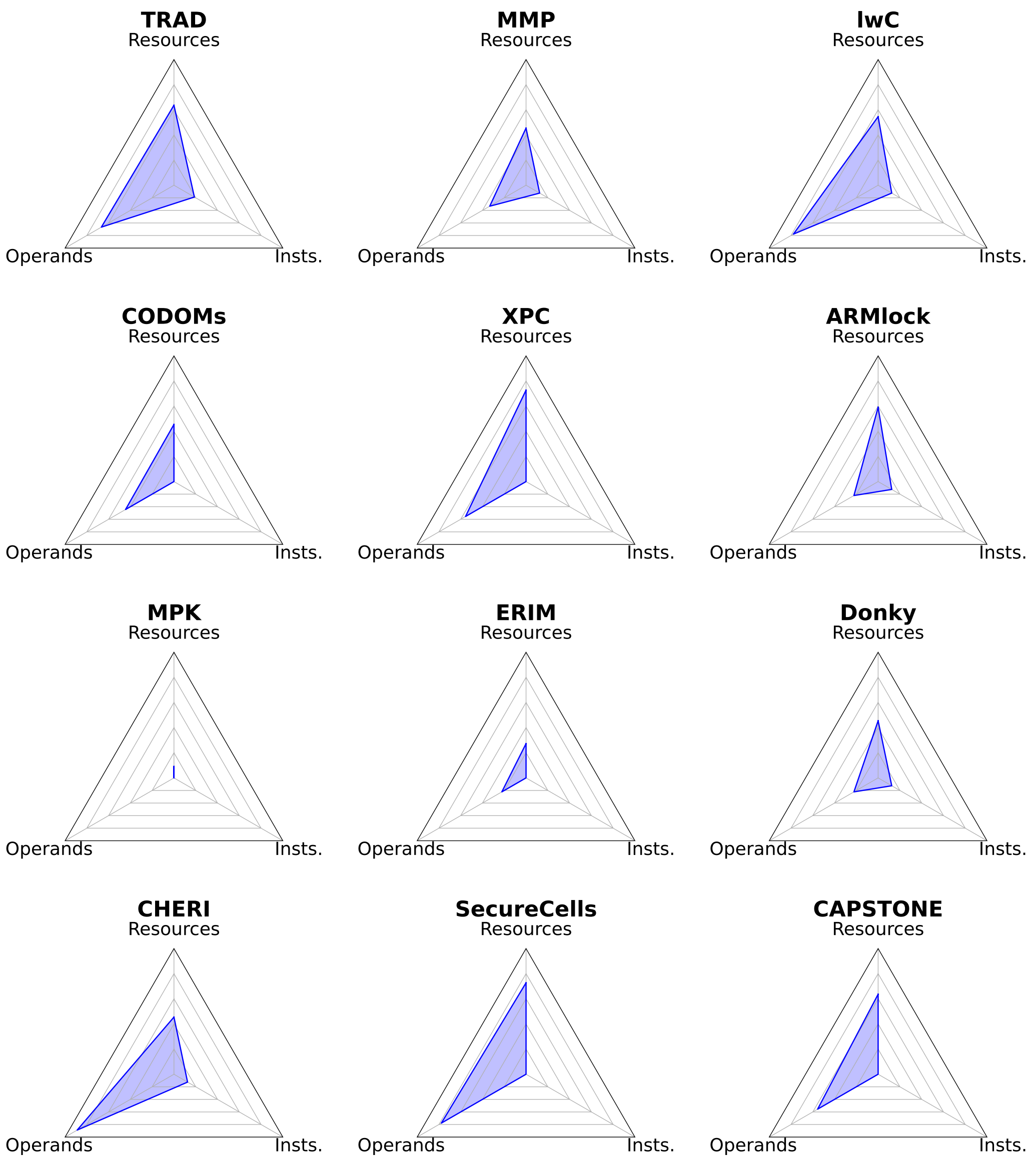


Figure 4.3: Scoring security properties of mechanisms.

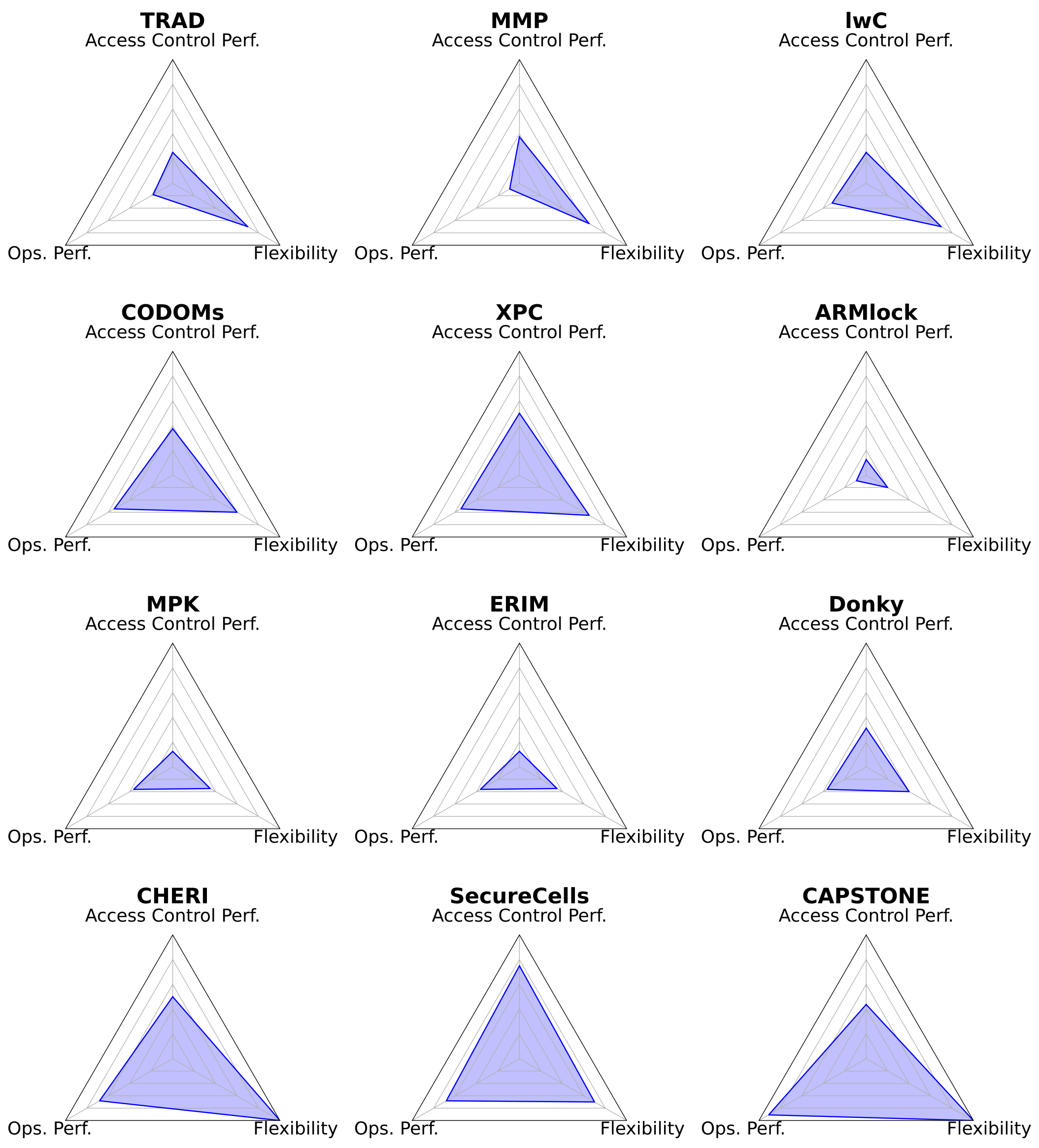


Figure 4.4: Scoring practicality and performance aspects of mechanisms.

regions (virtual memory areas in SecureCells). Systems with range-based permissions (MMP, CHERI, SecureCells and CAPSTONE) support memory ranges starting and ending at any byte (or word) boundary. Meanwhile, page tables also impose a page-sized alignment requirement for regions of memory. Fine-grained permissions allow developers to shrink the size of compartments, and better isolate components of their applications. However, the spatial granularity of permissions trades off security and performance as the mechanism is responsible for tracking and enforcing permissions for more regions as the granularity of permissions shrinks. Whereas modern browsers hold hundreds of millions of objects, they use millions of pages, and thousands of virtual memory areas. Most mechanisms also support an almost arbitrarily large number of memory regions. Key exceptions are mechanisms based on protection keys or page colors. ARMlock, based on ARM's memory domains feature, and MPK-based mechanisms support only 16 different colors, and Donky improves the limit slightly to support up to 1024 different colors.

Access control to virtual memory is a common feature across most of the mechanisms considered. Virtual memory permissions can prevent spatial memory safety violations, a major and relatively simple attack vector, from propagating beyond compartments. Traditional, OS-based mechanisms have a strong score for controls on memory access, since compartments are assigned separate processes which present the abstraction of isolated virtual memory spaces. Similar high scores are achieved by other research proposals with strong emphasis on security. However, commercial mechanisms trading off performance for security, such as Intel's MPK and mechanisms built on MPK, show lower scores here. Particularly, MPK only prevents illegal data access (read/write) to pages of a different color, protected by a register directly and arbitrarily writable by userspace. ARMlock, however, also checks code fetches. ERIM and Donky improve on MPK's design, adding integrity for compartment transitions and protection key updates, respectively. However, each of these mechanisms scores better than MPK, but still score poorly for virtual memory protection due to lacking execute permission checks and support for exclusive access. ERIM also lacks of support for context isolation, like MPK.

Support for access control to physical memory is scarce among the surveyed mechanisms, with only CAPSTONE providing capabilities for physical memory, and XPC providing a relay segment which the supervisor points to separate per-core physical memory regions. Permissions for physical memory alleviate the possibility of attackers leveraging aliasing in physical-to-virtual address translations to bypass memory protection and access prohibited physical memory. Practically, controlling permission to physical memory is inherently difficult as applications operate with virtual addresses and physical addresses are hidden from userspace and controlled by operating-system-managed translation tables. CAPSTONE reveals physical addresses to user applications, possibly simplifying attacks on physical memory like Rowhammer [89]. Further, a core designed to perform protection checks on physical addresses *after* translation stands to suffer a significant performance hit from the increased latency to access memory. RISC-V specifies Physical Memory Protection (PMP) registers, ranges of physical memory with specific addresses, but their current design is more suited for confidential computing than compartmentalization. PMP registers can only be modified by the machine-mode firmware, and are often sealed at boot time to predetermined processor-private memory regions. Instead, future mechanisms can focus on providing guarantees against aliasing for virtual-physical translations for data regions, preventing this class of attacks.

Access control to supervisor resources varies more between mechanisms. Assuming per-compartment system call filtering rules are implemented, the supervisor must be able to securely identify the calling compartment to perform the correct set of checks. Donky and ARMlock explicitly present system call filtering as part of the mechanism. Mechanisms where compartments switch using a system call (TRAD, MMP, lwC, CODOMs, ARMlock and CHERI) indirectly allow the supervisor to identify the caller. Linux, for example, stores a pointer to a process' supervisor metadata in a privileged register when the corresponding user process executes, allowing the supervisor to identify process on traps. Mechanisms which explicitly identify the executing compartment (XPC, SecureCells) enable the supervisor to directly identify the compartment. With Intel's MPK and ERIM, the supervisor may be able to indirectly identify the caller by the value of the permission-key register, if unique. Finally, compartments in CAPSTONE are implicit, relying on capabilities stored in registers and hardware calls for switching to compartments with sealed capabilities. The supervisor has no direct or indirect mechanism to identify the caller, since the caller's state is also sealed on traps. CAPSTONE, therefore, cannot implement system call filtering.

We notice that support for restricting per-compartment instructions or operations are a common shortcoming among most mechanisms. These restrictions are crucial to enable future systems offloading supervisor functionality to userspace, primarily for high-performance server applications, or for compartmentalizing supervisor kernels. These applications would benefit from limiting system management code with the corresponding privileges within individual compartments for each aspect of management (memory, per-device drivers, scheduling). Only traditional operating systems can selectively disable processor features, like the x87 floating-point unit, though this feature exists as a side-effect of power-saving features in the CPU or due to the requirement to disable faulty CPU functional units. We notice that researchers focus on memory isolation and build on a simplified system model with only computing elements (like cores) and memory. Future proposals should also consider restrictions to the many accessible system-wide resources on commercial CPUs.

Mechanisms are also designed to support different classes of applications. Traditional applications (specially desktop/mobile applications) are compartmentalized with processes reusing an existing operating system abstraction for isolating users and applications running on a shared machine, and assume the performance costs of using a mechanism not designed for fine-grained compartmentalization. Meanwhile, applications can also benefit from any additional security for data accesses introduced by commercially available mechanisms (MPK, ARM's memory domains). MPK deliberately enables userspace modification of permissions, as opposed to the supervisor (ARMlock), for improved performance. However, high-performance software like supervisor kernels and some server software require secure but high-performance mechanisms and may be better supported by research proposals. Compartments mapped to processes (TRAD, lwC) use separate address spaces and permissions stored in page-table entries to securely isolate memory. Consequently, page-table entries across compartments share capacity within an already size-constrained microarchitectural buffer. Commercial compartmentalization based on Intel's MPK also use page-table entries for storing translations and page colors, but rely on permissions stored in the PKRU register, slightly relieving TLB pressure. ARMlock slightly alleviates this pressure by storing colors for 2MiB hugepages. Due to the reliance on TLBs for permission metadata, mechanisms based around traditional computer architectures and page tables score poorly on performance for access control to memory. Two approaches to alleviate this pressure exist. First, MMP separates permissions from translations, implementing a separate permissions table and caching buffer. Second, MMP and SecureCells include support for storing permissions to variable-sized regions, ideal for separating permissions for virtual memory areas ubiquitous in modern applications. Technically, CODOMs removes TLB pressure by only allowing a single compartment access to each page, though this approach comes with flexibility constraints. Finally, capability systems move the microarchitectural bottleneck from the TLB to the core's data caches, by moving permissions into capabilities held in memory. Single-cycle memory access checks in capability systems (CHERI, CAPSTONE) relies on having capabilities available in registers, or within the L1 data cache. Fortuitously, core-private L1 data caches scale slightly better than TLBs. Capability systems also have a larger memory footprint due to inflating the size of each pointer held in memory. The memory system of capability systems also have to track a tag bit marking capabilities in memory, with these systems incurring a significant cost for the additional memory required and for redesigning the entire memory hierarchy. We can see that capability-based systems support fine-grained memory permissions with easy permission transfer between compartments, but offer these at a performance cost.

