Code-Centric Domain Isolation: A Hardware/Software Co-Design for Efficient Program Isolation

---

Lluís Vilanova

<vilanova@ac.upc.edu>

Barcelona, 2015

ADVISORS:

Nacho Navarro
Universitat Politècnica de Catalunya
Barcelona Supercomputing Center

Yoav Etsion
Technion — Israel Institute of Technology
Technion Computer Engineering Center

A thesis submitted in fulfillment of
the requirements for the degree of

DOCTOR OF PHILOSOPHY / DOCTOR PER LA UPC
International Doctorate Mention

Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya

Thanks to all the lights along the journey,
without whom it would be too dark to travel.

Abstract

Current Operating Systems (OSs) employ a process-centric isolation model. This is partly attributed to existing processors providing memory isolation across page tables. In this prevailing model, threads are bound to their creating process, and invoking functionality across processes requires costly OS kernel mediation and application developer involvement to synchronize and exchange information through Inter-Process Communication (IPC) channels. Therefore, using processes as an isolation unit imposes performance and programmability overheads. Nonetheless, processes also serve two other necessary roles. First, they act as resource containers; OSs track resources like memory and open files at the process granularity. Second, processes provide in-memory persistence; using a process ensures that its state is consistent across the coming and going of other processes that communicate with it. The architectural foundations used for building processes impose performance overheads in the excess of 10× and 1000× compared to a function call; that is, privilege level and page table switches, respectively. Even more, part of these overheads are not attributable to the hardware itself, but to the inherent overheads imposed by current OS designs.

This thesis proposes a hardware and software co-design to eliminate the overheads of process isolation, while providing a path for gradual adoption for more aggressive optimizations. On the hardware side, this thesis proposes the CODOMs protection architecture. It provides memory and privilege protection across software components in a way that is at the same time very efficient and very flexible. This hardware substrate is then used to propose DomOS, which provides changes to the OS at the runtime and kernel layers to allow threads to efficiently and securely cross process boundaries using regular function calls. That is, a thread in one process is allowed to call into a function residing in another process without involving the OS in the critical communication path. This is achieved by mapping processes into a shared address space and eliminating IPC overheads through a combination of new hardware primitives and compile-time and run-time optimizations.

IPC in DomOS is up to 24× times faster than Linux pipes, and up to 14× times faster than IPC in L4 Fiasco.OC. When applied to a multi-tier web server, DomOS performs up to 2.18× better than an unmodified Linux system, and 1.32× on average. On all configurations, DomOS provides more than 85% of the ideal system efficiency.

Contents

1 Introduction **1**
- 1.1 The Granularity, Performance and Programmability Dilemma . . . 1
- 1.2 Conjoining Isolation, Performance and Programmability . . . 2
- 1.3 Document Organization . . . 4

2 A Comparison of Related System Organizations and Isolation Primitives **7**
- 2.1 A Note on Nomenclature . . . 7
- 2.2 Hardware Protection Mechanisms . . . 7
  - 2.2.1 Privilege Levels . . . 8
  - 2.2.2 Address Spaces . . . 9
  - 2.2.3 Machine Virtualization . . . 11
  - 2.2.4 Capability Architectures . . . 12
- 2.3 Operating System Organization Models . . . 16
- 2.4 Software Isolation Mechanisms . . . 17
  - 2.4.1 Processes . . . 17
  - 2.4.2 Address Spaces . . . 17
  - 2.4.3 Machine Virtualization . . . 18
  - 2.4.4 Software Fault Isolation . . . 18

3 The Interplay Between Isolation and System Design **19**
- 3.1 The Inadequacies of Process-Based Isolation . . . 19
- 3.2 Mismatch Between Procedure Call and IPC Semantics . . . 23
- 3.3 False Concurrency . . . 25
- 3.4 A Case for Configurable Isolation Policies . . . 26

4 Efficient and Composable Isolation Primitives **27**
- 4.1 Security Model . . . 28
  - 4.1.1 Asymmetric Isolation Policies . . . 28
- 4.2 System Design Overview . . . 29
  - 4.2.1 CODOMs . . . 30
  - 4.2.2 Compiler Support . . . 33
  - 4.2.3 DomOS . . . 33
- 4.3 Isolation Scenarios . . . 34

5 Hardware Support **37**
5.1 Code-Centric Protection . . . 37
5.1.1 Page Table Capabilities . . . 37
5.1.2 Domain Access Permissions . . . 40
5.1.3 Access Protection List . . . 41
5.1.4 Example of Code-Centric Protection . . . 41
5.2 Capability Protection . . . 42
5.2.1 Example of Capability Protection . . . 44
5.2.2 Capability Confidentiality . . . 45
5.2.3 Capability Unforgeability and Integrity . . . 45
5.2.4 Domain Capability Stack . . . 45
5.2.5 Capability Revocation . . . 46
5.3 Enforcing Domain Entry and Exit Points . . . 47
5.4 Implementation of Access Protection Checks . . . 48
5.5 Efficient Execution in Out-of-Order Pipelines . . . 50
5.6 Revisiting Isolation Scenarios . . . 50

6 Compiler Support **53**
6.1 Language Interface . . . 53
6.2 Implementation . . . 55
6.3 Example . . . 56

7 Operating System Support **59**
7.1 Processes and Threads . . . 60
7.2 Low-Level Isolation Interface . . . 61
7.2.1 Domain Management . . . 61
7.2.2 Entry Point and Cross-Domain Proxy Management . . . 61
7.2.3 Capability Management . . . 64
7.3 Runtime Support . . . 65
7.3.1 Program Loading . . . 65
7.3.2 Entry Point Resolution . . . 66
7.4 Implementation Details . . . 67
7.4.1 Unified Virtual Address Space . . . 67
7.4.2 Thread Management . . . 68
7.4.3 Thread-Local Storage . . . 68
7.4.4 Cross-Process Proxies . . . 69
7.4.5 Fault Notification . . . 69
7.5 Capabilities as Opaque Handles to User-Defined Objects . . . 69
7.6 Unified Resource Access Controls . . . 70

8 Evaluation **71**
8.1 Hardware Mechanisms (CODOMs) . . . 71
8.1.1 Comparison of Different Cross-Domain Call Mechanisms . . . 71
8.1.2 Effect of Isolation Policies . . . 74
8.1.3 Area and Energy Overheads . . . 75
8.1.4 Linux Kernel Module Isolation . . . 76
8.2 System Performance (DomOS) . . . 77

8.2.1 Comparison of Different Domain Communication Primitives . . . 78
8.2.2 Thread Isolation . . . 81
8.2.3 High-Performance Device Driver Isolation . . . 83
8.2.4 Multi-Tier Web Server Isolation . . . 84

9 Conclusions **87**

A FlowTAS: Making Data-Centric Mandatory Access Control Practical **89**
A.1 Introduction . . . 89
A.2 Motivation . . . 93
A.2.1 Example Setting and Privacy Expectations . . . 93
A.2.2 Data Leaks with Untrusted Applications . . . 93
A.2.3 Plugging Untrusted-Application Data Leaks . . . 93
A.2.4 Limitations of Prior MAC systems . . . 94
A.2.5 Problem Definition . . . 96
A.2.6 FlowTAS Overview . . . 97
A.3 Design of the FlowTAS System . . . 98
A.3.1 Trusted User Interface . . . 98
A.3.2 Web Proxy . . . 98
A.3.3 Container Manager . . . 99
A.3.4 Template Processor . . . 99
A.3.5 Storage Declassifier . . . 100
A.4 FlowTAS Implementation . . . 100
A.4.1 Object Categories for Data-Centric MAC . . . 100
A.4.2 Web Proxy and Container Manager . . . 102
A.4.3 Container and Application Instance Management . . . 102
A.4.4 Template Processor . . . 104
A.5 Security Properties . . . 105
A.6 Containers Optimized for Multi-Execution . . . 106
A.6.1 Baseline Container Implementations . . . 106
A.6.2 Intrinsic Container Performance . . . 107
A.6.3 The CODOMs architecture . . . 107
A.6.4 Thread-level Containers using CODOMs . . . 107
A.7 Developer Porting Experience . . . 109
A.7.1 Application Design from Scratch . . . 109
A.7.2 Porting Existing Applications . . . 112

B Tools **115**
B.1 QDBI: QEMU Dynamic Binary Instrumentation . . . 115
B.2 SciExp^2: Scientific Experiment Exploration Framework . . . 115

Publications **117**

Bibliography **119**

Acronyms **127**

List of Figures

2.1 Illustration of a cross-domain request using various hardware protection mechanisms. 8
2.2 Comparison of the characteristics of a web server and a backend database service for different OS organization models. 16

3.1 Comparison of the costs of different communication primitives, normalized to a function call. 22
3.2 Anatomy of a RPC sequence. 23

4.1 Overview of the relationship between the different parts of the proposed system. 30
4.2 Comparison of CODOMs with other cross-domain request mechanisms. 31
4.3 Comparison of DomOS with other OS organization models. 34

5.1 The CODOMs architecture. 38
5.2 Location of added and relevant fields in the page table structures. 39
5.3 Example of code-centric protection in CODOMs. 41
5.4 Access protection check logic for memory accesses. 49