Surveyed mechanisms do not sufficiently solve the challenge of supporting efficient revocation. Each memory access with CAPSTONE, which is designed to support efficient capability revocation, also incurs an additional memory access to check the capability's revocation status. While the authors claim that the revocation check can be performed in parallel with data accesses, a Spectre-safe implementation requires permission checks prior to data accesses, putting the revocation check on the critical path to each memory access. The significantly increased latency to memory, due to the additional checks as well as increased L1-cache bandwidth required, will likely massively inflate average memory access times on CAPSTONE.

Switching compartments is another performance-critical operation and newer proposals have aimed at reducing the latency of switches to sub-microsecond scales requiring hundreds (CHERI, Donky) or even tens of cycles (SecureCells, XPC). Despite optimized fast paths, supervisor-controlled switches between compartments on commercial hardware only achieve relatively expensive switches in a few microseconds. Sub-microsecond switching time requires hardware support, as offered by some newer mechanisms. For a compartment switch on mechanisms with nanosecond-scale switches (XPC, SecureCells, CHERI, CAPSTONE), the costs of flushing the processor pipeline (to stop speculative side-channel attacks) and a full context switch (storing and restoring up to 32 registers in popular ISAs), become a major overhead. Microarchitectural optimizations will be crucial to approach the ideal compartment switch at function call latencies.

At compartment boundaries, passing permissions to memory regions holding arguments supports zero-copy computation on the same data between an application's compartments. Capabilities offer the fastest way to transfer permissions, by simply passing a capability to the target memory region during a cross-compartment call. Capabilities conveniently allow passing permissions to a complicated data structure fragmented throughout memory by passing a capability to the structure's entry point, such as a pointer to the head of a linked list. On capability systems, the target compartment can use the original capability but also load further capabilities stored in the transferred region, recursively, to access further elements of the data structure. However, this method of passing data must be used with care to prevent accidental permission leakage - application-level bugs causing an unintended capability from being transferred on cross compartment calls. Leaks can stem from the initial capability passed at the calling interface, or through any part of the complex data structures that may be referenced by that capability. On the other hand, systems with a centralized permissions table (as opposed to capabilities distributed through memory) offer the potential for faster permission transfers, as only a single permission needs to be changed for each transfer. Since permission tables are only accessible to the supervisor, the cost of the system call often dominates the cost of zero-copy permission transfers. Linux offers the `vmsplice` system call for zero-copy transfers. SecureCells also offers a hardware permission transfers costing a few hundred cycles. Unilateral permission transfers also open the door to confused-deputy data/code attacks. Traditional OS-based compartmentalization, that requires the receiving compartment to use a system call to receive data, and SecureCells, which requires receivers to use a special instruction, satisfy this criterion for security.

4.6 Discussion

Compartmentalization relies on strong isolation policies implemented on capable mechanisms. However, none of the mechanisms we compare in this chapter achieved strong scores in all surveyed restrictions, calling for future research on improved mechanisms. In this section, we will discuss opportunities we identified for future mechanisms to introduce comprehensive security, and discuss the limitations of our approach.

4.6.1 Opportunities

This SoK explores the main areas where future compartmentalization mechanisms can add value. We find that performance is the primary shortcoming for isolation with traditional processes. Newer, lighter supervisor abstractions [23, 76] offer a promising improvements. These mechanisms benefit from architectural improvement towards faster traps, faster hardware coherence for translations and permissions cached in TLBs across manycore systems, and better fast paths in supervisor software for compartmentalization operations. On the other end of the spectrum, lightweight security features like MPK suffer from severe security limitations. Further limitations on which and how compartments can modify the permissions register can help improve MPK's security.

Research proposals for future mechanisms can also improve on the operations described below. Mechanisms lack support for isolating instruction execution (specially traditionally privileged instructions) to specific compartments. This feature will allow supervisors to offload functionality to userspace libraries, eliminating the costs for system calls, and to even compartmentalize the supervisor itself. Low-overhead exclusive access to physical memory can provide stronger security than exclusive access to virtual memory. Many mechanisms (SecureCells) only give exclusive access to virtual memory, which can suffer from virtual-to-physical aliasing, and may be bypassed. CAPSTONE provides this feature, but might suffer from having to expose physical addresses to userspace, and increased costs for normal memory accesses. An intermediate address space used as a non-aliasing layer of indirection may offer a solution. We currently lack a mechanism with support for low-overhead revocation for transferred permissions without making normal memory accesses more expensive. Mechanisms might also explore the implications of introducing the concept of ownership for data, and the opportunities this feature provides for efficient revocation.

4.6.2 Limitations of our Methodology

In this section, we discuss some of the limitations of our comparison, and suggest that compartmentalization mechanism proposals require more standardized evaluation to enable a fair comparison.

Benchmark Suite for Compartmentalization. Publications for compartmentalization mechanisms vary greatly in the choice of benchmark software used to evaluate their performance. Most mechanisms present microbenchmarks measuring compartment switching latency with a few papers notably lacking this measurement (e.g., CAPSTONE). Even among mechanisms measuring switching, the exact methodology varies. CHERI, for example, includes the cost of context switching, whereas XPC and SecureCells do not. The Apache and Nginx webservers have been used by a few papers, whereas others have used SPEC, Binder, V8, NaCl, memcached, and a variety of systems libraries (XML parsing, Mbed TLS, SQLite, zlib). These applications vary greatly in their characteristics, and some require no compartmentalization whatsoever. To enable future research into compartmentalization mechanisms, security researchers need a comprehensive suite of common benchmarks spanning the varying use cases for compartmentalization. The benchmarks must include desktop (e.g., browsers) and server (e.g., Apache, Nginx, memcached) workloads spanning a range of performance requirements, from millisecond-scale compartment execution to nanosecond-scale execution for library isolation to highlight possible design trade-offs for mechanisms.

Evaluation Testbench. We were unable to reproduce and test the various mechanisms due to the wide range of experimental designs, and their different implementations. Only lwC and ERIM can be run on commercial machines without hardware changes. Other proposals have presented varying implementations based on microarchitectural simulators (geM5), experimental silicon (CHERI), and FPGA RTL for prototypes. With each mechanism requiring a highly customized setup, reproducibility is limited. We hope that future proposals will standardize the evaluation setup and implementation to allow comparisons with other mechanisms. Due to the complexity of creating RTL prototypes, we propose the use of full system simulator (like geM5) models.

Scoring mechanisms. This thesis uses a scoring scheme to compare mechanisms quantitatively, as described in Section 4.5. Our comparison gives equal weight to mechanisms for implementing orthogonal checks for instruction fetches and for microarchitecture management instructions. The lack of execute-permission checks makes code injection trivial whereas the lack of restrictions on flushing the data cache enables side-channel attacks leaking data across compartments. A better comparison would assign weights to security features based on how effective they were at preventing attacks. The lack of a standardized set of bugs and exploits hinders the systematic assignment of weights.

4.7 Summary

Compartmentalization is a crucial mitigation for systems security, and a steady stream of mechanisms has emerged to support applications requiring isolation between untrusted components. To align the wildly varying mechanism designs emerging from the variety of design requirements and enable a principled comparison of mechanisms along common axes, this chapter presents a systematization of the restrictions required by mechanisms to mitigate cross-compartment attack vectors, and their performance and flexibility characteristics. This systematization allows us to trade-offs present in existing mechanisms, and to highlight common shortcomings. Finally, we offer sketches for how future mechanisms can more comprehensively guarantee security by addressing these shortcomings.

Chapter 5

Future Work

The improvements proposed in this thesis are just the first step towards securing modern sytems. Further effort is required to both enable developers to implement and adopt the proposed interfaces, and to keep these interfaces concurrent with ever-emerging threats. Developers also need to adopt our proposed interfaces to their target systems. Future research can investigate how emerging hardware features can help improve our implementations of the proposed interfaces. Finally, we describe other improvements to the interfaces investigated in this thesis, and potential avenues to realize these improvements.

5.1 Beyond Midas: Double-fetch Protection and More

While Midas provides systematic double-fetch protection to the kernel interface, here we propose future use cases for Midas and suggest further improvements to its protection and performance guarantees.

Besides providing double-fetch protection, Midas can be used as a sanitizer for finding double-fetch bugs during dynamic testing of a OS kernel and for enabling deep-argument checks for system call filters. For example, Midas can be used as a sanitizer with Syzkaller [133] while fuzzing the kernel interface. System call filters like SECCOMP add hooks to checks which validate system call arguments before a system call executes, restricting processes from using potentially harmful system calls. Midas can help fix a major shortcoming for modern system call filtering mechanisms: the lack of deep-argument inspection. Essentially, the filters cannot “check” any by-reference arguments stored in user memory without creating a TOCTTOU instance since these arguments will later be “used” by the kernel. The introduced TOCTTOU bug makes the filtering check ineffective. Midas, however, can be used to eliminate the TOCTTOU condition. When running with Midas, the kernel would transparently create a snapshot when system call filters inspect userspace objects, with the same snapshots serving data during the system call itself. Midas, therefore, can make system call filtering more capable and enable better isolation of userspace processes.

Midas will benefit from hardware protection for physical memory ranges for preserving snapshots of accessed memory objects. Particularly, Midas can reduce the cost of creating snapshots, which potentially incurs the cost of changing page table permissions across each virtual address space where an accessed page is mapped, and the cost of corresponding TLB shootdowns. Virtual page-based snapshotting also introduces false-positives due to false-sharing of objects on the same page, as described in Section 2.5. New and upcoming hardware architectures can provide the physical memory protection mechanisms required to implement snapshotting at low cost. First, RISC-V's Physical Memory Protection (PMP) mechanism is ideal for protecting a small number of physical memory ranges, only requiring updates to per-core protection registers optionally accessible by the privileged supervisor. PMPs are also flexible, allowing specifying protections for naturally aligned power-of-2 address ranges enabling supervisors to protect memory at a smaller granularity than pages. However, PMP updates must be propagated across cores in order to protect against cross-core attacks. Midas can also benefit from architectures like Midgard, where a system wide page table stores permissions to physical frames applicable across all virtual address spaces. Essentially, Midas can leverage Midgard's backside-page table entries to make a single permission change for a snapshot. Finally, Midas can leverage proposed linear capability systems like Capstone [153]. Capstone guarantees that only a single active capability to an object in the physical address space exists at any point in time. During a system call, the kernel possessing a linear capability to a user object can be assured that userspace cannot simultaneously hold a capability to the same object, guaranteeing that the object remains unmodified.

Considering the increasing use of on- and off-chip accelerators accessible to userspace applications, future work can extend Midas' protection to include the threat of snapshot corruption through direct memory access (DMA) from accelerators. One strategy would be to use existing protection mechanisms for DMA, including I/O Memory Management Units (IOMMUs) to protect snapshots. Linux's reverse mapping must also be modified to include mappings for DMA-capable devices.

Hypervisor interfaces also require double-fetch protection, and can benefit from Midas' protection. We leave the investigation of double-fetch protection of hypervisors to future research.

5.2 Widespread Compartmentalization with SecureCells

Following the eras of battling memory corruption, code injection and control-flow attacks, we believe that the next frontier of defenses lies with compartmentalization. Particularly, this thesis highlights the need for isolation within application components sharing a virtual memory space, and how the architectural interface for virtual memory is crucial for supporting isolation of memory accesses.

Ushering the era of well-compartmentalized applications requires secure policies for decomposing applications into compartments, secure and performant mechanisms for implementing the policies, and developer-friendly toolchains for supporting the process. In this thesis, we propose SecureCells as a supporting mechanism and present a systematization of knowledge of related mechanisms in Chapter 4. Future work in compartmentalization is required to address the shortcomings of SecureCells, implement SecureCells as a back-end supporting existing compartmentalization toolchains and improve software support for SecureCells. We believe that parallel efforts into developing secure compartmentalization policies for existing programs plays an equally crucial part in realizing our vision of a compartmentalized future.

Our comparison of compartmentalization mechanisms shows potential for improving SecureCells. Particularly, SecureCells focuses on the issue of memory isolation. Future work can add supervisor support for isolating kernel resources to SecureCells, perhaps involving system call filtering. The granularity of passing data between compartments with zero-copy with SecureCells is a virtual memory area, which can be unsuitable for applications where compartments communicate with arguments held in fragmented data structures which do not map to a VMA. Improvements can consider integrating CHERI's capability model for fine-grained access to memory objects with SecureCells to support cheap permission granting for complicated and fragmented data structures. Finally, more research is required to investigate the requirement for revocation of granted permissions, and the addition of this feature to SecureCells. SecureCells' cell description table, for example, can be used to encode a VMA's owner, opening the potential for the implementation of an unprivileged instruction allowing a VMA's owner to revoke another compartment's permissions to that memory region.

Software support for emerging architectures is crucial for their adoption. We envision that future efforts will improve operating system support for SecureCells, along with porting of the requisite compiler toolchains and libraries. Automated or compiler-generated compartmentalization will be crucial to compartmentalization efforts, and SecureCells can be used as a backend to tools like Enclosures [44].

Finally, compartmentalization can improve microarchitectural security for applications by helping the hardware identify untrusted program components and limiting microarchitectural interactions or leakage channels between their compartments. Hardware mitigations for side channels, typically adopted at transitions at crucial boundaries such as during system calls, can also be introduced when transitioning between compartment. Hardware or software partitioning techniques typically used to microarchitecturally isolate untrusted code in separate processes or privilege levels can also bring benefits to isolating compartments within an application. However, applications will typically involve more compartment switches than system calls, and the performance overheads of introducing intra-application microarchitectural isolation remains to be determined and is left for future work.

Chapter 6

Conclusion

Systems security is a continuous arms race between attackers and defenses. This thesis highlights two key interfaces in modern systems which lag behind in this race, and proposes interfaces designed to mitigate attacks exploiting the user-kernel system call interface, and the userspace virtual memory interface.

Midas focuses on the vulnerability introduced by double-fetch bugs in privileged software like OS kernels, and describes a systematic mitigation mechanism to block this attack vector. Midas identifies the implicit assumption underlying the existing system call interface's design and secures the interface by elevating the assumption to an explicit guarantee. This thesis also shows a practical implementation of Midas' design for a popular OS running on commercial off-the-shelf hardware In general, Midas highlights the security implications of implicit assumptions made by system designers, and how changing computing systems can invalidate assumptions. Comprehensive defense, instead, can be achieved through well-defined and explicit security properties enforced across interfaces. Midas guarantees a security invariant preserving the values of userspace data objects accessed during system calls, ensuring that all reads to the same objects return the same value.

SecureCells investigates the mechanisms supporting userspace application compartmentalization and makes the case for a mechanism enabling secure, performant and flexible intra-address space compartmentalization. SecureCells identifies the requirements supporting the three key application objectives, and proposes a mechanism designed to support widespread application compartmentalization. SecureCells provides strong isolation between compartments for data accesses based on permissions stored in a permissions table storing per-compartment per-memory region permissions. SecureCells introduces unprivileged instructions for implementing frequent operations in compartmentalized applications, like inter-compartment control flow and zero-copy permission transfers at sub-microsecond time scales while also implementing strict security conditions. This thesis also describes our full-system prototype for SecureCells based on modified RISC-V

RocketChip cores, the secure seL4 microkernel OS and userspace benchmarks used for evaluating our design. We are optimistic that SecureCells will add momentum to the ongoing push towards compartmentalization, improving interfaces within userspace applications to reflect the varying trust relationships between application components. This thesis also contributes a survey comparing state-of-the-art and commercially used compartmentalization mechanisms. This survey explores the design space across reviewed mechanisms and highlight the key security and performance ideals which mechanisms strive to provide.