6.1 Example of source code annotations. 57

7.1 Domain contents and CODOMs configuration of the example application of Figure 6.1. . . 65

8.1 Comparison of domain switch overheads for CODOMs and other mechanisms. 73
8.2 Comparison of domain switch overheads for different isolation policies in CODOMs. 74
8.3 Distribution of memory accesses according to the owner domain of the accessed memory. . . 77
8.4 Performance of cross-domain calls using DomOS for different caller/callee isolation policies. 79
8.5 Execution time breakdown of a cross-domain call/return in DomOS. 80
8.6 Performance of isolating computations based on the data they operate with. 81
8.7 Bandwidth and latency overheads when isolating the Infiniband driver in three scenarios: DomOS, user-level processes (IPC) and kernel driver (user/kernel isolation). 83
8.8 Performance of different dynamic web server configurations using vanilla Linux and DomOS. 85

A.1 Application-centric access control in current systems. 90
A.2 High-level overview of the TAS (`template-app-storage`) pattern. 91
A.3 FlowTAS design pattern and infrastructure overview. 95
A.4 Start phase: sequence of operations for starting a new `app` instance. 103
A.5 Gather phase: sequence of operations for rendering cross-folder results. 103
A.6 Enter phase: sequence of operations for executing an application in a folder. 104

A.7 Isolating two threads (*T1* and *T2*) during a gather operation using a combination of CODOMs and SELinux. 108
A.8 Sequence of steps for a thread to execute the untrusted code of the *Health* application. . . . 108
A.9 Writing a cross-folder view for Gitlab. 110

List of Tables

3.1 Costs of common state isolation/recovery mechanisms normalized to a regular function call. 24

7.1 Domain-management operations in the DomOS kernel. 62
7.2 Entry point-management operations in the DomOS kernel. 62
7.3 Capability-management operations in the DomOS kernel. 64

8.1 Configuration of the simulated architecture for CODOMs. 72
8.2 Micro-benchmark execution parameters. 72
8.3 Average energy overheads of various mechanisms relative to a function call/return. 75
8.4 Number of domain switches (calls & returns) during the benchmarks' execution. 76
8.5 Runtime overheads incurred when considering each kernel module a separate domain. 77
8.6 Configuration of the evaluation machines for DomOS. 78
8.7 Configuration of the evaluation machine for the thread isolation experiments. 81

A.1 LOCs removed from the Trusted Computing Base (TCB) with FlowTAS (including third-party libraries). 112
A.2 LOCs we added and removed from off-the-shelf apps to enable FlowTAS functionality, and LOCs removed from the TCB with FlowTAS (including third-party libraries). 112

Chapter 1

Introduction

The security and reliability of computing systems are ever growing concerns in today's networked computing world. Complex software systems, ranging from financial services to mobile phones, have increasingly important roles in most facets of our lives. These systems have seen a very large growth in software complexity and are increasingly used for security-sensitive tasks. Whether they are handling sensitive private data such as passwords, banking information, corporate financial data, or user credit card numbers, any insecure or unreliable system is an instant threat of financial and personal calamity.

Complex software systems contain a multitude of software components: fragments of code with their associated internal data that communicate through well-defined interfaces. Such components include individual functions, libraries, multiple plugins in a single application, separate applications, or separate device drivers in an Operating System (OS) kernel; they may be as tiny as a handful of instructions or as large as an entire complex application or driver. Each of these components can be unreliable, or may distrust other components. Enforcing inter-component isolation can thus greatly enhance system security and reliability by, for example, preventing the spread of faults across multiple components, or by keeping private secrets such as encryption keys outside the reach of compromised components. System security and resiliency thus require that individual software components be isolated in separate *domains*.

1.1 The Granularity, Performance and Programmability Dilemma

Conventional systems impose large *performance* and *programmability* overheads when isolating components. Importantly, when performance and isolation are at stake, performance often takes precedence at the expense of security and reliability. Programmers employ coarse-grained isolation domains, running many distrustful and unreliable software components on the same domain. Functions are all located on the same code segment; plugins all run in the same address space; kernel drivers all run at the highest privilege level. The improved security and reliability that could be had through finer-grained isolation remains unattained.

These overheads are rooted at the co-evolution of conventional architectures and OSs, which expose isolation in terms of a loose "virtual CPU" model. Most architectures provide isolation through a combination of *privilege levels* and *page tables*. In turn, OSs expose isolation domains to users in the form of *processes*, which embody this virtual CPU model. The OS kernel is isolated from user code by running at a privileged level, from where it can manage external devices and perform privileged operations that are unavailable to user processes. At the same time, user processes are isolated from each other through the utilization of different page tables, which function as separate domains. The OS kernel then multiplexes these processes

across the available physical resources, providing processes the illusion of having a machine for their exclusive use. Since processes provide such virtual CPU model, they must usually interact through interfaces designed for networked distributed systems, making their programming more cumbersome.

An idealized *microkernel* is a clear example of bringing isolation of native code components to its inevitable conclusion [22, 27, 33, 53, 56, 76, 78, 86]. Each device driver, system service and user application runs as a separate process isolated from the rest. But as aforementioned, this comes at the expense of performance and programmability:

- Existing hardware isolation mechanisms are expensive. Switching between privilege levels (i.e., user and OS kernel) and page tables (i.e., processes) imposes non-negligible performance overheads [73, 78, 116] that can go beyond 10× and 1000× of the cost of a regular function call, respectively.
- The process isolation model is conservatively designed for the worst case: every process is completely isolated from each other for security and reliability purposes. This adds unnecessary overheads in cases where the semantics of an application do not require this level of isolation.
- Processes must use Inter-Process Communication (IPC) primitives to communicate with each other. Since processes are mutually isolated (acting as a networked system), the privileged OS kernel must mediate communication through its IPC primitives.
- Processes impose concurrency. Since threads are bound to processes, IPC triggers a control transfer across threads. This breaks the synchronous execution model assumed by the programmer, who must then wrap IPC calls to regain her synchronous model.
- IPC primitives are designed for generality to cater to workloads with diverse needs. This generality makes them necessarily inefficient. For example, IPC primitives require dedicated data (de)serialization, memory copies and request (de)multiplexing routines that can amount for more than a 1000× overhead compared to a regular function call.

The more processes a system uses to isolate software components, the more reliable and secure it can be but, at the same time, the more performance and programmability overheads it has to pay for such desirable features. Therefore, achieving high isolation and performance levels while maintaining the so desirable synchronous semantics of function calls requires a combined effort on both the hardware and software sides of the system.

1.2 Conjoining Isolation, Performance and Programmability

This thesis goes after two complementary targets, which can be applied to all kinds of systems (e.g., mobile, servers and High-Performance Computing (HPC)):

- Increase isolation without impacting performance. By gradually partitioning existing systems into an increasing number of efficient isolation domains, systems can increase their security and reliability. Examples include isolating OS kernel modules (Section 8.1.4) or high-performance user-level drivers (Section 8.2.3) into their own domains.
- Increase performance on systems with existing isolation. By making isolation primitives more efficient, systems that already have isolation in place can experience a performance boost. Examples include speeding up high-assurance systems (Section 8.2.2) and server environments (Section 8.2.4).

Since both targets are complementary, they can be gradually applied to the same system. To achieve this, programmers must be able to easily partition their systems into many isolation domains; like in a *microkernel* system, each component should be isolated from the rest [22, 27, 33, 53, 56, 76, 78, 86]. But this must come with the same performance and synchronous function call semantics that are provided by non-isolated software components; like in an *exokernel* or *library OS* system, both user and system components should interact through simple function calls [18, 44, 65, 66, 82, 92].

A very important observation that can be taken from existing systems is that different software components have different isolation and concurrency requirements. For example, the UNIX idea of processes concurrently streaming data to each other is very convenient to decouple and parallelize the stages of data processing pipelines, but does not fit applications that synchronously request an operation to another service. A typical example of such applications would be a web server performing a query to a database server; the web server computation for a request cannot continue until the database returns its result. Application plugins are an example of different isolation requirements. Plugins should be isolated from the rest of the application and other plugins, but there is no reason why the application itself should not be allowed to directly access the plugin's resources; this is something that processes prohibit.

This thesis describes the design and implementation of the *CODOMs* architecture (for *COde-centric memory DOMains*), which provides efficient memory and privilege protection, and the *DomOS* system (for *DOMain OS*), which maps its abstractions into that architecture. They have been co-designed with the following goals in mind:

- Offer efficient component isolation.
- Maintain existing programmability through a synchronous function call interface for cross-domain requests.
- Provide an expressive and composable set of isolation primitives to ease the implementation of disparate isolation policies with minimal performance overheads.
- Ease gradual adoption on existing systems.
- Be general enough to be applied across the whole software stack.

To this end, CODOMs and DomOS stress the concept of concern separation to allow programmers to express their own component isolation policies across its full spectrum. Two common policies exemplify the disparity of possibilities in this spectrum. On one hand, simple hierarchical isolation, like the one exhibited by applications and their plugins or user applications and the OS kernel, makes one domain fully accessible to the other. On the other hand, full-fledged mutual isolation, like the one provided by processes, makes two domains completely inaccessible to each other.

CODOMs provides the low-level mechanisms to enforce fine-grained protection (i.e., memory and privilege isolation) in a very efficient and flexible way. On one hand, regular call/return instructions can trigger domain and privilege level switches at negligible latency through CODOMs' *code-centric* isolation domains. On the other hand, CODOMs facilitates in-place data sharing across domains using application-controlled *capabilities*. Given its baseline efficiency, CODOMs avoids optimizing common operation sequences into a single instruction to maximize the flexibility of its mechanisms. This keeps the architecture free of predefined isolation models, leaving programmers to build their own.