To support the ideal of open science, the code and other artifacts supporting this thesis are available openly and freely. Detailed documentation for Midas is maintained on the project's website `https://hexhive.epfl.ch/midas`. The evaluation results for Midas were submitted for artifact evaluation, earning badges qualifying the artifact as "Available", "Functional" and "Reproduced". SecureCells' prototype, benchmarks, supporting infrastructure and requisite documentation are also available at `https://hexhive.epfl.ch/securecells`.

Appendices

Appendix A

Midas Artifact

Detailed documentation for Midas is available on the project website `https://hexhive.epfl.ch/midas`.

A.1 Artifact Appendix

For Midas, we present an artifact including the source code and binaries for the prototype based on Linux, an exploit which demonstrate that Midas mitigates a real CVE, and benchmarks for evaluating Midas' performance, and scripts which simplify the process. In the following sections, we describe the artifact, its requirements and how to run it, and what the expected results are.

A.1.1 Description

The primary artifact for this project is the code implementing Midas on the Linux kernel (v5.11), available on GitHub. We also provide a disk image suitable for recreating experiments from this project, containing the kernel as both source code and as compiled binaries. The disk image contains the CVE exploit used to test correctness in the project, all benchmarks evaluated in this project, and scripts to run these. This image allows recreation of all empirical evidence presented in the project's evaluation. Finally, we provide further information on the project website including a detailed description of the artifact, its contents, how to run it and expected outputs.

- Source code: `https://github.com/HexHive/midas`
- Disk image: `https://zenodo.org/record/5753026`

- Project website: `https://hexhive.epfl.ch/midas`

Hardware Dependencies

You can run the disk image within a QEMU virtual machine to test functionality. The host machine requires around 100GiB free disk space and at least 8GiB memory. You should run the disk image on a real machine for performance tests. Our Midas prototype supports machines with 64-bit x86 processors, and the results in the project were obtained on a machine with an Intel i7-9700 CPU. Further, the real machine requires an empty 1TiB disk, and a EUFI-enabled motherboard. In both setups, a SSD is preferred for storage, as it leads to faster compilation should you choose to re-compile the kernel. Evaluating the Nginx benchmark requires a second, networked machine to act as a load generator.

Software Dependencies

Running the Midas disk image requires a guest operating system which supports running QEMU. The image was tested on QEMU version `4.2.1` on a machine running Ubuntu 20.04 with Linux kernel version `5.4.0-88-generic`. Other virtualization software should also be supported, but the instructions focus on QEMU. Running the disk image on real hardware requires no special software support, apart from a tool to write the image to a disk. On Linux, we can use `dd`.

A.1.2 Installation

The installation procedure includes downloading and uncompressing the provided compressed disk image, then either running a VM directly from this image, or by writing the image to a disk and booting from it.

On Linux, the following command extracts the image.

```
  pv ae.img.xz | unxz -T <num threads> > ae.img
```

The uncompressed disk image can then either be run with QEMU, or written to a real disk. To run with QEMU, an example command is shown below.

```
qemu-system-x86_64                    \
  -m 4G                               \
  -cpu host                           \
-machine type=q35,accel=kvm         \
-smp 4                              \
-drive format=raw,file=ae.img       \
-display default                    \
-vga virtio                         \
-show-cursor                        \
-bios /usr/share/ovmf/OVMF.fd       \
-net user,hostfwd=tcp::2222-:22     \
-net nic
```

To run on real hardware, copy the image to a real disk using the command shown below, then install into the machine and start it.

```
dd if=ae.img of=/dev/<disk> bs=100M
```

A.1.3 Experiment Workflow

The experimental workflow compares the modified Midas kernel with the baseline Linux kernel. Detailed steps are available on the website at `https://hexhive.epfl.ch/midas/docs/ae.html`. You can validate the artifact by executing the following steps:

- Check that the code modifications described in the project correspond to the code.
- Compile the code to re-create the kernel binary.
- Run a script to check that a CVE exploit is mitigated, as claimed in the project.
- Run scripts to execute the benchmarks presented in the project, to verify their reported performance.

For the CVE exploitation test, the dmesg output must be checked to ensure that Midas prevents exploitation. For the performance experiments, the results must be compiled and compared to get the Midas' relative performance. The general workflow is:

- boot with the correct kernel (baseline or Midas),
- run the script for the benchmark/CVE exploit,
- reboot with the other kernel, and
- run the same script again.

A.1.4 Expected Results

Midas is evaluated to demonstrate effective mitigation of double-fetch bugs with low overhead. The artifact enables you to verify this claim, that the prototype provides the claimed protection and that it performs as claimed. We demonstrate the first property by including checks in the kernel and running an exploit for CVE-2016-6516 to demonstrate its mitigation. The remaining benchmarks measure performance, either as operations per second or as time taken to finish each operation. Below, we describe how to interpret the outputs of running the exploit and benchmarks.

Midas protects the kernel against double-fetch bugs, and in particular mitigates an exploit for CVE-2016-6516. In our prototype, you will execute the exploit with and without Midas' protections. When run with the baseline kernel, the exploit is triggered, and the string `"Triggered bug: CVE-2016-6516!"` will be printed to `dmesg` output. With the Midas kernel, the string is never printed.

We also run kernel-intensive benchmarks which demonstrate that Midas has a low runtime overhead. Our artifact also contains the performance benchmarks used for testing Midas' performance. The benchmarks must be run separately with both the baseline and Midas kernel. We include a script to plot the relative performance vs. the baseline kernel. Midas' performance is strongly dependent on the CPU used for evaluation, and exact performance values can vary significantly. However, we expect the trends of performance across benchmarks to roughly follow the following limits.

- Microbenchmarks see results in line with Chapter 2.
- NPB benchmarks experience 0-5% overhead, and should follow the numbers from Chapter 2.
- PTS benchmarks - openssl, git, pybench, redis see an overhead <1%.
- PTS benchmarks - apache sees a overhead < 10-15%.
- PTS benchmarks - IPC benchmark sees overhead < 5%.
- Nginx shows a constant overhead as request size changes, until the network link is saturated.

The setup for breaking down Midas' overhead is complicated, and omitted from this artifact.

A.1.5 Artifact meta-information

- **Program:** NASA Parallel Benchmarks (NPB), Phoronix Test Suite (PTS), Nginx, the Linux kernel, and exploits for CVE-2016-6516. All benchmarks and code are publicly available, and are installed in the provided disk image.

- **Binaries:** The disk image provides the compiled Linux kernel (v5.11) with and without Midas' protections.
- **Hardware:** For functionality evaluation, one machine with 100GiB free disk space, and QEMU (version 4.2). For results reproduction, one machine with modern Intel x86 CPU, and a free 1TiB disk. In both setups, a SSD is preferred.
- **Run-time state:** The disk image includes a program for fixing CPU frequency, eliminating run-time variance. This only works on native hardware, not QEMU.
- **Metrics:** NPB workloads report execution rate. PTS workloads report either execution time or operation rate. Nginx reports both request rate and throughput.
- **Output:** Most benchmarks and tests output to a console.
- **Experiments:** Experiments have been prepared within the disk image, and can be run using provided scripts.
- **How much time is needed to prepare workflow (approximately)?:** 3-4 hours, on a machine with an SSD.
- **How much time is needed to complete experiments (approximately)?:** For performance evaluation, approx. 8 hours.
- **Publicly available?:** All code is publicly available.
- **Code license:** GPL v2.0
- **Archived?:** DOI 10.5281/zenodo.5753026 available at `https://zenodo.org/record/5753026`.

Appendix B

SecureCells Details

Detailed documentation for SecureCells' prototype and code artifacts is available on the project website `https://hexhive.epfl.ch/securecells`.

B.1 Memory layout of the unified PTable-GTable

Figure B.1 shows the detailed implementation of the unified PTable-GTable in our prototype SecureCells implementation.

The table contains a sorted list of cell descriptors, including a metadata "cell" used for storing its sizing parameters. As described in Section 3.4.4, each cell descriptor stores virtual and physical frame numbers uniquely identifying a VMA, as well as a validity flag to track the cell's current validity. The metadata cell tracks the number of allocated cells (N), the number of SDs (M), and sizing factors T (upper bound on cell count) and R (upper bound on SD count). When software requires additional SDs or cells, it must request the supervisor via a system call. If the request overflows the bounds imposed by factors R and T, the supervisor must resize this table as required. The cell descriptor list is followed by the PTable, and then by the GTable. This layout assumes, and is optimized, for a cache line size of 64 bytes.

The size of parameters M and N, holding the current number of cells and SDs, in the metadata cell is 32 bits each. The maximum supported number of cells in our implementation, therefore, is 2^{32}. The number of SDs, however, are limited by the grant table layout. The 32-bit grant table entries use 29 bits for grantee SD identifiers, thereby limiting the maximum number of SDs to 2^{29}.

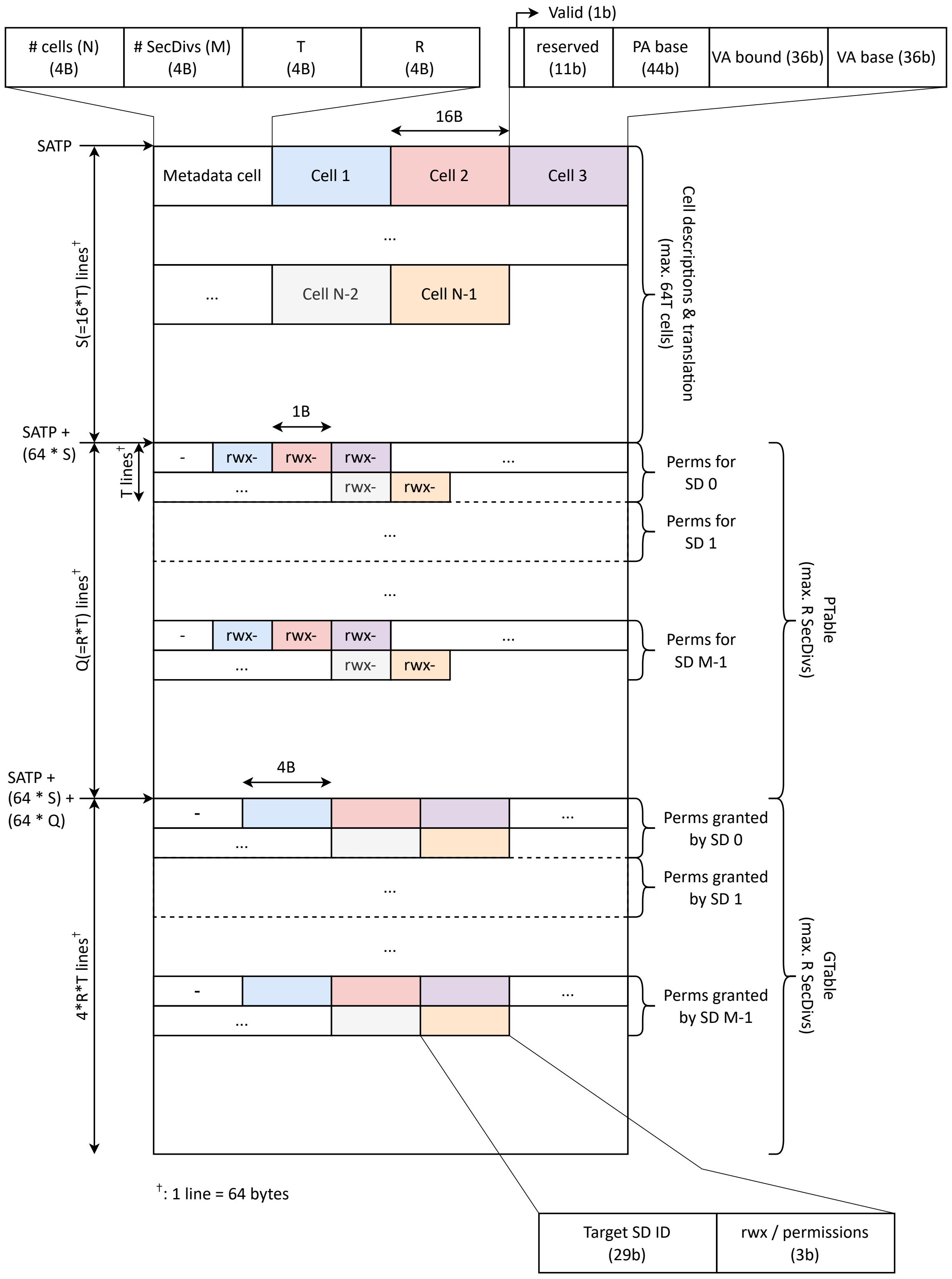


Figure B.1: Layout of SecureCells' unified PTable-GTable.

B.2 Justification for Table 3.1

Obj. O1a. MPK, ERIM and Donky do not check permissions for instruction fetches, simplifying code injection. Under our threat model, an attacker can inject `wrpkru` instructions to corrupt permissions.

Obj. O1b. Through code injection, call gates in MPK and ERIM can be bypassed.

Obj. O1c. CODOM requires migrating threads without context isolation. MPK, ERIM and Donky rely on call gates if context isolation is desired. However, MPK and ERIM cannot enforce call gates under our threat model. Donky gives no mechanism for a compartment to restore its state without trusting general-purpose registers. Further, Donky cannot adopt a SecureCells-like software approach because a compartment has no way to identify itself.

Obj. O1d. CHERI allows one compartment to unilaterally send a capability to another compartment, unchecked by the TCB and unacknowledged by the receiver.

Obj. O1e. No mechanism except XPC considers the challenge of exclusive access.

Obj. O1f. A compartment in MPK and ERIM cannot check the value of the `pkru` register for another compartment, hindering audits. Cross-core `pkru` reads are not possible. CHERI requires an expensive full memory scan for capabilities to perform an audit.

Obj. O2a. Page-table based translation and permission checking encounter TLB-reach limits leading to multi-cycle common case access verification for many widely-used programs including `memcached`. The mechanisms relying on such page tables for either translation or permission checking fail this requirement.

Obj. O2b. Supervisor-mediated cross-compartment calls in UNIX-like OSs, Mondrian, lwC and CHERI require 100s or 1000s of cycles to complete.

Obj. O2c. Supervisor-mediated permission transfers are slow (UNIX, MMP, lwC). MMP proposes the use of redundant mappings with different permissions to implement a form of zero-copy transfer which is not generic. CODOM does not really support permission transfers. XPC restricts permission transfer to a single relay segment.