DomOS provides the OS primitives that programmers use to define component isolation domains and build component isolation policies and communication models. At its lowest level, DomOS' primitives are mapped into the CODOMs architecture. Processes in DomOS are mapped to a global virtual address space,

and are no longer used as isolation primitives. Instead, a thread from one process can simply call into a routine of another process, eliminating all the unnecessary overheads imposed by process-based isolation and without involving the OS kernel in the call. Essentially, an isolation policy is a sequence of steps that enforce certain properties when control crosses a domain boundary. To maximize efficiency and flexibility, programmers can define their own per-domain policies. Applications benefit from the fact that most of the steps in a policy can be more efficiently implemented at user-level, without losing its isolation properties. When necessary, DomOS generates optimized code at run-time to enforce the isolation of privileged resources as requested by the communicating domains. This model leaves concurrency to programmer discretion (unlike traditional processes that impose concurrency among domains), and using CODOMs' capabilities avoids intermediate copies during cross-domain calls. Programmers are still free to use concurrency and perform copies, but these are no longer a side-effect of the isolation model imposed by the OS. Finally, DomOS ships with a thin compiler wrapper and run-time that simplifies the construction and management of the isolation domains.

Attacking a problem from multiple layers bears a higher potential for large benefits by co-optimizing the system. But more interestingly, it helps uncovering the subtle relationships that exist among layers, and how they influence each other. The evaluation of the proposed design shows up to 24× speedup for cross-domain communication compared to Linux pipes, and up to 14× compared to IPC calls in the L4 Fiasco.OC microkernel. Furthermore, a multi-tier web workload provides up to 2.18× speedup over an unmodified Linux system, and 1.32× on average. On all configurations, these results provide more than 85% of the ideal system efficiency.

1.3 Document Organization

Chapter 2: A Comparison of Related System Organizations and Isolation Primitives Presents some of the many efforts that have gone into hardware and software system design in order to make isolation more efficient. It discusses the strong and weak points of those works that are more closely related to this thesis.

Chapter 3: The Interplay Between Isolation and System Design Discusses the effects that OS design has on user applications, as well as discusses and quantifies the various issues that make processes necessary in current systems but, at the same time, ill-suited for the role they fulfill as an isolation primitive. This analysis leads to the conclusion that co-designing hardware and OS primitives has the best potential in yielding better performance results and programmability.

Chapter 4: Efficient and Composable Isolation Primitives Provides a technical overview of the design and relationship between the pieces that conform this thesis. Namely, CODOMs, DomOS and the associated compiler support.

Chapter 5: Hardware Support Describes the design of the CODOMs architecture, which provides the efficient hardware protection mechanisms used by other software layers.

Chapter 6: Compiler Support Describes the thin compiler support that helps programmers define isolation domains and their relationship. The generated code makes programming multi-domain applications simpler by mapping them to the underlying OS primitives.

Chapter 7: Operating System Support Describes the design of DomOS, which provides the necessary system primitives for programmers to define their isolation domains. The OS maps these primitives to the underlying CODOMs architecture to offer efficient isolation.

Chapter 8: Evaluation Shows a quantitative comparison of the proposed changes to some of the relevant state of the art. It also evaluates the potential benefits of such changes on large non-trivial systems.

Chapter 9: Conclusions Draws some conclusions for this thesis and points to some of the interesting open roads for future work.

Appendix A: FlowTAS: Making Data-Centric Mandatory Access Control Practical Describes the design of a secure infrastructure for cloud applications managing security-sensitive information. Using the ideas proposed in this thesis, the infrastructure can achieve an order of magnitude speedup compared to state-of-the-art isolation mechanisms.

Chapter 2

A Comparison of Related System Organizations and Isolation Primitives

Software isolation is a two-axis problem. On one hand hardware has a huge impact on isolation efficiency, since it defines what software organizations can be efficiently expressed through its primitives. On the other hand, the underlying OS or system model defines how hardware resources are exposed to applications. The OS defines the system abstractions that software components must adhere to, how they can communicate, and how they can be isolated from each other. Therefore, OS design also plays a major role on isolation efficiency, since isolated components in a real system need to communicate beyond the boundaries of a single application.

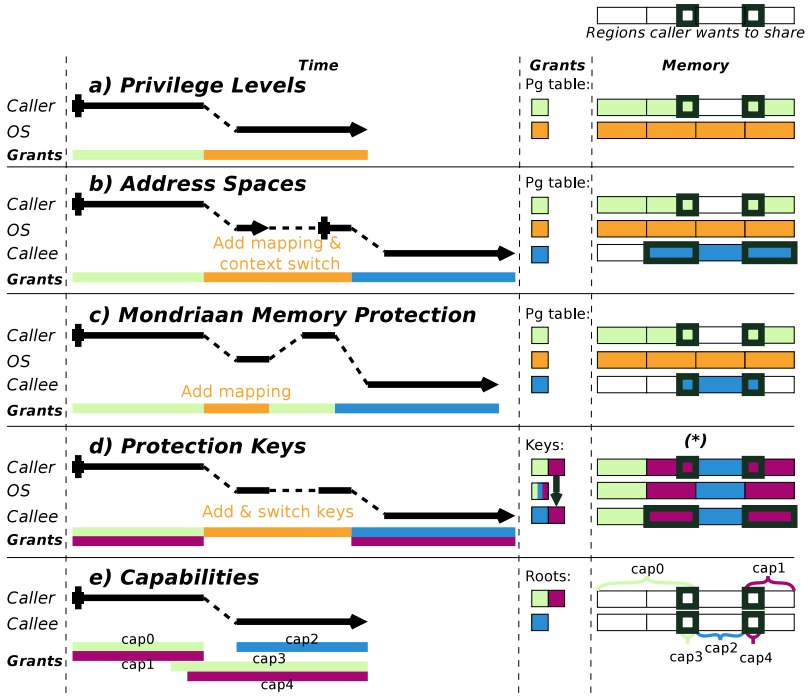
2.1 A Note on Nomenclature

This thesis repeatedly uses the terms *protection*, *isolation* and *domain*, each with a slightly different meaning. Protection refers to hardware resources such as memory accessibility and the ability of accessing privileged hardware resources; i.e., memory protection and privilege levels. Isolation refers to the higher-level model exposed to software components, like isolating the OS kernel from its users, isolating an application plugin or isolating applications from each other. Finally, the domain concept refers to an isolated unit, and can be applied to both protection and isolation. The extent of a domain is defined as the transitive closure of resources it can access without switching to another domain.

2.2 Hardware Protection Mechanisms

Hardware protection mechanisms provide the basic blocks for software to build its isolation abstractions and enforce its policies. The design of all hardware mechanisms is a result of a trade-off between three main axes: (1) the complexity of defining and switching between domains, (2) the performance of switching between domains, and (3) the performance of sharing data between domains. These hardware mechanisms are managed by the TCB code, which must be protected from other code in order to maintain the integrity of the system isolation policies. For example, the OS kernel is part of the TCB, since it manages privileged hardware resources like the page table pointer. This section discusses commercial and academic hardware

protection mechanisms while focusing on these three main axes. Figure 2.1 illustrates some of these mechanisms in a scenario where one domain (the caller) invokes some functionality on another domain (the callee) which requires access to two buffers in the caller to perform its operations.



(*) Caller and callee execute within the same page/protection table.

Figure 2.1: Illustration of a cross-domain request using various hardware protection mechanisms. The "caller" (or "client") domain passes two data buffers as arguments to the "callee" (or "server") domain (identified in the top-right corner). The *Time* column illustrates the stages of a domain crossing, indicating the grants (regions of memory) available at each stage at the bottom. The *Grants* column shows the per-domain configuration of each mechanism. The *Memory* column shows the layout (and grants) from the point of view of each domain.

2.2.1 Privilege Levels

Privilege Levels (or protection rings) are one of the basic mechanisms to enforce isolation between domains, as depicted in Figure 2.1a. Its use was introduced by Multics [102], providing hardware support for eight of them. This was a multi-level generalization of the *supervisor mode* concept, used by system-level software (the OS kernel or TCB) to isolate itself from user-level software. Multics defined a privilege level as a descriptor for a set of capabilities[1] hold by the currently executing domain, where level N contained, at least, all the capabilities of level $N + 1$. That is, each privilege level was a separate isolation domain and each domain could, at least, execute all the instructions and access all the memory available to domains with

[1] Note that the term "capability" is used here to refer to the ability to execute certain instructions and access resources like memory.

a higher privilege level number. Multics also provided *call gates*, a mechanism used to request a call or a return into a different privilege level [100]. Privilege levels in current Intel x86-based processors identify which subset of instructions and virtual memory is currently available to the executing code. This is used by the OS kernel to forbid user processes from accessing the kernel memory, and to limit access to instructions that could otherwise subvert the security policies that the OS kernel is trying to enforce.

Limitations

Privilege levels provide strong isolation guarantees by placing each domain on a separate level, but have their own limitations:

- Their number is limited by hardware.
- Switching between privilege levels is a relatively expensive operation when compared to a simple function call. In most cases, it hampers Instruction-Level Parallelism (ILP) by inducing a pipeline flush. This is because all younger instructions have a Read-After-Write (RAW) dependency against the micro-architectural register that holds the current privilege level. Furthermore, instructions typically used to switch between privilege levels (like `int` or `syscall` in x86–64) not only switch between levels, but also have other (visible) side-effects like changing the values of other registers, adding to their latency.
- Privilege levels define a total order of the capabilities available to each of them. That is, isolation based on privilege levels only works in one way, as the more privileged level holds all the capabilities of all less privileged levels, as can be seen in the "callee" line of Figure 2.1a. Sharing in the opposite way requires data copies or changing page permissions (see below). Therefore, they are not appropriate for cases where pairs of components do not have a hierarchical relationship; e.g., privilege levels cannot be exclusively used to isolate two processes from each other.
- The use of call gates hardwires isolation policies into the architecture. For example, call gates in x86 provide a function call interface, but also define that each side of the gate should use a different stack to avoid one domain interfering with the other through the stack. Therefore, a call gate changes the privilege level and what code and data segments are accessible on each side of the gate, but also switches between different stacks and copies arguments across them. While handy on situations of symmetric (mutual) distrust between domains, a stack switch is not necessary in asymmetric isolation scenarios.

Modern implementations for privilege level switching like Intel's `syscall` provide faster switch times compared to previous implementations, but at the expense of even simpler models. First, it only supports two privilege levels, user and OS kernel level. Second, it drops the concept of call gates and instead forces the OS kernel to demultiplex requests through a single entry point.

2.2.2 Address Spaces

Address spaces (or memory virtualization) are the second pillar for constructing processes in modern OSs, the first being privilege levels. They are depicted in Figure 2.1b. Hardware enforces address space isolation by translating all (*virtual*) memory addresses generated by code into their respective *physical* memory address, and generates an exception if no such translation exists. There are two mechanisms for virtualizing memory addresses:

Paging was first introduced in the Atlas Computer [51]. It divided memory into fixed-size pages of 512 words each, and translation information was stored in a Content-Addressable Memory (CAM).

Segmentation was first introduced in the Burroughs B5000 [79]. Every memory operation was performed relative to a segment descriptor that encoded the actual physical address and length of the segment.

Memory virtualization and protection do not necessarily come hand by hand, but it is common to mix both in a single mechanism. The data structures describing memory virtualization can be made inaccessible to regular user code, making address spaces the basis for memory protection. The introduction of memory virtualization simplified storage management, relieving applications from the burden of managing the multiple levels of storage. But it also was key to efficiently and securely support multiprogramming. Each application (each process, in fact) is set up as a separate protection domain, and the memory virtualization mechanisms ensure they do not interact with each other's physical memory unless they are explicitly allowed to do so.

Limitations

A few shortcomings prevent their use as the basis for fine-grained domain isolation:

- Page table and segment switching is a necessarily protected operation. Therefore, the OS kernel has to mediate the operation in order to enforce the correctness of its security policies ("OS" line in Figure 2.1b).
- Page table switches also impose large performance overheads due to hardware design. At the very least, they impose a pipeline RAW dependency that negatively impacts ILP because all instructions depend on the register that identifies the current page table. These overheads are usually larger due to additional micro-coded operations. Also, non-tagged Translation Lookaside Buffers (TLBs) require flushing their entries to forbid the new domain from using stale cached translations pertaining to the old page table. In comparison, tagged TLBs avoid flushes as long as tags do not need to be multiplexed across different page tables. Nonetheless, they add lookup latency costs for non-trivial associativities and tag sizes, making them not viable because TLB lookups are on the critical path.
- Page tables limit the minimum granularity at which memory can be shared between domains; i.e., the callee in Figure 2.1b is shared two entire pages instead of the buffers passed by the caller.
- Memory access grants and revocations are costly operations. Both require modifying the page table, which (as page table switches) requires OS intervention. Revocations also require costly TLB shoot-down operations to maintain the consistency of translations cached in the TLB across all CPUs using the same page table [121]. Revocation costs can be coarsely reduced to Inter-Processor Interrupts (IPIs) and TLB entry invalidation costs on both local and remote CPUs.
- Segmentation does not suffer from the granularity problem, but they can only describe consecutive memory ranges and the number of active segments is limited by hardware. The system must therefore multiplex all logical segments among the ones provided by the hardware. Since segment switches are protected operations, the OS kernel must mediate them.

Refinements to Page Table-Based Protection

To alleviate some of the drawbacks in page table based protection, other architectures (comercial and academic alike) have provided very interesting refinements:

Protection Tables decouple memory protection from address translation [67]. They were proposed to allow processes to run on a single address space, where the OS kernel manages address translation through the page table and TLB, and manages protection through an additional the protection table and Protection Lookaside Buffer (PLB). The PLB is a separate hardware TLB-like structure that maps virtual page addresses and protection domains to access permissions, where entries are tagged with the protection domain identifier.

The PLB exhibits the same limitations found on paging and tagged TLBs.

Mondriaan Memory Protection (MMP) builds on the PLB concepts [124, 125], but instead provides memory protection at arbitrary granularity, as shown in Figure 2.1c. MMP also supports unsupervised domain switches, at the cost of an additional in-memory entry point table and a hardware CAM cache (the GLB) that controls the ability to switch domains at call/return boundaries.

While MMP has the benefit of fine-grained protection, it also incurs in all the other limitations found on paging and tagged TLBs when switching between protection tables and managing protection grants and revocations. Furthermore, to keep hardware-assisted domain switches fast, the architecture hardwires semantics that control grants to the data stack.

Sentry proposes a scheme conceptually similar to MMP with cache line granularity, but with an entirely different implementation [106]. Instead of performing one permission check per access, accesses to any line present in the L1 are always permitted, as long as its C (for correct) bit is set. Cache misses and accesses to lines with an insufficient coherency state perform a lookup on an ancillary software-managed table that caches domain permissions, and set the C bit for that line.

This implementation is quite performance- and complexity-efficient for the average memory access case, since it does not require modifying the processor pipeline nor the cache coherency protocol. Nonetheless, domain switches require resetting the C bit of all lines. More importantly, domain switches can only be managed through exceptions; i.e., a call to an address of another domain raises a user-level exception that signals a domain switch to the (user-level) TCB and resumes execution on the target address.

Protection Keys have been implemented in commercial processors like PA-RISC [67], Itanium [55] and POWER 6 [61], and are shown in Figure 2.1d. Every page table entry contains a tag identifier, and a small key set implemented in hardware (*Grants* column) describes the tags that can be accessed at any given time.

The key set is a privileged resource that can only be managed by the OS kernel. Also, the number of tags is limited by hardware and multiplexing them requires changing the corresponding page table entries and expensive TLB shootdown operations.

2.2.3 Machine Virtualization

The late implementation of machine virtualization hardware in off-the-shelf processors has renovated the interest in using virtualization for reliability and security purposes. Even though implementation details are vastly different, it follows the same concept of process isolation through a mix of privilege levels and address spaces.

Just as the OS kernel manages privileged machine resources, the trusted hypervisor manages the privileged machine virtualization resources. A hypervisor is typically simpler than a general-purpose OS kernel, and is thus easier to ensure the correctness of the TCB (in this case, the hypervisor).

Limitations

- Precisely because the hardware implementation is much more complex, domain switches are very expensive. This applies both to calling a more privileged level (hypercalls) and to switching between virtual machines.

Bastion provides an extension to maintain data confidentiality and integrity in the face of intermediate untrusted software stacks [31]. It tags pages at the hypervisor level to identify protected components, and encrypts and hashes memory that interacts with other protection domains, including I/O devices.

Its confidentiality and integrity properties are largely orthogonal to isolation. Furthermore, it not only incurs the overheads imposed by virtualization; encryption and hashing (whose information is interspersed with code and data) also delays memory accesses and induces poor memory bandwidth utilization.

2.2.4 Capability Architectures

Capabilities are communicable and unforgeable tokens that at the same time authorize and identify the destination of an operation. Although early architectures provided similar structures, the term was coined by Jack Dennis and Earl Van Horn in a paper that generalizes their use in the construction of an OS kernel for a multiprogrammed system [40]. In a *capability addressing* architecture, capabilities are implemented in hardware and are typically used to grant access to ranges of memory that contain code, data and/or other capabilities. A typical analogy is that of "fat pointers"; when used to access memory, hardware capabilities identify a *range* of addresses and authorize specific permissions to that range. Any access outside this range or requiring more permissions than granted is simply not authorized. For example, one capability can grant read-only access to a buffer, while another can grant read-write access to the same buffer. Therefore, capabilities can be thought of (and in fact implemented as) handles to memory segment descriptors. Capabilities must be loaded from memory into a register in order to use them, and then user code can index memory accesses through them. Instructions are also provided to manipulate the extents of a capability, as well as the permissions it grants. Some systems also use hardware capabilities to interact with higher-level abstractions implemented by the hardware, as will be described later.

The extents of a protection domain are defined as the transitive closure of an initial set of "root" capabilities (i.e., the set of capability registers), from which memory and other capabilities can be accessed. Therefore, new domains can be created by sub-dividing a large segment into smaller ones. This allows users to implement systems where trust between domains is organized in a distributed and non-hierarchical manner.

Several capability-based architectures have been proposed in the last 40–50 years, both for research and commercial purposes [28, 72, 126], but all have been exposed to the following concerns:

Paging vs. segmentation: Early systems implemented capabilities as segments to physical memory. On larger and more complex systems this posed a problem when managing external memory fragmentation. The IBM System/38 later used segmentation of virtual (paged) address spaces instead. This traded physical memory fragmentation for virtual memory fragmentation. Since the virtual address space is much larger than the physical one, the fragmentation problem is less pressing.