Obj. O3a. CODOM identifies the executing compartment by the instruction pointer, limiting the flexibility to share code/data regions between compartments.

Obj. O3b. UNIX, MMP, lwC, XPC and CHERI cannot eliminate context switching when a permissive policy allows migrating threading between compartments.

B.3 Existing mechanisms with SecureCells

Many existing performance or security mechanisms can be integrated with SecureCells, either unmodified or with modifications described in this section.

Physical Memory Protections. SecureCells enforces permissions on the virtual address space, and is therefore trivially compatible with physical memory protection schemes including RISC-V's Physical Memory Protection (PMP) mechanism, processor reserved memory for Intel's SGX and vendor-specific protections like Qualcomm's XPU [72]. These mechanisms will apply to the physical address output by SecureCells' MMU after PTable access control checks.

Pointer authentication and capabilities. ARM's pointer authentication code (PAC) feature and CHERI's capabilities improve memory safety by protecting pointers from illegal modifications (overwriting when stored in memory and out-of-bound increment respectively). Both mechanisms are orthogonal to, and can integrate with SecureCells, which checks accesses against PTable permissions when the pointers protected by these mechanisms are finally dereferenced, providing another layer of protection against attacks like PACMAN [105].

Hardware and Software Control Flow Integrity. Hardware (e.g., Intel CET) and software (e.g., LLVM-CFI) control-flow protections can integrate with SecureCells, improving intra-compartment control-flow protection to complement SecureCells' inter-compartment call gates (`SDEntry`). CET can continue to check indirect call targets for `endbr` instructions. LLVM's and other fine-grained CFI pointer checks are implemented in software, orthogonal to hardware control flow checks.

Page-based mechanisms. By itself, SecureCells restricts popular mechanisms (e.g., guard pages, swapping) operating on pages and page tables since translations and protections are tracked at cell granularity. However, SecureCells can be integrated with upcoming intermediate address-space systems like Midgard re-enabling programmers to implement these crucial features. Midgard couples SecureCells-like range-based translation at the core with a second level of page-granularity translations at the backside of the last-level cache. Guard pages and swapping can both be implemented by unmapping the requisite pages in the backside translation.

B.4 SecureCells Implementation Trade-Offs

SecureCells permits a range of implementations scaling from simple microcontrollers with firmware emulation for added userspace instructions to server grade processors with microcode or hardware implementations. In this section, we describe the trade-offs and justify our implementation in Section 3.4.4.

Firmware. On the simplest side of the spectrum, instructions can be emulated by firmware using trap-and-emulate. Firmware is programmable code which runs in a privileged execution mode and uses native ISA instructions. SecureCells' instructions will trap into firmware, and be dispatched to the emulation code. Firmware implementations are cheap, requiring no additional hardware, but slower than alternate implementations. For the simple RISC-V RocketChip microcontroller, we choose firmware emulation for permission transfer instructions. Note that the firmware can also forward traps to be emulated by either the supervisor or even a privileged userspace library. However, the additional security risk of emulation by less trusted software risk and the overhead of forwarding traps makes such implementations less attractive.

Hardware. Alternatively, instructions can be implemented in hardware with finite-state machine circuits. While this design option implies better performance, designing complex hardware comes with silicon and power costs and substantial complexity. Hardware bug fixes incur the significant cost of the tape-out process. Server and desktop processors generally include beefy cores with large silicon area, where hardware implementations may match the processor's targeted performance. We implement the crucial `SDSwitch` instruction in hardware to reap the performance advantage, and because of the simplicity of its design.

Microcode. A third option, microcode, is programmable code provided by the processor manufacturer, built from low level operations including ones not available through the ISA interface. When a instruction implemented in microcode is encountered, a microcode sequencer fetches microcode from an on-chip RAM and executes them in the pipeline. Microcode eliminates the cost of trapping and dispatch encountered in firmware emulation (77% of the latency of emulating `SCProt`), and can also leverage hardware-specific optimizations. Microcode is popular for implementing complicated instructions with high performance like SGX's `EENTER`/`EEXIT` instructions. Microcode also has the advantage of being programmable, and have been leveraged to fix processor errata and bugs. While the simple RocketChip lacks a microcode sequencer, we envision microcode to be ideal for implementing SecureCells' permission transfer instructions for high-performance processors.

B.5 SecureCells ISA definitions

Below, we describe the definitions of SecureCells' instructions used in our RISC-V prototype.

Virtual memory mode. SecureCells describes an custom virtual memory mode for RISC-V, using the `0xf` value for the `Mode` field.

Instruction encodings. SecureCells uses existing spaces left for custom extensions to RISC-V to implement the added unprivileged instructions. Our added instructions occupy the *custom-0* and *custom-1* spots in RISC-V's instruction encoding map. Table B.2 shows the encoding formats for all added instructions.

SecureCells instruction	Arguments	Output	Prototype instruction	Arguments Mapping
SDSwitch	addr, SD_{tgt}	return addr	`JALS`	addr = `pc` + `imm`, SD_{tgt} = `rs`
SDSwitch	addr, SD_{tgt}	return addr	`JALRS`	addr = `rs1`, SD_{tgt} = `rs2`
SDEntry			`ENTRY`	
SCProt	addr, perm		`PROT`	addr = `rs1`, perm = `rs2`
SCGrant	addr, SD_{tgt}, perm		`GRANT`	addr = `rs1`, SD_{tgt} = `rs2`, perm = `imm`
SCRecv	addr, SD_{src}, perm		`RECV`	addr = `rs1`, SD_{src} = `rs2`, perm = `imm`
SCTfer	addr, SD_{tgt}, perm		`TFER`	addr = `rs1`, SD_{tgt} = `rs2`, perm = `imm`
SCReval	addr, perm		`REVAL`	addr = `rs1`, perm = `rs2`
SCInval	addr		`INVAL`	addr = `rs1`
SCExcl	addr, perm	True/False	`EXCL`	addr = `rs1`, perm = `rs2`

Table B.1: Mapping SecureCells instructions and arguments to the RISC-V prototype instructions. Instructions returning values write to the `rd` register

Instruction Fields						Type
`func7`	`rs2`	`rs1`	`func3`	`rd`	`opcode`	R-type
`imm[11:0]`		`rs1`	`func3`	`rd`	`opcode`	I-type
`imm[11:5]`	`rs2`	`rs1`	`func3`	`imm[4:0]`	`opcode`	S-type
`imm[20|10:1|11|19:12]`				`rd`	`opcode`	J-type

Encodings						Instruction
`imm[20:1]`				`rs/rd`	0101011	`JALS(rs, imm) = rd`
0000000	`rs2`	`rs1`	000	`rd`	0001011	`JALRS(rs1, rs2) = rd`
000000000000		`rs1 = 0`	001	`rd = 0`	0001011	`ENTRY`
1000000	`rs2 = 0`	`rs1`	010	`rd = 0`	0001011	`INVAL(rs1)`
0000000	`rs2`	`rs1`	010	`rd = 0`	0001011	`REVAL(rs1, rs2)`
0000000	`rs2`	`rs1`	011	`rd = 0`	0001011	`PROT(rs1, rs2)`
`imm[11:5]`	`rs2`	`rs1`	100	`imm[4:0]`	0001011	`GRANT(rs1, rs2, imm)`
`imm[11:5]`	`rs2`	`rs1`	101	`imm[4:0]`	0001011	`TFER(rs1, rs2, imm)`
`imm[11:5]`	`rs2`	`rs1`	110	`imm[4:0]`	0001011	`RECV(rs1, rs2, imm)`
0000000	`rs2`	`rs1`	111	`rd`	0001011	`EXCL(rs1, rs2) = rd`

Table B.2: Instruction encodings for SecureCells' prototype. The first part shows the four instruction encoding types supported by RISC-V.

Exception details. SecureCells also adds new exceptions, corresponding to failed security checks while executing the added unprivileged instructions. Table B.3 lists the added exceptions, and the corresponding values for the `scause` and `stval` registers storing exception details.

B.6 Accelerating SecureCells Instruction Emulation

B.6.1 Generic Trap-and-Emulate Acceleration

The RISC-V architecture embraces the trap-and-emulate mechanism for maintaining software compatibility across a wide range of hardware. Essentially, beefier cores can implement complex instructions in hardware where leaner cores can rely on traps into the firmware which emulates the same instructions in software. SecureCells' prototype utilizes the trap-and-emulate mechanism for implementing permission modification and transfer instructions (`SCProt`, `SCGrant`, `SCRecv`, `SCTfer`, `SCReval`, `SCInval`, `SCExcl`), in line with the complexity of the RocketChip core which resembles an embedded processor. To accelerate the instruction emulation, our prototype also introduces a mechanism for accelerating the trap-and-emulate mechanism. The mechanism described below is generic, and not restricted to emulating SecureCells' instructions.

The naive trap-and-emulate mechanism relies on saving the core's entire general-purpose register (GPR) context to memory, to later read argument registers for the emulated instruction. Consequently, the trap-and-emulate mechanism is relatively costly, consuming almost a hundred cycles to just switch contexts. Since the firmware can only find the argument register after reading the trapping instruction, decoding the instruction in software and finding the argument register fields, saving every register is essential.

While illegal instruction traps can originate from actually illegal byte sequences in code, the trap-and-emulate mechanism applies to actually valid instruction encodings. These instructions are fetched, and pass through the decode stage, which deciphers the instruction's encoding and the argument register names before raising the illegal instruction trap. We realized that the hardware can, therefore, read and save the relevant argument registers to special system registers allowing the firmware to directly access register arguments. Moreover, traps cause a pipeline flush, allowing the decode stage to ensure that the most recent value of the architectural register is saved to the special registers, even in an out-of-order core. The firmware, therefore, can include fast paths for trap-and-emulate, only storing the registers overwritten by the firmware during emulation to save tens of cycles. The firmware also saves cycles used for entirely decoding instructions to find their argument registers. For instructions which can be emulated in tens or a hundred cycles, including the aforementioned SecureCells instructions, saving tens of cycles during trap entry and exit can account for a significant speedup of up to 70%.

Our FPGA prototype, therefore, implements five extra machine-mode control-and-status registers (CSRs) for handling instruction arguments for trapping instructions. Three read-only registers, `mtirs1`, `mtirs2` and `mtiimm` store the argument register values, and the immediate value as relevant to the trapping instruction. These registers can directly be read by the firmware during emulation. A fourth read-only register, `mtird` stores the name of the destination register, if required by the trapped instruction. A write-only register, `mtirdval` can be written to by the firmware during instruction emulation, representing the value generated by the instruction. When the trap returns, the `mret` instruction writes the returned value into the correct destination register. Instructions without an output will have the `mtird` register pointing to the `x0` register, which is hardware pinned to the value zero.

B.6.2 SecureCells-specific PTable-GTable Access Helpers

SecureCells' PTable structure is designed to specifically make lookups for cell descriptions and permissions straightforward. The addresses of permissions and descriptions can be generated with simple arithmetic involving addition and bit-shift operations. Our prototype exposes single instructions for generating addresses within the PTable and for checking permissions in the PTable, implemented as simple hardware arithmetic circuits. The four instructions are listed below.

- **Check cell** The `SCCCK` instruction resembles ARM's `AT` instruction [5], and allows the firmware to check whether a instruction has a valid translation by accessing the core's range-TLB.
- **Cell Address** Generates the address for the cell description for a cell with a specified index.
- **Permission Address** Generates the address for the permissions byte within the PTable for a specific cell and SD.
- **Grant Address** Generates the address for the grant table entry within the GTable for a specific cell and SD.

Instruction	**Exception Cause**	`scause` **code**	`stval`
EXCL	Illegal address	`_ILL_ADDR`	addr
EXCL	Illegal permissions	`_ILL_PERM`	(type $\ll$ 8) | perm $\text{type} = \begin{cases} 0 & \text{if perm not in RWX} \\ 1 & \text{if perm is zero} \\ 2 & \text{if SD}_{cur} \text{ has insufficient perms} \end{cases}$
EXCL	Cell description invalid	`_INV_CELL_STATE`	0 if cell is invalid
GRANT	Illegal address	`_ILL_ADDR`	addr
GRANT	Target too high	`_INV_SDID`	SD_{tgt}
GRANT	Illegal permissions	`_ILL_PERM`	(type $\ll$ 8) | perm $\text{type} = \begin{cases} 0 & \text{if perm not in RWX} \\ 1 & \text{if perm is zero} \\ 2 & \text{if SD}_{cur} \text{ has insufficient perms} \end{cases}$
GRANT	Cell description invalid	`_INV_CELL_STATE`	0 if cell is invalid
RECV	Illegal address	`_ILL_ADDR`	addr
RECV	Cell description invalid	`_INV_CELL_STATE`	0 if cell is invalid
RECV	Illegal permissions	`_ILL_PERM`	(type $\ll$ 8) | perm $\text{type} = \begin{cases} 0 & \text{if perm not in RWX} \\ 1 & \text{if perm is zero} \\ 3 & \text{if requested perms not subset of granted perms} \end{cases}$
RECV	Invalid SD	`_INV_SDID`	(type $\ll$ 32) | SD $\text{type} = \begin{cases} 0 & \text{if SD source too high or SD=SD}_{src} \\ 1 & \text{if grant SD != SD}_{cur} \text{ or SD=SD}_{cur} \end{cases}$
PROT	Illegal address	`_ILL_ADDR`	addr
PROT	Illegal permissions	`_ILL_PERM`	(type $\ll$ 8) | perm $\text{type} = \begin{cases} 0 & \text{if perm not in RWX} \\ 2 & \text{if SD}_{cur} \text{ has insufficient perms} \end{cases}$
PROT	Cell description invalid	`_INV_CELL_STATE`	0 if cell is invalid
TFER	Illegal address	`_ILL_ADDR`	addr
TFER	Target too high	`_INV_SDID`	SD_{tgt}
TFER	Illegal permissions	`_ILL_PERM`	(type $\ll$ 8) | perm $\text{type} = \begin{cases} 0 & \text{if perm not in RWX} \\ 1 & \text{if perm is zero} \\ 2 & \text{if SD}_{cur} \text{ has insufficient perms} \end{cases}$

Table B.3: Exception causes, `scause` and `stval` register values for SecureCells exceptions

Instruction	**Exception Cause**	`scause` **code**	`stval`
TFER	Cell description invalid	`_INV_CELL_STATE`	0 if cell is invalid
INVAL	Illegal address	`_ILL_ADDR`	addr
INVAL	Cell description invalid	`_INV_CELL_STATE`	type $\text{type} = \begin{cases} 0 & \text{if cell is already invalid} \\ 2 & \text{if other SD has non-zero perms} \\ 3 & \text{outstanding grants for cell} \end{cases}$
REVAL	Illegal address	`_ILL_ADDR`	addr
REVAL	Cell description invalid	`_INV_CELL_STATE`	1 if cell is already valid
REVAL	Illegal permissions	`_ILL_PERM`	(type $\ll$ 8) $\mid$ perm $\text{type} = \begin{cases} 0 & \text{if perm not in RWX} \\ 1 & \text{if perm is zero} \end{cases}$
JAL(R)S	Target too high	`_INV_SDID`	SD$_{tgt}$
JAL(R)S	Illegal target instruction	`_ILL_TGT`	Target addr

Table B.3: (cont.) Exception causes, `scause` code and `stval` register values for SecureCells exceptions

`scause` **code**	**Value**
`_ILL_ADDR`	0x18
`_ILL_PERM`	0x19
`_INV_SDID`	0x1a
`_INV_CELL_STATE`	0x1b
`_ILL_TGT`	0x1c

Table B.4: Mapping SecureCells exception codes to values

Appendix C

Details for SoK on Compartmentalization Mechanisms

C.1 Classification Data

Below, we present the data used for creating the scoring diagrams in Figure 4.3 and Figure 4.4. We include descriptions for each column, which is a separate scoring factor, and show how each mechanism scores on this feature. For each feature, the mechanism fully satisfying the feature, or achieving the best performance in a category gets a full circle. Partially-filled circles show the relative score assigned to a mechanism for a feature, compared to the max score for that feature. Red circles indicate a negative score, awarded when a mechanism has a negative contribution to a feature.