Domain switching: A domain switch in a capability architecture requires a switch of the "root" capabilities. A typical approach is to implement domain switching instructions in hardware or microcode to make the operation as fast as possible by avoiding a call into the OS kernel. Systems like CHERI [126] or the

one presented in Carter et al. [28] limit roots to a single capability. An "enter capability" grants access to a well-defined entry point in another domain. When calling into that entry point through the enter capability, the hardware changes some capability registers to those identified in the entry capability, granting access to the memory region that contains the entry point of that domain. By allowing code, data and capabilities to be mixed in the same memory region, other capabilities can be loaded from it to access other memory regions that conform the same domain (remember that a domain is defined as the transitive closure of its "root" capabilities).

Capability unforgeability and integrity: Hardware must ensure that unprivileged code cannot create arbitrary capabilities (they are unforgeable) nor can tamper with the contents of capabilities (their integrity is ensured). Failing to do so allows bypassing memory protection. Capability register unforgeability is very easy to enforce: new capabilities can only be created from other capabilities, either by copying them or by loading them from memory (the OS kernel can create arbitrary capabilities). Likewise, capability register integrity is also easy to enforce: capability-modifying instructions must never allow the amplification of the permissions or memory range that they encode (although it is safe to allow their reduction). Enforcing the unforgeability and integrity of capabilities stored in memory is a bit more complex, and architectures have used one of two main approaches:

- The first approach is to keep data and code segregated from capabilities, and making the latter not accessible by unprivileged code. Dennis termed these as *C-lists* [40] (for capability lists), which saw their way into commercial processors in the form of Intel's segment descriptor list. Each segment descriptor was referenced as an index to this list. The Chicago Magic Number Machine [72] provided a more sophisticated implementation. Each segment descriptor (a capability) had a bit indicating whether its memory stored capabilities. Unforgeability was trivially enforced by disallowing the load of capabilities from memory segments without the capability storage bit. Likewise, enforcing their integrity was as simple as disallowing raw memory modifications to segments with the capability storage bit. This also allowed software to construct more complex C-list organizations, where any list could contain capabilities pointing to another list, instead of relying on a single global array.
- The second approach is byte-level memory tagging[2]. Architectures like the IBM System/38 [72] only allowed capability-related operations on bytes with the tag set, and non-capability operations on bytes without the tag set. To allow memory to be re-purposed, a store into memory set the tag bits according to whether the source operand contained a capability.

Capability revocation: The common understanding of capabilities is that of granting access to a logical structure defined by the programmer. Structures are allocated in specific virtual memory addresses, and virtual memory is typically reused across multiple allocations in order to minimize internal memory fragmentation and, to some extent, virtual memory space usage. In turn, most capability addressing architectures grant access to virtual memory addresses, not logic structures. Therefore, failing to "destroy" a capability (i.e., *revoking* it) before reusing the addresses of the data structure it points to, will let others with that capability have access to a structure different than the one that was originally intended. For example, a web server might use a capability to grant access to a buffer where one of its plugins will generate results. If the web server then reuses that buffer to store some encryption keys without revoking the capability, the plugin will then have access to the private encryption keys. Note that not revoking capabilities after reusing some memory might be safe in some situations, depending

[2]Tag granularity can be optimized by disallowing unaligned capability loads and stores, and keeping tags at the granularity of the storage size of a single capability.

on the application's semantics. Therefore it is not always mandatory to revoke a capability whenever memory is reused.

Revocation has two properties that are not necessarily always satisfied:

Immediate: Revoke capabilities to a specific memory region at any point in time even if they are still being used.

Selective: Only revoke capabilities (to a specific memory region) that were granted to specific domains. For example, the web server example above might decide to revoke all capabilities to its buffer that were granted to a specific plugin, but not to other plugins.

A typical approach is to garbage-collect memory allocations and capabilities, so that a byte of virtual memory cannot be reused until there is no capability pointing to it in the system. An alternative approach is to add a revocation table to capabilities; every time a capability is used, it is indirected to check if it has been revocated in the revocation table.

In contrast, Multics [72, 100] provides a different "back pointers" mechanism. Each "object" in the system has a list of capabilities pointing to it, and every time a new capability is created for an object, the list must be updated. When an object is "destroyed", the system goes through its capability list to invalidate each of the capabilities pointing to it.

Limitations

- Segregating regular memory from capabilities at the segment level made it harder for older compilers to manage both. On the opposite side, tagging memory words to enforce capability unforgeability and integrity can have a negative effect on memory space and bandwidth utilization.

- Sharing non-consecutive regions of memory requires setting up multiple capabilities. For example the caller in Figure 2.1e has the two roots *cap0* and *cap1* where each of the argument buffers is contained; therefore it must create *cap3* and *cap4* from these roots to only pass the intended argument buffers to the callee. This makes their management much more complex, and increases the memory overheads when capabilities occupy more memory than a regular pointer. Code and data can be carefully laid out in memory to keep it consecutive and accessible through a single capability, but that is not always feasible for dynamic data structures or when dynamically loading code into an existing domain. In the worst case, each node in a linked-list or tree could be pointed to through a separate capability. In the case of dynamically loaded code this can be even more complex, since each page would potentially have to use a different capability for its code if pages from different domains are interspersed.

  Carter et al. [28] addressed the memory overhead problem by making capabilities and regular pointers occupy the same space, at the expense of limiting capabilities to memory ranges aligned according to their grant size.

- Switching between domains adds the overhead of switching the root capabilities, and the target (callee) domain cannot start executing until the root capabilities have been switched.

  Explicitly managing root capability switches in software is a complex operation and burdens the programmer. An alternative is to manage it through the OS kernel using some form of exception, making it very flexible at the expense of performance. Another alternative is to automatize root capability switches through hardware (see next bullet). Still, this comes at the expense of embedding domain organization and switch semantics into the hardware, and is an operation that potentially requires

multiple cycles. In all cases, switching capability roots necessarily hinders ILP by introducing RAW dependencies in the pipeline.

- Embedding high-level semantics into the hardware constricts higher-level software to the few primitives envisioned by the hardware architects. If we look at isolation, embedding certain semantics into hardware mechanisms can make them insufficient or too heavyweight.

  Domain switches are a clear example of this. The Plessey System 250 [72] went as far as embedding the concept of processes and inter-process calls into the hardware, and Intel's iAPX 432 [72] even extended hardware reach to process scheduling. These implementations make the targeted operations much faster than a pure software implementation through the OS kernel. However, they make it harder to implement system organizations that deviate from the organizations and semantics intended by the hardware architects. For example, whether a separate stack and register state should be used on different domains depends on the trust they have with each other; a trusted memory encryption domain could use the same stack and register state of its caller as long as the caller cannot interfere with it while the encryption domain is executing. Therefore, no isolation policy should be imposed by the hardware mechanisms.

- Revoking the access granted by a capability is a complex operation. Depending on the implementation, support for revocation can have a negative performance impact during regular execution of code that does not perform any revocations.

  Garbage-collection is a non-trivial task; capability and regular memory life-cycle must be tracked in coordination between the OS and user code, inducing performance overheads, and not immediately reusing virtual memory addresses induces larger internal memory fragmentation. Furthermore, it does not support immediate nor selective revocation.

  Using a revocation table supports immediate and selective revocation, but checking the revocation table imposes additional latency every time a capability is used. It also makes capability creation more complex, since an entry must be created in the table. Furthermore, an entry in the table cannot be reused until the system ensures no capability pointing to it remains in the system.

  Using the back pointers mechanism makes capability creation even more expensive, since a list must be updated, and the costs of revocation scale with the number of capabilities pointing to an object. Furthermore, this approach does not support selective revocation; one can either revoke all capabilities to that object, or can revoke none of them.

Language and Compiler Support to Manage Capabilities

To simplify the life of programmers in capability addressing architectures, the language and the compiler must understand their existence to automatically manage them and minimize programming burden. The lack of such compiler support has typically been argued as a longstanding roadblock for capability adoption, specially when applied to existing low-level languages like C.

Fortunately, the latest iteration of the recent CHERI project has shown very exciting progress in this area [34]. The project shows very positive results in securing C codebases against pointer errors through minimal extensions to their previous CHERI hardware prototype; e.g., to eliminate vulnerabilities like buffer overruns. This new prototype was developed as a response to their previous experience and a careful examination of the C language specification and common coding idioms found on a large selection of public codebases. The CHERI compiler is now able to automatically use capabilities as pointers, or through minimal

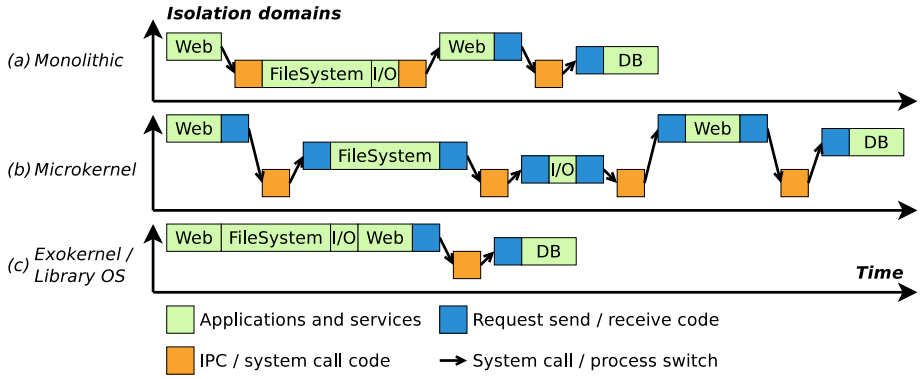


Figure 2.2: Comparison of the characteristics of a web server and a backend database service for different OS organization models.

code annotations when mixing capability-aware code with other code that is not. Therefore, the management of capability-protected arguments passed across domains can be offloaded to the compiler.

2.3 Operating System Organization Models

Figure 2.2 illustrates a web server example to highlight the performance and isolation differences between common OS design models. Upon receiving a request, the web server dynamically generates a web page by first reading data from disk and then querying a backend database service.

In a traditional *monolithic kernel* system (Figure 2.2a), the web server and database are isolated as separate processes. OS services run inside the privileged kernel, and are thus unprotected from each other. The web server must first retrieve its data from the disk through a system call, while the file system and disk I/O subsystems use regular function calls to interact. Next, the web server must issue a query to the database process, which must be actively listening for requests to service. Since both reside in different processes, the programmer must adapt code from a regular function call for requests and responses to the interface provided by IPC primitives (blue areas).

In an idealized *microkernel* [22, 27, 33, 53, 56, 76, 78, 86] system (Figure 2.2b), OS services run as separate concurrent user-level processes. This minimizes kernel code, and potentially provides higher flexibility, security and reliability [59]. As a result, all OS services and the web and database servers must listen for incoming requests and communicate with each other through IPC.

In an *exokernel* [44, 65] or *library OS* [18, 66, 82, 92] system (Figure 2.2c), OS services can run inside the application processes, at the expense of eliminating isolation. This potentially simplifies code because no IPC calls are needed, just regular function calls. It also improves performance mainly because hardware isolation is bypassed, but also because service implementations can be tailored to the application at hand. Nevertheless, the web server still incurs IPC overheads, since the database may service other clients and thus runs as a separate process.

2.4 Software Isolation Mechanisms

Software isolation mechanisms provide abstractions that software can use to build isolation domains. These abstractions are usually mapped into the lower-level memory protection offered by hardware, even though some isolation systems work purely in software.

2.4.1 Processes

Processes are the most common isolation mechanism in current systems, and traditional OSs provide them as the only isolation container for user code. Since processes are completely isolated from each other, they must use the OS as a mediator to communicate with each other through IPC primitives. Users thus isolate functional components by mapping them onto multiple processes, some of which serve as frontends and others as service processes. For example, the web server process in Figure 2.2 (*Web*) acts as a frontend listening to incoming network connections, and the database process (*DB*) services requests issued by *Web*. On mapping these into the underlying architecture, the OS uses a separate page table for each process, and the privileged IPC primitives are invoked through system calls that transition into the OS kernel privileged level. Being the most common mechanisms, processes, page tables and privilege levels have co-evolved along time to expose users to an illusion of running code on a machine for their exclusive use. Privilege levels (through system calls) provide a software interface that extends the hardware instruction interface to more complex (protected) abstractions; each process acts as a virtual CPU; and multiple processes (isolated through page tables) interact through IPC as if they were networked machines.

Other forms of software isolation exist, but all common OS organization models rely on the use of processes. Therefore, this thesis provides a more detailed analysis of the limitations attained by process-based isolation in Chapter 3.

2.4.2 Address Spaces

Page tables and privilege levels are the basic hardware protection mechanisms to build process isolation in current OSs, but other applications and mechanisms have also been proposed. *Nooks* [112] and *Fides* [110] take the kernel and user process domains (respectively) and further subdivide them to offer additional component isolation. Isolation of each subdomain is enforced through separate page tables [3].

Alternatively, *small spaces* [77] uses the segment registers found in older x86 to isolate multiple processes inside the same page table. Instead of using segments as the only isolation mechanism, they are used as an acceleration method for processes that communicate frequently.

Limitations

- Nooks and Fides are bound to the limitations of using address spaces; namely page table switch and management overheads, as well as granularity.

- Small spaces is limited by the number of segments available in the system. Domains must occupy non-overlapping memory ranges, which coupled with the 32-bit address space in x86 makes it not suitable for general use.

[3] In addition, Fides leverages machine virtualization to provide a much smaller TCB to enforce its policies.

2.4.3 Machine Virtualization

Privileged resources are typically out of the reach of user processes because the integrity of the OS kernel depends on this precise condition. In such cases, user processes must use system calls to request modifications of such privileged resources under the scrutiny of the OS kernel policies. Machine virtualization has proven an excellent target for library OSs like Drawbridge [92], OSv [66] and Dune [18, 19]. In these systems, the hardware enforces isolation across virtual machines, but each virtual machine can directly manage the privileged resources typically reserved to the OS kernel.

Limitations

- Isolation is not necessarily enforced between the components that are running inside a virtual machine.
- Such approaches are still subject to the overheads incurred by domain switches in virtualized machine environments (see Section 2.2.3).

2.4.4 Software Fault Isolation

Software Fault Isolation (SFI) foregoes the protection mechanisms provided by hardware and instead enforces isolation purely in software. SPIN [23], Java [54], BGI [30] and Mirage [82] are clear examples of SFI. Applications run in a software virtual machine that ensures data access safety. This same virtual machine can also be used to provide component isolation by programmatically applying restrictions on which components can interact with each other. Interaction with system services or other isolated user components can be as cheap as a simple function call, since there is no longer a need to switch between page tables or privilege levels. Similarly, Singularity [60] relies on a trusted toolchain that generates signed binaries that are ensured to adhere to memory safety and the system interfaces.

Native Client [104, 127] provides an alternative approach where the program loader can statically verify if a component is well-behaved, and applies dynamic checks otherwise. This goal is achieved by generating binaries that use a subset of the available instruction sequences, such that this subset can be statically verified.

Interestingly, the first incarnation of Native Client [127] provides a hybrid approach with hardware segmentation; whenever segmentation can ensure memory access safety, software checks are ellided. XFI [45] and Vx32 [48] provide similar tradeoffs (using paging and segmentation, respectively), and use dynamic binary translation to allow execution of binaries that cannot be statically verified.

Limitations

- Requires non-trivial additions to the TCB (e.g., trusted toolchains, static verifiers or binary translators), substantially increasing the attack surface.
- Relying on a trusted software virtual machine or toolchain makes it harder for third parties to develop and distribute their software.
- Perfectly safe computational code can see overheads associated to the base costs of the SFI infrastructure.

Chapter 3

The Interplay Between Isolation and System Design

The OS must support the partitioning of a system into isolated functional components in order to provide security and reliability, as well as fault detection to prevent fault propagation. Security is built on:

Resource isolation: i.e., which hardware and software resources can be accessed, like memory addresses or files. This is implemented on top of the hardware protection mechanisms.

State isolation: i.e., not allowing domains to interfere with the state of other domains, like modifying their registers. This is typically implemented in software; e.g., the OS kernel code that implements IPC saves and restores the state of the communicating processes.

Argument immutability: i.e., whether a domain should not modify the arguments it sent while they are being processed (e.g., between a server validates and uses them). This is implemented purely in software, since it usually requires copying data in a form that is specific to the system at hand; e.g., a `write` system call copies data across both ends of a pipe.

Fault detection requires resource isolation and **fault notification**, since the OS needs a way to notify its users of a fault in case they want to handle it. Reliability builds on fault detection and state isolation, but also on application-specific knowledge to survive faults [59]. Given the requirement of application-specific knowledge, reliability is beyond the scope of this thesis, even though the low-level OS mechanisms that it requires are addressed.

Current systems use privilege levels and page tables for resource isolation, and exceptions for fault notification. OSs thus build their abstractions on top of them. Ideally, OS and hardware primitives should naturally integrate into common programming paradigms and not affect performance. Unfortunately, existing OSs and hardware have co-evolved around models that unnecessarily increase code complexity and limit the ability to efficiently partition code into domains.

3.1 The Inadequacies of Process-Based Isolation

Processes are deeply embedded into the abstractions of current OSs, and serve multiple roles:

Resource container: The OS uses processes as a container for both hardware (i.e., memory allocations) and software resources (e.g., threads and open file descriptors). A process thus defines a subject for accounting and managing the life cycle of resources. When a process is destroyed, the OS also destroys the resources associated with it.

The EROS software capability system [105] provides a slight deviation from this model. The *space bank* abstraction is used to hierarchically sub-partition storage. When a parent process creates a child process, the parent provides a sub-partition of its space bank from where the OS will allocate memory for that child. A space bank can also be destroyed, giving users control of storage for an entire subsystem that could contain multiple processes. Therefore, a process in EROS still identifies its own resource container, but resource reclamation can be managed independently (although reclaiming storage will also destroy the processes it contains).

Persistence: Processes also serve as units of computation state persistence. All resources in a process are teared down upon its destruction, obviously including its state. Therefore, separate processes must be used to ensure state survives the coming and going of other processes. A typical example would be a database server process, which for performance reasons caches some structures in memory instead of performing all its operations on the disk (which is persistent by definition). If a database client process is destroyed, the state cached in the database server process is persisted across requests from multiple client processes. Without a separate database server process, maintaining consistency of the data cached in memory would generally be impossible without performing disk operations on every request.