Mechanism	**loadstore**[1]	**exec**[2]	**excl**[3]	**ctx**[4]	**kernelid**[5]	**syscallfilter**[6]	**pmem**[7]	**callpairs**[8]	**gates**[9]	**permtxperm**[10]	**permtxtarget**[11]	**permtxrevok**[12]	**permtxbilateral**[13]	**syscalls**[14]	**permmod**[15]	**permrevoke**[16]	**memalloc**[17]	**uarch**[18]	**xcore**[19]	**sysregacc**[20]	**xcompcalls**[21]
TRAD	●	●	◑	●	◑	●	○	●	●	○	●	○	●	○	○	○	◑	●	○	○	○
MMP	●	●	○	●	◑	◑	○	◑	●	○	○	○	○	○	○	○	○	●	○	○	○
lwC	●	●	◑	●	◑	◑	○	◑	●	●	●	○	●	○	○	○	○	●	○	○	○
CODOMs	●	●	◑	○	◑	◑	○	●	○	◑	●	○	○	○	○	○	○	○	○	○	○
XPC	●	●	●	○	●	◑	◑	●	◑	○	●	●	○	○	○	○	○	○	○	○	○
ARMlock	●	●	◔	●	◑	●	○	○	●	○	○	○	○	●	○	○	○	○	○	○	○
MPK	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
ERIM	●	○	○	○	○	●	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○
Donky	●	○	○	●	◑	●	○	○	●	○	○	○	○	●	○	○	○	○	○	○	○
CHERI	●	●	○	●	◑	◑	○	●	●	●	●	●	○	○	○	○	○	●	○	○	○
SecureCells	●	●	●	●	●	◑	○	◑	●	●	●	○	●	○	○	○	○	○	○	○	○
CAPSTONE	●	●	●	●	○	○	●	○	●	●	○	●	○	○	○	○	○	○	○	○	○

[1] Access control for load/stores
[2] Access control for inst. fetch
[3] Support for exclusive access to memory
[4] Support for context isolation
[5] Support for supervisor to ID compartment using system call
[6] System call filtering support
[7] Physical memory access control
[8] Caller-callee pairs allowed for cross-compartment calls
[9] Fixed entry points to compartments, for call gates
[10] Permission transfers for memory
[11] Specify target for permission transfers for memory
[12] Revoke transferred permissions
[13] Permission transfers require bilateral involvement
[14] Per-compartment restriction for system calls
[15] Per-compartment restriction for perm. modification ops.
[16] Per-compartment restriction for perm. revocation ops.
[17] Per-compartment restriction for memory alloc. ops.
[18] Per-compartment restriction for microarch. mgmt. ops.
[19] Per-compartment restriction for cross-core ops.
[20] Per-compartment restriction for access to system registers
[21] Per-compartment restriction for cross-compartment calls

Table C.1: Scoring matrix for mechanisms for security guarantees

Mechanism	**memacctl**[1]	**excl**[2]	**memmgmt**[3]	**switching**[4]	**revocation**[5]	**permtx**[6]	**memmgmt**[7]	**regalign**[8]	**reggrain**[9]	**regcount**[10]	**permtxgrain**[11]
TRAD	○	◔	◑	○	○	◔	◔	◕	◑	●	◕
MMP	◕	○	◑	○	○	○	◔	●	◕	●	○
lwC	○	◔	◑	◔	○	◔	◑	◕	◑	●	◕
CODOMs	◔	◔	◑	●	○	●	○	◕	◑	●	◔
XPC	○	●	◑	●	○	●	○	◕	◑	●	◑
ARMlock	◔	○	○	○	○	○	◔	◔	◔	◔	○
MPK	◔	○	○	●	○	○	◔	◕	◑	◔	○
ERIM	◔	○	○	●	○	○	◔	◕	◑	◔	○
Donky	◔	○	◕	◕	○	○	◕	◕	◑	◑	○
CHERI	●	○	◑	◕	◔	●	◕	●	●	●	●
SecureCells	◕	◕	●	●	○	◕	◕	◕	◕	●	◑
CAPSTONE	◔	●	◑	●	◑	●	●	●	●	●	●

[1] Performance of memory access control
[2] Cost of exclusive memory access
[3] Cost of memory management
[4] Performance of compartment switching
[5] Performance of permission revocation
[6] Performance of permission transfers
[7] Performance of memory management ops.
[8] Alignment of region for permissions
[9] Granularity of region for permissions
[10] Limit on count of permission regions
[11] Permission transfer granularity

Table C.2: Scoring matrix for mechanisms for performance and flexibility metrics

Bibliography

[1] *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 2015. ISBN: 978-1-4673-6949-7. URL: https://ieeexplore.ieee.org/xpl/conhome/7160813/proceeding.

[2] Carmine Abate, Arthur Azevedo de Amorim, Roberto Blanco, Ana Nora Evans, Guglielmo Fachini, Catalin Hritcu, Théo Laurent, Benjamin C. Pierce, Marco Stronati, and Andrew Tolmach. "When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. Ed. by David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang. ACM, 2018, pp. 1351–1368. ISBN: 978-1-4503-5693-0. DOI: 10.1145/3243734.3243745. URL: https://doi.org/10.1145/3243734.3243745 (cited on page 79).

[3] Eslam G. AbdAllah, Yu Rang Kuang, and Changcheng Huang. "Advanced Encryption Standard New Instructions (AES-NI) Analysis: Security, Performance, and Power Consumption". In: *Proceedings of the 12th International Conference on Computer and Automation Engineering*. 2020 (cited on page 59).

[4] *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*. IEEE Computer Society, 2014. ISBN: 978-1-4799-4396-8. URL: https://ieeexplore.ieee.org/xpl/conhome/6847316/proceeding.

[5] Arm Limited (or its affiliates). *Arm Armv8-A Architecture Registers*. 2022. URL: https://developer.arm.com/documentation/ddi0595/2021-12/AArch64-Instructions/AT-S12E0R--Address-Translate-Stages-1-and-2-EL0-Read (visited on 05/01/2022) (cited on pages 70, 138).

[6] Advanced Micro Devices (AMD). *AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions*. https://www.amd.com/system/files/TechDocs/24594.pdf (cited on page 39).

[7] George Argyros and Aggelos Kiayias. "I Forgot Your Password: Randomness Attacks Against PHP Applications". In: *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*. Ed. by Tadayoshi Kohno. USENIX Association, 2012, pp. 81–

96. URL: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/argyros (cited on page 95).

[8] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. "The Rocket Chip Generator". In: *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17* (2016) (cited on page 69).

[9] Vaggelis Atlidakis, Jeremy Andrus, Roxana Geambasu, Dimitris Mitropoulos, and Jason Nieh. "POSIX abstractions in modern operating systems: the old, the new, and the missing". In: *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016*. Ed. by Cristian Cadar, Peter R. Pietzuch, Kimberly Keeton, and Rodrigo Rodrigues. ACM, 2016, 19:1–19:17. ISBN: 978-1-4503-4240-7. DOI: 10.1145/2901318.2901350. URL: https://doi.org/10.1145/2901318.2901350 (cited on page 5).

[10] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Robert L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. "The NAS parallel benchmarks". In: *The International Journal of Supercomputing Applications* 5.3 (1991), pp. 63–73 (cited on pages 41, 42).

[11] Adam Barth, Collin Jackson, Charles Reis, TGC Team, et al. "The security architecture of the chromium browser". In: *Technical report*. Stanford University, 2008 (cited on pages 2, 63, 86, 87, 99).

[12] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. "Efficient virtual memory for big memory servers". In: *The 40th Annual International Symposium on Computer Architecture, ISCA'13, Tel-Aviv, Israel, June 23-27, 2013*. Ed. by Avi Mendelson. ACM, 2013, pp. 237–248. ISBN: 978-1-4503-2079-5. DOI: 10.1145/2485922.2485943. URL: https://doi.org/10.1145/2485922.2485943 (cited on pages 61, 70, 78).

[13] Adam Belay, Andrea Bittau, Ali José Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. "Dune: Safe User-level Access to Privileged CPU Features". In: *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*. Ed. by Chandu Thekkath and Amin Vahdat. USENIX Association, 2012, pp. 335–348. ISBN: 978-1-931971-96-6. URL: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/belay (cited on page 98).

[14] Abhishek Bhattacharjee. *Preserving the Virtual Memory Abstraction*. https://www.sigarch.org/preserving-the-virtual-memory-abstraction/. 2017 (cited on page 61).

[15] Atri Bhattacharyya, Florian Hofhammer, Yuanlong Li, Siddharth Gupta, Andrés Sánchez, Babak Falsafi, and Mathias Payer. "SecureCells: A Secure Compartmentalized Architecture". In: *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*. IEEE, 2023, pp. 2921–2939. ISBN: 978-1-6654-9336-9. DOI: 10.1109/SP46215.2023.10179472. URL: https://doi.org/10.1109/SP46215.2023.10179472 (cited on pages 11, 84, 87, 98, 106).

[16] Atri Bhattacharyya, Andrés Sánchez, Esmaeil Mohammadian Koruyeh, Nael B. Abu-Ghazaleh, Chengyu Song, and Mathias Payer. "SpecROP: Speculative Exploitation of ROP Chains". In: *23rd International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2020, San Sebastian, Spain, October 14-15, 2020*. Ed. by Manuel Egele and Leyla Bilge. USENIX Association, 2020, pp. 1–16. ISBN: 978-1-939133-18-2. URL: https://www.usenix.org/conference/raid2020/presentation/bhattacharyya (cited on page 11).

[17] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. "SMoTherSpectre: Exploiting Speculative Execution through Port Contention". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*. Ed. by Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz. ACM, 2019, pp. 785–800. ISBN: 978-1-4503-6747-9. DOI: 10.1145/3319535.3363194. URL: https://doi.org/10.1145/3319535.3363194 (cited on page 11).

[18] Atri Bhattacharyya, Uros Tesic, and Mathias Payer. "Midas: Systematic Kernel TOCTTOU Protection". In: *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*. Ed. by Kevin R. B. Butler and Kurt Thomas. USENIX Association, 2022, pp. 107–124. ISBN: 978-1-939133-31-1. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/bhattacharyya (cited on page 10).

[19] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. "Wedge: Splitting Applications into Reduced-Privilege Compartments". In: *5th USENIX Symposium on Networked Systems Design & Implementation, NSDI 2008, April 16-18, 2008, San Francisco, CA, USA, Proceedings*. Ed. by Jon Crowcroft and Michael Dahlin. USENIX Association, 2008, pp. 309–322. ISBN: 978-1-931971-58-4. URL: http://www.usenix.org/events/nsdi08/tech/full_papers/bittau/bittau.pdf (cited on page 98).

[20] Zack Bloom. *Cloud Computing without Containers*. https://blog.cloudflare.com/cloud-computing-without-containers/. 2018 (cited on page 50).

[21] Srdjan Capkun and Franziska Roesner, eds. *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*. USENIX Association, 2020. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20.

[22] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. "Sharing and Protection in a Single-Address-Space Operating System". In: *ACM Trans. Comput.*

Syst. 12.4 (1994), pp. 271–307. DOI: 10.1145/195792.195795. URL: https://doi.org/10.1145/195792.195795 (cited on page 39).

[23] Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. "Shreds: Fine-Grained Execution Units with Private Memory". In: *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*. IEEE Computer Society, 2016, pp. 56–71. ISBN: 978-1-5090-0824-7. DOI: 10.1109/SP.2016.12. URL: https://doi.org/10.1109/SP.2016.12 (cited on pages 60, 98, 114).

[24] Shaanan N. Cohney, Matthew D. Green, and Nadia Heninger. "Practical State Recovery Attacks against Legacy RNG Implementations". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. Ed. by David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang. ACM, 2018, pp. 265–280. ISBN: 978-1-4503-5693-0. DOI: 10.1145/3243734.3243756. URL: https://doi.org/10.1145/3243734.3243756 (cited on page 95).

[25] R. Joseph Connor, Tyler McDaniel, Jared M. Smith, and Max Schuchard. "PKU Pitfalls: Attacks on PKU-based Memory Isolation Systems". In: *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*. Ed. by Srdjan Capkun and Franziska Roesner. USENIX Association, 2020, pp. 1409–1426. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/connor (cited on pages 38, 80).

[26] *CVE-2013-1332*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-1332 (cited on pages 5, 14).

[27] *CVE-2015-8550*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-8550 (cited on pages 5, 14).

[28] *CVE-2016-10433*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-10433 (cited on pages 5, 14).

[29] *CVE-2016-10435*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-10435 (cited on pages 5, 14).

[30] *CVE-2016-10439*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-10439 (cited on pages 5, 14).

[31] *CVE-2016-8438*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-8438 (cited on pages 5, 14).

[32] *CVE-2018-12633*. https://bugzilla.redhat.com/show_bug.cgi?id=CVE-2018-12633 (cited on pages 5, 14, 18).

[33] *CVE-2018-12633 Fix*. https://github.com/torvalds/linux/commit/bd23a7269834dc7c1f93e83
2020 (cited on page 18).

[34] *CVE-2019-20610*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-20610 (cited on pages 5, 14).

[35] *CVE-2019-5519*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-5519 (cited on pages 5, 14).

[36] *CVE-2020-12652*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-12652 (cited on pages 5, 14).

[37] Leonardo Dagum and Ramesh Menon. "OpenMP: an industry standard API for shared-memory programming". In: *IEEE computational science and engineering* 5.1 (1998), pp. 46–55 (cited on page 42).