An alternative exists to eliminate the persistence role from the process abstraction; nonetheless it deviates from existing OS designs. A "persistent library" could be implemented by ensuring it is always loaded at the same addresses across processes, including a persistent copy of its dynamic allocations. The addresses used by such library must be the same across all processes because otherwise the values of pointers stored in memory would be inconsistent across processes that use that same library "instance". Nonetheless, such design is beyond the scope of this thesis.

Isolation unit: By instantiating each process on a separate page table, the OS provides memory isolation across them. In addition, each process has a separate set of high-level software resource descriptors (e.g., the file descriptor table), which are implemented in and only accessible by the OS domain. Therefore, processes serve as an isolation unit for all types of resources, both software and hardware.

As this chapter will show, using processes as an isolation unit degrades performance and programmability, even though processes are still necessary to provide its container and persistence roles. Some of the trade-offs involved in isolation have been well studied in the past, such as the overhead of context switches [73, 116] and the performance of OS-mediated IPC [78]. Figure 3.1 shows the overheads of performing a synchronous call to a function on another domain (the callee) that receives a single buffer from its caller domain as an argument. The following primitives are analyzed, which exist in current OSs:

***Syscall*:** A user application invokes a Linux system call using the `syscall` function provided by `libc` (implemented with the `syscall` instruction in x86–64), much like the example shown in Figure 2.1a. The "*Syscall*" experiment simply performs the system call. In the "*Syscall +copy*" variant, the kernel uses Linux's `copy_from_user` function to copy the user-level buffer (caller argument) into the kernel memory (where the callee resides).

***L4 inline*:** The caller and callee are two separate processes that use the IPC primitives in L4 Fiasco.OC to communicate synchronously. The argument buffer is transferred as inlined data on the IPC message using L4's "message registers" through functions `l4_ipc_call` and `l4_ipc_reply_and_wait`.

***L4 remap*:** The caller and callee are two separate processes that use the IPC primitives in L4 Fiasco.OC to communicate synchronously. The argument buffer is passed "by reference" across processes using a software capability to the range of pages that contain it. The pages are mapped by the callee during its operation and are unmapped before returning, much like the example shown in Figure 2.1b. The benchmark is implemented using L4's runtime library to pass a "dataspace" through an "iostream" `call` operation. The callee receives requests through an "iostream" `reply_and_wait` operation, and performs an "attach" and "detach" operation of the "dataspace" before and after executing its workload, respectively.

***Semaphore*:** The caller and callee are two separate processes that use raw IPC primitives to communicate synchronously, implemented on top of POSIX semaphores (the `futex` system call in Linux). The semaphores are used to synchronize the cross-process request, but the argument buffer is passed "by reference" by allocating it in memory pages that are pre-shared among the two processes; therefore, no data copy is necessary. The "*Semaphore (=CPU)*" and "*Semaphore (≠CPU)*" variants identify whether the communicating processes are pinned to the same or different CPUs, respectively. The same CPU variants are also applied to the following experiments.

***Pipe*:** The caller and callee are two separate processes that use raw IPC primitives to communicate synchronously, implemented on top of POSIX pipes. A first pipe is used to signal the call and copy the argument buffer contents between processes, while a second pipe is used to signal the return.

***RPC*:** The caller and callee are two separate processes that use a Remote Procedure Call (RPC) to provide a synchronous function call interface for communicating. The RPC implementation internally wraps the use of UNIX sockets as the low-level IPC primitive, as provided by *glibc*'s `rpcgen`. Like in "*Pipe*", the argument buffer is copied across processes.

The experiment results are normalized to a regular function call, and cross-domain calls are repeated in a loop to ensure the standard deviation is below 1% and timing overheads below 0.01% of the mean. Figure 3.1a shows how efficiency is affected by the amount of instructions executed by the callee, modelled as a sequence of register increments (i.e., a measure of how bound the benchmark is to IPC performance). Figure 3.1b quantifies how the amount of communicated data impacts performance (note the different logarithmic scales). Finally, Figure 3.1c provides a more detailed comparison of the results when transferring a 1-byte buffer argument; the parenthesized "*(+proc)*" annotation indicates that communication happens between different processes, while "*(+copy)*" indicates that copies are implicit in the interface. The first two (Figures 3.1a and 3.1b) do not show the results for "*Pipe*" to avoid cluttering them, since its results consistently fall between "*Semaphore*" and "*RPC*" (see Figure 3.1c). It is important to note that all these experiments were performed without frequency scaling to maximize their performance. Unless explicitly noted, results are referred to with their prefix name; i.e., "*Semaphore*" refers to all semaphore experiments (i.e., "*Semaphore* *").

Privilege level switches have non-negligible costs for short routines, even though they are fairly optimized in modern processors; compare "*Syscall*" with other mechanisms in Figure 3.1c, and with less than 1K instructions in Figure 3.1a. In comparison, isolation between processes yields orders of magnitude higher overheads (see "*Semaphore (=CPU)*"). This is because the OS has to manage costly resource isolation (page table switches), and state isolation (register state of processes) whenever multiple processes communicate.

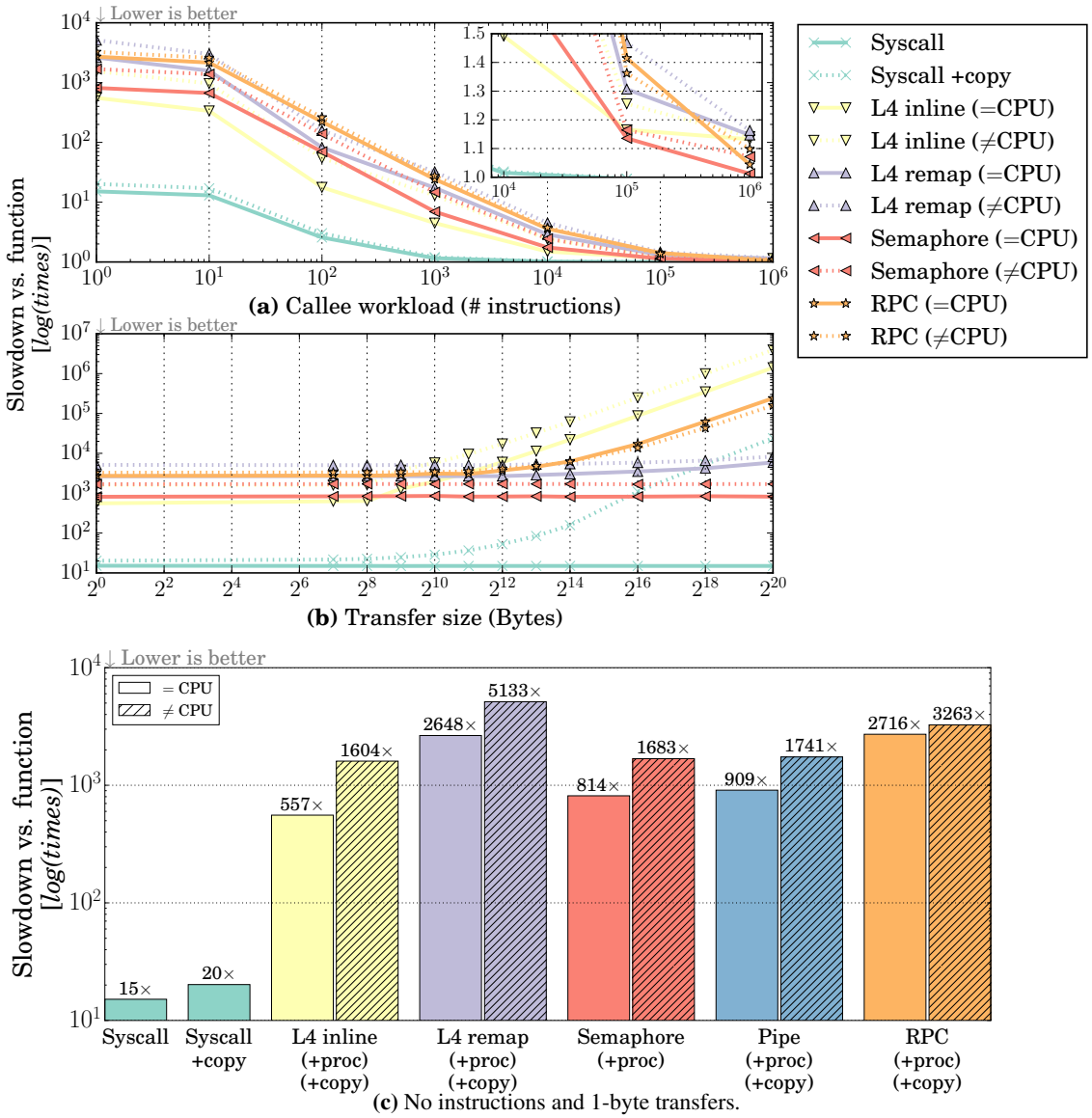
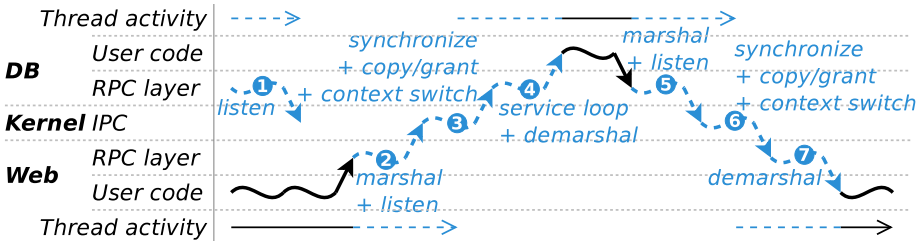
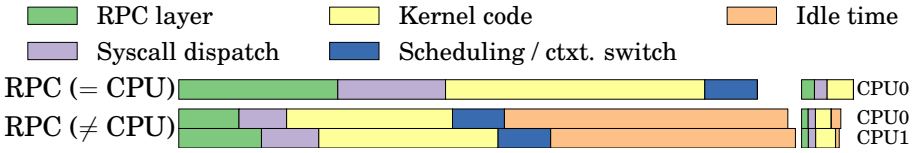


Figure 3.1: Comparison of the costs of different cross-domain communication primitives, normalized to a function call. Figure 3.1a depicts the cost of a synchronous caller/callee control transfer for different amounts of work performed by the callee (X axis). Figure 3.1b depicts the overhead of passing an argument of different sizes to the callee (the callee performs no work). Note the different logarithmic scale on the axes. Figure 3.1c depicts in more detail the results of Figure 3.1b with a 1-byte argument (leftmost points). Note that "*Pipe*" is not shown in Figures 3.1a and 3.1b to avoid cluttering them. Frequency scaling has been disabled, which otherwise increases cross-CPU overheads.