[38] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram S. Adve. "Nested Kernel: An Operating System Architecture for Intra-Kernel Privilege Separation". In: *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2015, Istanbul, Turkey, March 14-18, 2015*. Ed. by Özcan Özturk, Kemal Ebcioglu, and Sandhya Dwarkadas. ACM, 2015, pp. 191–206. ISBN: 978-1-4503-2835-7. DOI: 10.1145/2694344.2694386. URL: https://doi.org/10.1145/2694344.2694386 (cited on page 98).

[39] Jeffrey Dean and Luiz André Barroso. "The tail at scale". In: *Commun. ACM* 56.2 (2013), pp. 74–80. DOI: 10.1145/2408776.2408794. URL: https://doi.org/10.1145/2408776.2408794 (cited on page 90).

[40] Dong Du, Zhichao Hua, Yubin Xia, Binyu Zang, and Haibo Chen. "XPC: architectural support for secure and efficient cross process call". In: *Proceedings of the 46th International Symposium on Computer Architecture, ISCA 2019, Phoenix, AZ, USA, June 22-26, 2019*. Ed. by Srilatha Bobbie Manne, Hillery C. Hunter, and Erik R. Altman. ACM, 2019, pp. 671–684. ISBN: 978-1-4503-6669-4. DOI: 10.1145/3307650.3322218. URL: https://doi.org/10.1145/3307650.3322218 (cited on pages 3, 51, 58, 60, 78, 84, 98, 102).

[41] Bryan Ford and Jay Lepreau. "Evolving Mach 3.0 to A Migrating Thread Model". In: *USENIX Winter 1994 Technical Conference, San Francisco, California, USA, January 17-21, 1994, Conference Proceedings*. USENIX Association, 1994, pp. 97–114. ISBN: 1-880446-58-8. URL: https://www.usenix.org/conference/usenix-winter-1994-technical-conference/evolving-mach-30-migrating-thread-model (cited on page 60).

[42] Tommaso Frassetto, Patrick Jauernig, Christopher Liebchen, and Ahmad-Reza Sadeghi. "IMIX: In-Process Memory Isolation EXtension". In: *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. Ed. by William Enck and Adrienne Porter Felt. USENIX Association, 2018, pp. 83–97. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/frassetto (cited on pages 51, 60, 78, 98).

[43] Tal Garfinkel. “Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools”. In: *Proceedings of the Network and Distributed System Security Symposium, NDSS 2003, San Diego, California, USA*. The Internet Society, 2003. ISBN: 1-891562-16-9. URL: `https://www.ndss-symposium.org/ndss2003/traps-and-pitfalls-practical-problems-system-call-interposition-based-security-tools/` (cited on page 18).

[44] Adrien Ghosn, Marios Kogias, Mathias Payer, James R. Larus, and Edouard Bugnion. “Enclosure: language-based restriction of untrusted libraries”. In: *ASPLOS ’21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*. Ed. by Tim Sherwood, Emery D. Berger, and Christos Kozyrakis. ACM, 2021, pp. 255–267. ISBN: 978-1-4503-8317-2. DOI: `10.1145/3445814.3446728`. URL: `https://doi.org/10.1145/3445814.3446728` (cited on pages 50, 59, 80, 119).

[45] David B. Golub, Randall W. Dean, Alessandro Forin, and Richard F. Rashid. “UNIX as an Application Program”. In: *Proceedings of the Usenix Summer 1990 Technical Conference, Anaheim, California, USA, June 1990*. USENIX Association, 1990, pp. 87–95 (cited on page 50).

[46] Brendan Gregg. “The flame graph”. In: *Commun. ACM* 59.6 (2016), pp. 48–57. DOI: `10.1145/2909476`. URL: `https://doi.org/10.1145/2909476` (cited on page 45).

[47] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. “Flush+Flush: A Fast and Stealthy Cache Attack”. In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. 2016 (cited on page 94).

[48] Le Guan, Jingqiang Lin, Bo Luo, Jiwu Jing, and Jing Wang. “Protecting Private Keys against Memory Disclosure Attacks Using Hardware Transactional Memory”. In: *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 2015, pp. 3–19. ISBN: 978-1-4673-6949-7. DOI: `10.1109/SP.2015.8`. URL: `https://doi.org/10.1109/SP.2015.8` (cited on page 98).

[49] 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, Denver, CO, USA, October 12-16, 2015*. Ed. by Indrajit Ray, Ninghui Li, and Christopher Kruegel. ACM, 2015, pp. 1016–1031. ISBN: 978-1-4503-3832-5. DOI: `10.1145/2810103.2813611`. URL: `https://doi.org/10.1145/2810103.2813611` (cited on page 79).

[50] Part Guide. “Intel® 64 and ia-32 architectures software developer’s manual”. In: *Volumes 1-4* 2.11 (2011) (cited on page 51).

[51] Siddharth Gupta, Atri Bhattacharyya, Yunho Oh, Abhishek Bhattacharjee, Babak Falsafi, and Mathias Payer. "Rebooting Virtual Memory with Midgard". In: *48th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2021, Valencia, Spain, June 14-18, 2021*. IEEE, 2021, pp. 512–525. ISBN: 978-1-6654-3333-4. DOI: 10.1109/ISCA52012.2021.00047. URL: `https://doi.org/10.1109/ISCA52012.2021.00047` (cited on pages 4, 11, 39, 61, 64, 70, 80).

[52] 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: *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*. Ed. by Dahlia Malkhi and Dan Tsafrir. USENIX Association, 2019, pp. 489–504. URL: `https://www.usenix.org/conference/atc19/presentation/hedayati-hodor` (cited on pages 3, 51, 60, 78, 98).

[53] Dan Hildebrand. "An Architectural Overview of QNX". In: *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures, Seattle, WA, USA, 27-28 April 1992s*. USENIX, 1992, pp. 113–126. ISBN: 1-880446-42-1 (cited on page 50).

[54] Andrew D. Hilton and Amir Roth. "SMT-Directory: Efficient Load-Load Ordering for SMT". In: *IEEE Comput. Archit. Lett.* 9.1 (2010), pp. 25–28. DOI: 10.1109/L-CA.2010.8. URL: `https://doi.org/10.1109/L-CA.2010.8` (cited on page 29).

[55] Terry Ching-Hsiang Hsu, Kevin J. Hoffman, Patrick Eugster, and Mathias Payer. "Enforcing Least Privilege Memory Views for Multithreaded Applications". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. Ed. by Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi. ACM, 2016, pp. 393–405. ISBN: 978-1-4503-4139-4. DOI: 10.1145/2976749.2978327. URL: `https://doi.org/10.1145/2976749.2978327` (cited on pages 3, 59, 78, 80, 98).

[56] Intel Corporation. *Control-flow Enforcement Technology Specification*. `https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf`. Accessed: 2020-03. May 2019 (cited on page 66).

[57] Lana Josipovic, Atri Bhattacharyya, Andrea Guerrieri, and Paolo Ienne. "Shrink It or Shed It! Minimize the Use of LSQs in Dataflow Designs". In: *International Conference on Field-Programmable Technology, FPT 2019, Tianjin, China, December 9-13, 2019*. IEEE, 2019, pp. 197–205. ISBN: 978-1-7281-2943-3. DOI: 10.1109/ICFPT47387.2019.00031. URL: `https://doi.org/10.1109/ICFPT47387.2019.00031` (cited on page 11).

[58] Yannis Juglaret, Catalin Hritcu, Arthur Azevedo de Amorim, Boris Eng, and Benjamin C. Pierce. "Beyond Good and Evil: Formalizing the Security Guarantees of Compartmentalizing Compilation". In: *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*. IEEE Computer Society, 2016, pp. 45–60. ISBN: 978-1-5090-2607-4. DOI: 10.1109/CSF.2016.11. URL: `https://doi.org/10.1109/CSF.2016.11` (cited on page 79).

[59] GC Mateusz Jurczyk and Gynvael Coldwind. "Bochspwn: Identifying 0-days via system-wide memory access pattern analysis". In: *Black Hat USA Briefings (Black Hat USA)* (2013) (cited on pages 3, 18, 47).

[60] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman S. Unsal. "Redundant memory mappings for fast access to large memories". In: *Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, June 13-17, 2015*. ACM, 2015, pp. 66–78. ISBN: 978-1-4503-3402-0. DOI: 10.1145/2749469.2749471. URL: https://doi.org/10.1145/2749469.2749471 (cited on page 61).

[61] Khaled N. Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael B. Abu-Ghazaleh. "SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation". In: *Proceedings of the 56th Annual Design Automation Conference 2019, DAC 2019, Las Vegas, NV, USA, June 02-06, 2019*. ACM, 2019, p. 60. ISBN: 978-1-4503-6725-7. DOI: 10.1145/3316781.3317903. URL: https://doi.org/10.1145/3316781.3317903 (cited on page 47).

[62] Douglas Kilpatrick. "Privman: A Library for Partitioning Applications". In: *Proceedings of the FREENIX Track: 2003 USENIX Annual Technical Conference, June 9-14, 2003, San Antonio, Texas, USA*. USENIX, 2003, pp. 273–284. ISBN: 1-931971-11-0. URL: http://www.usenix.org/events/usenix03/tech/freenix03/kilpatrick.html (cited on page 60).

[63] 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: *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022*. Ed. by Yérom-David Bromberg, Anne-Marie Kermarrec, and Christos Kozyrakis. ACM, 2022, pp. 132–148. ISBN: 978-1-4503-9162-7. DOI: 10.1145/3492321.3519582. URL: https://doi.org/10.1145/3492321.3519582 (cited on pages 60, 79).

[64] 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 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*. Ed. by Jeanna Neefe Matthews and Thomas E. Anderson. ACM, 2009, pp. 207–220. ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629596. URL: https://doi.org/10.1145/1629575.1629596 (cited on pages 51, 60).

[65] 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: *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE,

2019, pp. 1–19. ISBN: 978-1-5386-6660-9. DOI: 10.1109/SP.2019.00002. URL: `https://doi.org/10.1109/SP.2019.00002` (cited on pages 19, 47, 66, 81).

[66] 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". In: *Proc. ACM Program. Lang.* 6.POPL (2022), pp. 1–30. DOI: 10.1145/3498688. URL: `https://doi.org/10.1145/3498688` (cited on page 60).

[67] 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 Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*. Ed. by Gustavo Alonso, Ricardo Bianchini, and Marko Vukolic. ACM, 2017, pp. 437–452. ISBN: 978-1-4503-4938-3. DOI: 10.1145/3064176.3064217. URL: `https://doi.org/10.1145/3064176.3064217` (cited on pages 51, 60, 78, 98).

[68] Gwangmu Lee, Woochul Shim, and Byoungyoung Lee. "Constraint-guided Directed Greybox Fuzzing". In: *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*. Ed. by Michael D. Bailey and Rachel Greenstadt. USENIX Association, 2021, pp. 3559–3576. ISBN: 978-1-939133-24-3. URL: `https://www.usenix.org/conference/usenixsecurity21/presentation/lee-gwangmu` (cited on page 84).

[69] 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 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. Ed. by David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang. ACM, 2018, pp. 1441–1454. ISBN: 978-1-4503-5693-0. DOI: 10.1145/3243734.3243748. URL: `https://doi.org/10.1145/3243734.3243748` (cited on pages 3, 51, 60, 78).

[70] Roy Levin, Ellis S. Cohen, William M. Corwin, Fred J. Pollack, and William A. Wulf. "Policy/Mechanism Separation in HYDRA". In: *Proceedings of the Fifth Symposium on Operating System Principles, SOSP 1975, The University of Texas at Austin, Austin, Texas, USA, November 19-21, 1975*. Ed. by James C. Browne and Juan Rodriguez-Rosell. ACM, 1975, pp. 132–140. DOI: 10.1145/800213.806531. URL: `https://doi.org/10.1145/800213.806531` (cited on page 50).

[71] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. "Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency". In: *Proceedings of the ACM Symposium on Cloud Computing, Seattle, WA, USA, November 3-5, 2014*. Ed. by Ed Lazowska, Doug Terry, Remzi H. Arpaci-Dusseau, and Johannes Gehrke. ACM, 2014, 9:1–9:14. ISBN: 978-1-4503-3252-1. DOI: 10.1145/2670979.2670988. URL: `https://doi.org/10.1145/2670979.2670988` (cited on page 90).

[72] Qing Li, David Hartley, and Brian Rosenberg. *An Introduction to Access Control on Qualcomm Snapdragon Platforms*. `https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/whitepaper_0.pdf`. 2020 (cited on page 134).

[73] David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, eds. *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. ACM, 2018. ISBN: 978-1-4503-5693-0. URL: `http://dl.acm.org/citation.cfm?id=3243734`.

[74] 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: *USENIX Security Symposium*. 2018 (cited on pages 81, 95).

[75] Richard J. Lipton and Lawrence Snyder. "A Linear Time Algorithm for Deciding Subject Security". In: *J. ACM* 24.3 (1977), pp. 455–464. DOI: `10.1145/322017.322025`. URL: `https://doi.org/10.1145/322017.322025` (cited on pages 58, 79).

[76] 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: *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. Ed. by Kimberly Keeton and Timothy Roscoe. USENIX Association, 2016, pp. 49–64. URL: `https://www.usenix.org/conference/osdi16/technical-sessions/presentation/litton` (cited on pages 3, 51, 60, 61, 78, 80, 84, 89, 98, 100, 114).

[77] 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, Denver, CO, USA, October 12-16, 2015*. Ed. by Indrajit Ray, Ninghui Li, and Christopher Kruegel. ACM, 2015, pp. 1607–1619. ISBN: 978-1-4503-3832-5. DOI: `10.1145/2810103.2813690`. URL: `https://doi.org/10.1145/2810103.2813690` (cited on page 98).

[78] ARM Ltd. *ARM Architecture Reference Manual (ARMv8, for ARMv8-Aarchitecture profile)*. 2013 (cited on page 39).

[79] Kai Lu, Peng-Fei Wang, Gen Li, and Xu Zhou. "Untrusted hardware causes double-fetch problems in the I/O memory". In: *Journal of Computer Science and Technology* 33.3 (2018), pp. 587–602 (cited on page 18).

[80] Yingqi Luo, Pengfei Wang, Xu Zhou, and Kai Lu. "DFTinker: Detecting and Fixing Double-Fetch Bugs in an Automated Way". In: *Wireless Algorithms, Systems, and Applications - 13th International Conference, WASA 2018, Tianjin, China, June 20-22, 2018, Proceedings*. Ed. by Sriram Chellappan, Wei Cheng, and Wei Li. Vol. 10874. Lecture Notes in Computer Science. Springer, 2018, pp. 780–785. ISBN: 978-3-319-94267-4. DOI: `10.1007/978-3-319-94268-1\_67`. URL: `https://doi.org/10.1007/978-3-319-94268-1\_67` (cited on pages 3, 46, 47).

[81] Dahlia Malkhi and Dan Tsafrir, eds. *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*. USENIX Association, 2019. URL: `https://www.usenix.org/conference/atc19`.

[82] Srilatha Bobbie Manne, Hillery C. Hunter, and Erik R. Altman, eds. *Proceedings of the 46th International Symposium on Computer Architecture, ISCA 2019, Phoenix, AZ, USA, June 22-26, 2019.* ACM, 2019. ISBN: 978-1-4503-6669-4. URL: `https://dl.acm.org/citation.cfm?id=3307650`.

[83] João Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Andrei 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 2014, Seattle, WA, USA, April 2-4, 2014.* Ed. by Ratul Mahajan and Ion Stoica. USENIX Association, 2014, pp. 459–473. ISBN: 978-1-931971-09-6. URL: `https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/martins` (cited on page 55).

[84] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil D. Gligor, and Adrian Perrig. "TrustVisor: Efficient TCB Reduction and Attestation". In: *31st IEEE Symposium on Security and Privacy, SP 2010, 16-19 May 2010, Berleley/Oakland, California, USA.* IEEE Computer Society, 2010, pp. 143–158. ISBN: 978-0-7695-4035-1. DOI: `10.1109/SP.2010.17`. URL: `https://doi.org/10.1109/SP.2010.17` (cited on page 98).

[85] Derrick McKee, Yianni Giannaris, Carolina Ortega Perez, Howard Shrobe, Mathias Payer, Hamed Okhravi, and Nathan Burow. "Preventing Kernel Hacks with HAKC". In: *Proceedings 2022 Network and Distributed System Security Symposium. NDSS.* Vol. 22. 2022, pp. 1–17 (cited on page 60).

[86] Microsoft. *ChakraCore.* URL: `https://github.com/chakra-core/ChakraCore` (cited on pages 2, 99).

[87] Loïc Miller, Pascal Mérindol, Antoine Gallais, and Cristel Pelsser. "Towards Secure and Leak-Free Workflows Using Microservice Isolation". In: *2021 IEEE 22nd International Conference on High Performance Switching and Routing (HPSR).* 2021, pp. 1–5. DOI: `10.1109/HPSR52026.2021.9481820` (cited on page 50).

[88] Jan de Mooij. *W^X JIT-code enabled in Firefox.* `https://jandemooij.nl/blog/wx-jit-code-enabled-in-firefox/`. 2015 (cited on pages 2, 87).

[89] Onur Mutlu and Jeremie S Kim. "RowHammer: A retrospective". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2019) (cited on pages 19, 111).

[90] National Vulnerability Database (NVD). *CVE-2014-0160.* `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160`. 2014 (cited on page 2).

[91] National Vulnerability Database (NVD). *CVE-2015-0235.* `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-0235`. 2014 (cited on page 2).

[92] National Vulnerability Database (NVD). *CVE-2015-3036.* `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3036`. 2014 (cited on page 2).

[93] National Vulnerability Database (NVD). *CVE-2021-44228*. `https://www.cve.org/CVERecord?id=CVE-2021-44228`. 2021 (cited on page 50).

[94] Lena E. Olson, Jason Power, Mark D. Hill, and David A. Wood. "Border control: sandboxing accelerators". In: *Proceedings of the 48th International Symposium on Microarchitecture, MICRO 2015, Waikiki, HI, USA, December 5-9, 2015*. Ed. by Milos Prvulovic. ACM, 2015, pp. 470–481. ISBN: 978-1-4503-4034-2. DOI: 10.1145/2830772.2830819. URL: `https://doi.org/10.1145/2830772.2830819` (cited on page 21).

[95] *OSBench*. `https://github.com/mbitsnbites/osbench/` (cited on page 41).

[96] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. "libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK)". In: *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*. Ed. by Dahlia Malkhi and Dan Tsafrir. USENIX Association, 2019, pp. 241–254. URL: `https://www.usenix.org/conference/atc19/presentation/park-soyeon` (cited on pages 3, 51, 57, 59, 84, 87, 98, 103).

[97] Mathias Payer and Thomas R Gross. "Protecting applications against TOCTTOU races by user-space caching of file metadata". In: *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*. 2012, pp. 215–226 (cited on pages 18, 19).

[98] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. "CoLT: Coalesced Large-Reach TLBs". In: *45th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2012, Vancouver, BC, Canada, December 1-5, 2012*. IEEE Computer Society, 2012, pp. 258–269. ISBN: 978-1-4673-4819-5. DOI: 10.1109/MICRO.2012.32. URL: `https://doi.org/10.1109/MICRO.2012.32` (cited on pages 61, 64, 78).

[99] *Phoronix Test Suite*. `https://www.phoronix-test-suite.com/`. Aug. 2020 (cited on pages 41, 43).

[100] *Project Fission*. Accessed: Nov 2021. 2021. URL: `https://wiki.mozilla.org/Project_Fission` (cited on pages 50, 87).

[101] Niels Provos, Panayiotis Mavrommatis, Moheeb Abu Rajab, and Fabian Monrose. "All Your iFRAMEs Point to Us". In: *Proceedings of the 17th USENIX Security Symposium, July 28-August 1, 2008, San Jose, CA, USA*. Ed. by Paul C. van Oorschot. USENIX Association, 2008, pp. 1–16. ISBN: 978-1-931971-60-7. URL: `http://www.usenix.org/events/sec08/tech/full\_papers/provos/provos.pdf` (cited on page 86).

[102] Niels Provos, Dean McNamee, Panayiotis Mavrommatis, Ke Wang, and Nagendra Modadugu. "The Ghost in the Browser: Analysis of Web-based Malware". In: *First Workshop on Hot Topics in Understanding Botnets, HotBots'07, Cambridge, MA, USA, April 10, 2007*. Ed. by Niels Provos. USENIX Association, 2007. URL: `https://www.usenix.org/conference/hotbots-07/ghost-browser-analysis-web-based-malware` (cited on page 86).

[103] Calton Pu and Jinpeng Wei. "A Methodical Defense against TOCTTOU Attacks: The EDGI Approach". In: *Proceedings of the 2006 International Symposium on Secure Software Engineering*. 2006 (cited on pages 18, 19).

[104] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. "CrossTalk: Speculative Data Leaks Across Cores Are Real". In: *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 2021, pp. 1852–1867. ISBN: 978-1-7281-8935-2. DOI: `10.1109/SP40001.2021.00020`. URL: `https://doi.org/10.1109/SP40001.2021.00020` (cited on page 95).

[105] Joseph Ravichandran, Weon Taek Na, Jay Lang, and Mengjia Yan. "PACMAN: attacking ARM pointer authentication with speculative execution". In: *ISCA '22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022*. Ed. by Valentina Salapura, Mohamed Zahran, Fred Chong, and Lingjia Tang. ACM, 2022, pp. 685–698. ISBN: 978-1-4503-8610-4. DOI: `10.1145/3470496.3527429`. URL: `https://doi.org/10.1145/3470496.3527429` (cited on page 134).

[106] Indrajit Ray, Ninghui Li, and Christopher Kruegel, eds. *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*. ACM, 2015. ISBN: 978-1-4503-3832-5. URL: `http://dl.acm.org/citation.cfm?id=2810103`.

[107] Nick Roessler, Lucas Atayde, Imani Palmer, Derrick Paul McKee, Jai Pandey, Vasileios P. Kemerlis, Mathias Payer, Adam Bates, Jonathan M. Smith, André DeHon, and Nathan Dautenhahn. "μSCOPE: A Methodology for Analyzing Least-Privilege Compartmentalization in Large Software Artifacts". In: *RAID '21: 24th International Symposium on Research in Attacks, Intrusions and Defenses, San Sebastian, Spain, October 6-8, 2021*. Ed. by Leyla Bilge and Tudor Dumitras. ACM, 2021, pp. 296–311. ISBN: 978-1-4503-9058-3. DOI: `10.1145/3471621.3471839`. URL: `https://doi.org/10.1145/3471621.3471839` (cited on pages 60, 79).

[108] Bogdan F. Romanescu, Alvin R. Lebeck, Daniel J. Sorin, and Anne Bracy. "UNified Instruction/Translation/Data (UNITD) coherence: One protocol to rule them all". In: *16th International Conference on High-Performance Computer Architecture (HPCA-16 2010), 9-14 January 2010, Bangalore, India*. Ed. by Matthew T. Jacob, Chita R. Das, and Pradip Bose. IEEE Computer Society, 2010, pp. 1–12. ISBN: 978-1-4244-5659-8. DOI: `10.1109/HPCA.2010.5416643`. URL: `https://doi.org/10.1109/HPCA.2010.5416643` (cited on pages 16, 39).

[109] Marc Rozier, Vadim Abrossimov, François Armand, Ivan Boule, Michel Gien, Marc Guillemont, Frédéric Herrmann, Claude Kaiser, Sylvain Langlois, Pierre Léonard, and Will Neuhauser. "CHORUS Distributed Operating System". In: *Comput. Syst.* 1.4 (1988), pp. 305–370 (cited on page 50).

[110] Jerome H. Saltzer. "Protection and the Control of Information Sharing in Multics". In: *Commun. ACM* 17.7 (1974), pp. 388–402 (cited on page 84).

[111] Jerome H. Saltzer and Michael D. Schroeder. “The protection of information in computer systems”. In: *Proceedings of the IEEE* 63.9 (1975), pp. 1278–1308. DOI: 10.1109/PROC.1975.9939. URL: https://doi.org/10.1109/PROC.1975.9939 (cited on pages 50, 53).

[112] David Schrammel, Samuel Weiser, Richard Sadek, and Stefan Mangard. “Jenny: Securing Syscalls for PKU-based Memory Isolation Systems”. In: *USENIX Security Symposium*. 2022 (cited on page 80).

[113] 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: *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*. Ed. by Srdjan Capkun and Franziska Roesner. USENIX Association, 2020, pp. 1677–1694. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/schrammel (cited on pages 51, 60, 79, 84, 87, 98, 104).

[114] Michael Schwarz, Daniel Gruss, Moritz Lipp, Clémentine Maurice, Thomas Schuster, Anders Fogh, and Stefan Mangard. “Automated Detection, Exploitation, and Elimination of Double-Fetch Bugs using Modern CPU Features”. In: *Proceedings of the 2018 on Asia Conference on Computer and Communications Security, AsiaCCS 2018, Incheon, Republic of Korea, June 04-08, 2018*. Ed. by Jong Kim, Gail-Joon Ahn, Seungjoo Kim, Yongdae Kim, Javier López, and Taesoo Kim. ACM, 2018, pp. 587–600. DOI: 10.1145/3196494.3196508. URL: https://doi.org/10.1145/3196494.3196508 (cited on pages 3, 47).

[115] *SecComp*. https://www.kernel.org/doc/html/latest/userspace-api/seccomp_filter.html (cited on pages 6, 14).

[116] *Seccomp and deep argument inspection*. https://lwn.net/Articles/822256/ (cited on page 14).

[117] Fermin J. Serna. *MS08-061 : The case of the kernel mode double-fetch*. https://msrc-blog.microsoft.com/2008/10/14/ms08-061-the-case-of-the-kernel-mode-double-fetch/. 2008 (cited on page 14).

[118] Hovav Shacham. “The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)”. In: *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007*. Ed. by Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson. ACM, 2007, pp. 552–561. ISBN: 978-1-59593-703-2. DOI: 10.1145/1315245.1315313. URL: https://doi.org/10.1145/1315245.1315313 (cited on page 72).

[119] Monirul Islam Sharif, Wenke Lee, Weidong Cui, and Andrea Lanzi. “Secure in-VM monitoring using hardware virtualization”. In: *Proceedings of the 2009 ACM Conference on Computer and Communications Security, CCS 2009, Chicago, Illinois, USA, November 9-13, 2009*. Ed. by Ehab Al-Shaer, Somesh Jha, and Angelos D. Keromytis. ACM, 2009, pp. 477–

487. ISBN: 978-1-60558-894-0. DOI: 10.1145/1653662.1653720. URL: https://doi.org/10.1145/1653662.1653720 (cited on page 98).

[120] Simon Shillaker and Peter R. Pietzuch. "Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing". In: *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*. Ed. by Ada Gavrilovska and Erez Zadok. USENIX Association, 2020, pp. 419–433. ISBN: 978-1-939133-14-4. URL: https://www.usenix.org/conference/atc20/presentation/shillaker (cited on page 50).

[121] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. "Reasoning about a Machine with Local Capabilities: Provably Safe Stack and Return Pointer Management". In: *ACM Trans. Program. Lang. Syst.* 42.1 (2020), 5:1–5:53. DOI: 10.1145/3363519. URL: https://doi.org/10.1145/3363519 (cited on page 61).

[122] Marc Snir, William Gropp, Steve Otto, Steven Huss-Lederman, Jack Dongarra, and David Walker. *MPI–the Complete Reference: the MPI core*. Vol. 1. MIT press, 1998 (cited on page 42).

[123] riscv-software src. *RISC-V open source supervisor binary interface*. 2022. URL: https://github.com/riscv-software-src/opensbi (visited on 05/01/2022) (cited on page 69).

[124] Chrome Security Team. *What's up with in-the-wild exploits? Plus, what we're doing about it.* URL: https://security.googleblog.com/2022/03/whats-up-with-in-wild-exploits-plus.html (cited on page 6).

[125] Google Open Source Security Team. *Linux Kernel Security Done Right*. URL: https://security.googleblog.com/2021/08/linux-kernel-security-done-right.html (cited on page 6).

[126] Google Vulnerability Rewards Team. *Vulnerability Reward Program: 2022 Year in Review*. URL: https://security.googleblog.com/2023/02/vulnerability-reward-program-2022-year.html (cited on page 6).

[127] Dan Tsafrir, Tomer Hertz, David A Wagner, and Dilma Da Silva. "Portably Solving File TOCTTOU Races with Hardness Amplification." In: *FAST*. Vol. 8. 2008, pp. 1–18 (cited on pages 18, 19).

[128] twiz and sgrakkyu. "From RING 0 to UID 0". In: *CCC* (2007) (cited on pages 14, 46).

[129] 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: *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*. Ed. by Nadia Heninger and Patrick Traynor. USENIX Association, 2019, pp. 1221–1238. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/vahldiek-oberwagner (cited on pages 51, 60, 78, 79, 84, 87, 98, 104).

[130] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, and Jonathan M. Smith. "Towards Fine-grained, Automated Application Compartmentalization". In: *Proceedings of the 9th Workshop on Programming Languages and Operating Systems, Shanghai, China, October 28, 2017*. Ed. by Julia Lawall. ACM, 2017, pp. 43–50. ISBN: 978-1-4503-5153-9. DOI: `10.1145/3144555.3144563`. URL: `https://doi.org/10.1145/3144555.3144563` (cited on pages 60, 79).

[131] Lluís Vilanova, Muli Ben-Yehuda, Nacho Navarro, Yoav Etsion, and Mateo Valero. "CODOMs: Protecting software with Code-centric memory Domains". In: *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*. IEEE Computer Society, 2014, pp. 469–480. ISBN: 978-1-4799-4396-8. DOI: `10.1109/ISCA.2014.6853202`. URL: `https://doi.org/10.1109/ISCA.2014.6853202` (cited on pages 58, 60, 78, 84, 98, 101).

[132] Carlos Villavieja, Vasileios Karakostas, Lluís Vilanova, Yoav Etsion, Alex Ramírez, Avi Mendelson, Nacho Navarro, Adrián Cristal, and Osman S. Unsal. "DiDi: Mitigating the Performance Impact of TLB Shootdowns Using a Shared TLB Directory". In: *2011 International Conference on Parallel Architectures and Compilation Techniques, PACT 2011, Galveston, TX, USA, October 10-14, 2011*. Ed. by Lawrence Rauchwerger and Vivek Sarkar. IEEE Computer Society, 2011, pp. 340–349. ISBN: 978-1-4577-1794-9. DOI: `10.1109/PACT.2011.65`. URL: `https://doi.org/10.1109/PACT.2011.65` (cited on pages 16, 39).

[133] Dmitry Vyukov and Andrey Konovalov. *Syzkaller: an unsupervised, coverage-guided kernel fuzzer*. 2019 (cited on page 117).

[134] David A Wagner. "Janus: an approach for confinement of untrusted applications". MA thesis. University of California, Berkeley, 1999 (cited on page 14).

[135] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. "Efficient Software-Based Fault Isolation". In: *Proceedings of the Fourteenth ACM Symposium on Operating System Principles, SOSP 1993, The Grove Park Inn and Country Club, Asheville, North Carolina, USA, December 5-8, 1993*. Ed. by Andrew P. Black and Barbara Liskov. ACM, 1993, pp. 203–216. ISBN: 0-89791-632-8. DOI: `10.1145/168619.168635`. URL: `https://doi.org/10.1145/168619.168635` (cited on page 98).

[136] Pengfei Wang, Jens Krinke, Kai Lu, Gen Li, and Steve Dodier-Lazaro. "How double-fetch situations turn into double-fetch vulnerabilities: A study of double fetches in the linux kernel". In: *26th {USENIX} Security Symposium ({USENIX} Security 17)*. 2017, pp. 1–16 (cited on pages 3, 46).

[137] Pengfei Wang, Kai Lu, Gen Li, and Xu Zhou. "A survey of the double-fetch vulnerabilities". In: *Concurrency and Computation: Practice and Experience* 30.6 (2018), e4345 (cited on pages 14, 18).

[138] Pengfei Wang, Kai Lu, Gen Li, and Xu Zhou. "DFTracker: detecting double-fetch bugs by multi-taint parallel tracking". In: *Frontiers of Computer Science* 13.2 (2019), pp. 247–263 (cited on pages 3, 46).

[139] Robert N. M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav H. Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert M. Norton, Michael Roe, Stacey D. Son, and Munraj Vadera. "CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization". In: *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 2015, pp. 20–37. ISBN: 978-1-4673-6949-7. DOI: 10.1109/SP.2015.9. URL: https://doi.org/10.1109/SP.2015.9 (cited on pages 58, 59, 76, 79, 84, 87, 98, 105).

[140] Robert NM Watson. "Exploiting Concurrency Vulnerabilities in System Call Wrappers." In: *WOOT* 7 (2007), pp. 1–8 (cited on pages 14, 19, 46).

[141] Jinpeng Wei and Calton Pu. "Modeling and preventing TOCTTOU vulnerabilities in Unix-style file systems". In: *computers & security* 29.8 (2010), pp. 815–830 (cited on pages 18, 19).

[142] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas F. Wenisch, and Baris Kasikci. "NDA: Preventing Speculative Execution Attacks at Their Source". In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*. ACM, 2019, pp. 572–586. DOI: 10.1145/3352460.3358306. URL: https://doi.org/10.1145/3352460.3358306 (cited on page 81).

[143] Felix Wilhelm. "Xenpwn: Breaking paravirtualized devices". In: *Black Hat USA* (2016) (cited on pages 3, 14, 18, 47).

[144] Emmett Witchel, Josh Cates, and Krste Asanovic. "Mondrian memory protection". In: *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X), San Jose, California, USA, October 5-9, 2002*. Ed. by Kourosh Gharachorloo and David A. Wood. ACM Press, 2002, pp. 304–316. ISBN: 1-58113-574-2. DOI: 10.1145/605397.605429. URL: https://doi.org/10.1145/605397.605429 (cited on pages 51, 60, 79, 98, 100).

[145] Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert M. Norton, and Michael Roe. "The CHERI capability model: Revisiting RISC in an age of risk". In: *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*. IEEE Computer Society, 2014, pp. 457–468. ISBN: 978-1-4799-4396-8. DOI: 10.1109/ISCA.2014.6853201. URL: https://doi.org/10.1109/ISCA.2014.6853201 (cited on pages 79, 105).

[146] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. "An Empirical Evaluation of In-Memory Multi-Version Concurrency Control". In: *Proc. VLDB Endow.* 10.7 (2017), pp. 781–792. DOI: 10.14778/3067421.3067427. URL: http://www.vldb.org/pvldb/vol10/p781-Wu.pdf (cited on page 22).

[147] Meng Xu, Chenxiong Qian, Kangjie Lu, Michael Backes, and Taesoo Kim. "Precise and Scalable Detection of Double-Fetch Bugs in OS Kernels". In: *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. IEEE Computer Society, 2018, pp. 661–678. ISBN: 978-1-5386-4353-2. DOI: 10.1109/SP.2018.00017. URL: https://doi.org/10.1109/SP.2018.00017 (cited on pages 3, 46, 47).

[148] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. "InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy (Corrigendum)". In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*. ACM, 2019, p. 1076. DOI: 10.1145/3352460.3361129. URL: https://doi.org/10.1145/3352460.3361129 (cited on page 47).

[149] Zi Yan, Daniel Lustig, David W. Nellans, and Abhishek Bhattacharjee. "Translation ranger: operating system support for contiguity-aware TLBs". In: *Proceedings of the 46th International Symposium on Computer Architecture, ISCA 2019, Phoenix, AZ, USA, June 22-26, 2019*. Ed. by Srilatha Bobbie Manne, Hillery C. Hunter, and Erik R. Altman. ACM, 2019, pp. 698–710. ISBN: 978-1-4503-6669-4. DOI: 10.1145/3307650.3322223. URL: https://doi.org/10.1145/3307650.3322223 (cited on pages 61, 64, 70, 78).

[150] Zi Yan, Ján Veselý, Guilherme Cox, and Abhishek Bhattacharjee. "Hardware Translation Coherence for Virtualized Systems". In: *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017, Toronto, ON, Canada, June 24-28, 2017*. ACM, 2017, pp. 430–443. ISBN: 978-1-4503-4892-8. DOI: 10.1145/3079856.3080211. URL: https://doi.org/10.1145/3079856.3080211 (cited on pages 16, 39).

[151] Junfeng Yang, Ang Cui, Salvatore J. Stolfo, and Simha Sethumadhavan. "Concurrency Attacks". In: *4th USENIX Workshop on Hot Topics in Parallelism, HotPar'12, Berkeley, CA, USA, June 7-8, 2012*. Ed. by Hans-Juergen Boehm and Luis Ceze. USENIX Association, 2012. URL: https://www.usenix.org/conference/hotpar12/workshop-program/presentation/yang (cited on page 46).

[152] Yuval Yarom and Katrina Falkner. "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack". In: *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*. Ed. by Kevin Fu and Jaeyeon Jung. USENIX Association, 2014, pp. 719–732. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom (cited on page 94).

[153] Jason Zhijingcheng Yu, Conrad Watt, Aditya Badole, Trevor E. Carlson, and Prateek Saxena. “Capstone: A Capability-based Foundation for Trustless Secure Memory Access”. In: *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*. Ed. by Joseph A. Calandrino and Carmela Troncoso. USENIX Association, 2023, pp. 787–804. URL: `https://www.usenix.org/conference/usenixsecurity23/presentation/yu-jason` (cited on pages 84, 87, 98, 118).

[154] Lixin Zhang, Evan Speight, Ramakrishnan Rajamony, and Jiang Lin. “Enigma: architectural and operating system support for reducing the impact of address translation”. In: *Proceedings of the 24th International Conference on Supercomputing, 2010, Tsukuba, Ibaraki, Japan, June 2-4, 2010*. Ed. by Taisuke Boku, Hiroshi Nakashima, and Avi Mendelson. ACM, 2010, pp. 159–168. ISBN: 978-1-4503-0018-6. DOI: `10.1145/1810085.1810109`. URL: `https://doi.org/10.1145/1810085.1810109` (cited on pages 64, 80).

[155] Yajin Zhou, Xiaoguang Wang, Yue Chen, and Zhi Wang. “ARMlock: Hardware-based Fault Isolation for ARM”. In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*. Ed. by Gail-Joon Ahn, Moti Yung, and Ninghui Li. ACM, 2014, pp. 558–569. ISBN: 978-1-4503-2957-6. DOI: `10.1145/2660267.2660344`. URL: `https://doi.org/10.1145/2660267.2660344` (cited on page 98).

[156] Xiaogang Zhu and Marcel Böhme. “Regression Greybox Fuzzing”. In: *CCS ’21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*. Ed. by Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi. ACM, 2021, pp. 2169–2182. ISBN: 978-1-4503-8454-4. DOI: `10.1145/3460120.3484596`. URL: `https://doi.org/10.1145/3460120.3484596` (cited on page 84).

Atri Bhattacharyya

Homepage: atrib.bitbucket.io
LinkedIn: /atri-bhattacharyya
atri.bhattacharyya@epfl.ch

Lausanne, Vaud
Switzerland

INTERESTS

Security at the HW/SW interface: security-focused ISA extensions, OS security, microarchitectural security. Datacenter architectures: tackling challenges in protection of the virtual memory abstraction

EDUCATION

PhD candidate, Computer Science ’18 - ’24
Ecole Polytechnique Federale de Lausanne, Switzerland

MS in Computer Science ’16 - ’18
Ecole Polytechnique Federale de Lausanne, Switzerland
GPA: 5.73/6 (equivalent to 3.73/4)

BT in Electrical Eng. with major in Comp. Sc. and Eng. ’11 - ’16
Indian Institute of Technology Kanpur, India
GPA: 9.1/10 (equivalent to 3.64/4)

INDUSTRY EXPERIENCE

Engineering Intern Jun ’22 - Aug ’22
Qualcomm, San Diego, US
Worked on analyzing configurations and investigating improvements for physical-address based **on-chip access control** for Qualcomm products.

Research Assistant August ’17 - Feb ’18
Oracle Labs, Zurich, Switzerland
Developed a **DPDK**-based multi-user capable **userspace-networking framework in C** capable of saturating 10Gbit/s interfaces with a single core while providing the benefits of in-kernel networking: isolation, flexibility and security.

- Integrated the framework with a network-processing bound DDoS detection pipeline to increase its maximum throughput by around 15x.

UnnaTI Embedded Software Intern May ’14 - July ’14
Texas Instruments, Bangalore, India
Developed a **profiler for cycle-wise timing** of C/assembly level instruction execution on an embedded platform to enable rapid profiling and benchmarking to:

- Augment and verify manual benchmarking results in less than 20% of the time.
- Benchmark, optimize TI’s low-power MCU software and FreeRTOS on Cortex-M0+ processor. Achieved >50% improvement for certain OS benchmarks.

RESEARCH

Secure and high-performance virtual memory compartmentalization
SecureCells: A Secure Compartmentalized Architecture, *IEEE S&P’23.*
Rebooting Virtual Memory with Midgard *ISCA’21*
SecureCells proposes a **novel virtual memory architecture** for high-performance hardware-enforced intra-address space isolation of compartments. SecureCells builds on Midgard and RISC-V, using virtual memory areas (VMAs) as the granularity of protection and translation. For SecureCells, we **ported the full RISC-V software stack** including an OS (seL4), the GNU compiler toolchain and benchmarks. We also created two implementations, **modifying QEMU emulator** and the **RISC-V RocketChip FPGA**.
Skills: C, RISC-V, QEMU, gcc, seL4, Linux

Systematic data race protection for the kernel
Midas: Systematic Kernel TOCTTOU Protection *Usenix SEC'22*
Midas is a **systematic and comprehensive protection mechanisms** for OS kernels to defend against Time-of-Check-to-Time-of-Use attacks. I **modified kernel APIs**, creating a multiversioning system for userspace data, assuring the kernel that user data accessed from a system call remains immutable and userspace is not blocked.
Skills: Linux kernel development, C

Speculative side-channel exploitation
SpecROP: Speculative Exploitation of ROP Chains *RAID'20*
SMoTherSpectre: Exploiting speculative execution through port contention CCS'19
SMoTherSpectre presents a **speculative-execution attack** using port contention as a side-channel, enabling leakage of private key from OpenSSH server. SpecROP leverages **binary analysis** and improves attacks by speculatively chaining code gadgets leveraging the CPU's prediction structures, enabling previously impossible leakage scenarios.
Skills: Side-channel exploitation, binary analysis, CPU microarchitecture

Optimizing LSQ generation for High-Level Synthesis
Shrink It and Shred It! Minimize the Use of LSQs in Dataflow Designs *FPT'19*
Developed an optimized load-store queue design for dynamically-scheduled elastic circuits for high-level synthesis, using an **LLVM analysis pass** to determine temporal-ordering between memory operations to reduce the estimated hardware cost for LSQs by as much as 93%.
Skills: LLVM, FPGA programming

SKILLS

Programming: C/C++, Java, LaTeX Python (>5000 LoC), Assembly(x86, ARMv6, RISC-V), VHDL, Shell(Bash) (>1000 LoC), Scala, Matlab.
Simulators: SimFlex, gem5
Software: QEMU, Linux kernel, LLVM, GNU compiler toolchain, seL4

ACHIEVEMENTS AND AWARDS

'23		Qualcomm Innovation Fellowship, Europe
'21, '22		Best TA Award, EPFL
'20		IBM Research Fellowship
'18		EPFL IC School Fellowship
'16		MS Research Scholarship, EPFL
'12	University level	Academic Excellence Award

TEACHING
S=Spring
F=Fall

Undergraduate	CS212	Systems Programming Project (S'19)
Undergraduate	CS323	Introduction to Operating Systems (F'19, F'20, F'21)
Graduate	CS412	Software Security (S'20, S'21)
Graduate	CS422	Database Systems (S'18)
Graduate	COM402	Internet Security and Privacy (F'22, F'23)

TALKS

IEEE S&P '23	SecureCells: A Secure Compartmentalized Architecture
Usenix Security '22	Midas: Systematic Kernel TOCTTOU Protection
RAID '20	SpecROP: Speculative Execution of ROP chains
CCS '19	SMoTherSpectre: Exploiting spec. exec. through port contention

SERVICE

Reviewer	ACM Computing Surveys (CSUR)
Reviewer	ACM Transactions on Computers