(a) Steps for a RPC between the web server and the database of Figure 2.2. Dashed arrows highlight operations imposed by the RPC/IPC layers, as compared to a regular function call.



(b) Modelled per-CPU overhead breakdown. Left side shows the base costs; right side shows the mean transfer costs for every 1 KB.

Figure 3.2: Anatomy of a RPC sequence.

Finally, "*L4 inline (=CPU)*" is a testament to the role that OS design can play in improving IPC performance; even though the underlying hardware is the same, it is more efficient than Linux semaphores, but still has to cater to the same needs of the process model.

These results suggest that IPC semantics and general kernel design play a major role in performance, as can be seen by the large performance variability shown by the different non-"*Syscall*" experiments. Also, comparing "*Syscall*" to the other mechanisms shows the importance that hardware mechanisms have in performance. In fact, hardware and OSs have co-evolved in a way that processes provide a loose form of machine virtualization, whereas threads provide a loose CPU virtualization abstraction. These concepts provided a very convenient way to evolve from the early single-program machine model to the later multi-programmed systems, but the networked model implied by processes is not necessarily the most efficient. Therefore, co-designing the hardware protection mechanisms and the OS isolation primitives looks as a promising approach to improving isolation performance and programmability. The following sections highlight other implications of the existing process and IPC designs, often overlooked.

3.2 Mismatch Between Procedure Call and IPC Semantics

IPC primitives provide semantics very different from those of the function calls that programming languages offer. Where a function call is synchronous and can pass multiple arguments with references to data, IPC primitives are typically asynchronous and can only pass a copy of a single buffer of data. This semantic mismatch impoverishes the programmability of the system, and is often hidden under a RPC layer that preserves function call semantics. Therefore, RPC regains the programmability lost to raw IPC primitives. But comparing "*RPC (=CPU)*" with other mechanisms in Figure 3.1 shows that bridging this semantic gap adds large overheads due to the added complexity of the underlying code.

Mechanism	***Slowdown (times)***
try/except	1.00
try/except + throw	681.79
setjmp	2.50
setjmp + longjmp	6.36

Table 3.1: Costs of common state isolation/recovery mechanisms normalized to a regular function call.

Figure 3.2a depicts the anatomy of an RPC; dashed arrows represent operations added to bridge the semantic mismatch. Figure 3.2b shows the sources of its overheads. The results show a time breakdown of the "*RPC*" experiments in Figure 3.1, using a modified Linux kernel to measure where time is spent in a very light-weight and accurate manner (see Section 8.2.4 for more details).

First, *RPC layer* in Figure 3.2b shows the performance costs of the glue code added in the user code. This glue must ***marshal*** and ***demarshal*** the arguments and return values over IPC (steps ❷, ❹, ❺, ❼). This is because most IPC primitives send a single buffer of consecutive memory, and even in a simple RPC benchmark these overheads are far from negligible. In contrast, these operations are not needed in regular function calls, which can use pointers to memory as arguments. The RPC glue also requires a ***service loop*** in the callee to dispatch requests to routines (❹), and it must invoke low-level routines to ***listen*** on the communication channel and wait for a request/response (❶, ❷, ❺). This last overhead corresponds to the time difference in *RPC layer* between the caller (*CPU0*) and the callee (*CPU1*) for "*RPC (≠CPU)*" in Figure 3.2b.

At the OS kernel layer, IPC provides ***resource and state isolation*** (❶, ❸, ❻). This requires tracking the currently executing process and saving/restoring its register state when switched. Conversely, the compiler can optimize register state management in regular function calls by not saving unused registers, since it knows what registers are used by each function. For example, Table 3.1 shows a comparison of two common state isolation and recovery mechanisms. The implementation typically used by the OS kernel during an IPC call saves and restores all registers due to process switching, equivalent to the *setjmp* and *setjmp + longjmp* experiments. A regular function call does not require any of these operations, since the compiler knows the registers used by each side of the function call. Nevertheless, if the callee severely tampers with the computation's state (e.g., thrashes the stack), the caller function may not be able to properly resume execution; in comparison, the strict isolation provided by processes ensures the caller can resume execution no matter what happens in the callee. Interestingly, the C++ language provides very efficient state isolation across functions inside the same process in the case of non-malicious errors (e.g., does not handle cases of a domain randomly thrashing its state). The C++ compiler is able to hide these costs for the common case of non-faulting exceptions (*try/except*) by generating ancillary information structures that describe how to recover the state of the computation from well-known safe values. When an exception is actually raised (*try/except + throw*) the costs are much higher because the runtime must reconstruct the state at the call site from previous values. The important bit here is that cross-domain communication could be implemented using similar techniques to safeguard against errors, instead of saving and restoring all the state across domains. Since errors are the least common case, it should not be a problem if fault processing code is penalized.

IPC also ***transfers data*** between buffers of the communicating processes (❸, ❻). Interestingly, Figure 3.1b shows that none of the primitives is a best fit for all sizes. "*L4 inline*" incurs the least overhead for small transfers. This comes at the cost of one L4 IPC for every 512 B (a limitation in the implementation necessary to make it this fast), leading "*RPC*" to perform better for ≥ 2 KB. Copies are necessary

in these two mechanisms because they provide argument immutability, leading to the copy semantics they implement. Larger buffers can be better amortized by sharing memory pages on demand like in "*L4 remap*" for ≥ 4 KB. "*Semaphore*" shows a faster alternative for large transfers where pages are pre-shared and IPC is just used to synchronize processes. Unfortunately, this strategy is limited to page granularity, requires agreeing on buffer sizes beforehand, and data must usually be copied back and forth from the shared buffer during cross-domain transfers. Evidently, copies are not required in regular function calls. Using copies in function calls is left to programmer discretion, who knows when they are necessary to maintain the program semantics.

IPC also imposes additional hardware operations when compared to regular function calls. First, cross-process copies generate cache and TLB capacity misses. Interestingly, "*RPC (≠CPU)*" in Figure 3.1b shows better results starting at ≥ 16 KB because the L1 data cache (32 KB) is overflown with "*RPC (=CPU)*". Nevertheless, this might not be beneficial on a busy system due to the added cache coherence traffic. Second, sharing pages on-demand across multi-threaded processes can be more harmful than large copies due to the costs of additional TLB shootdowns and the IPIs they require [121]. Third, the added IPC and RPC code pollutes caches and TLBs and contaminates the branch predictor. In fact, the aggregate *RPC layer* time in Figure 3.2b is slightly lower for "*RPC (≠CPU)*" due to an order of magnitude less cache and TLB misses (since each process executes a different subset of the code).

3.3 False Concurrency

Ideally, one would use threads only when concurrency is desired by the programmer. However, threads cannot typically cross process boundaries. Therefore, different threads must be used on each process, and threads must explicitly synchronize to communicate across processes, incurring performance and programmability overheads. As a result, a thread that initiates an inter-process transaction, which we refer to as a ***primary thread***, must transfer control to a ***service thread*** that executes the transaction in the server process. Service threads only serve as remote process proxies for primary threads, imposing concurrency across processes even when the application has synchronous semantics. For example, whenever a web server thread (a primary thread) interacts with a database thread (a service thread), they must coordinate control transfers back and forth. The same applies to microkernels whenever a process issues a synchronous request to a service process (e.g., the file system service). Synchronization triggers costly context switch operations (register state and page table), which are shown in Figure 3.2b as *Scheduling / ctxt. switch*. Ultimately, this translates into the performance difference between "*Syscall*" and "*Semaphore (=CPU)*" in Figure 3.1. Waking up a process on a different core also incurs a costly IPI, which can be seen as higher costs for the "* *(≠CPU)*" experiment variants in Figures 3.1 and 3.2b. As noted before, the experiments were performed without frequency scaling. When enabled, frequency scaling increases cross-CPU communication delays and further accentuates the IPI overheads.

This false concurrency also isolated processes in terms of their threading model (i.e.,, each process must separately manage its set of threads). Each process must dispatch requests to its service threads, which are either created on-demand or kept in a thread pool. This either incurs substantial run-time overheads or increases programming complexity, respectively. For example, the web server and database processes from Figure 2.2 each allocates and manages its own set of threads. In addition, the database tends to many clients, and must provision for a high load even though load is dictated by the number of client primary threads.

Summarizing, service threads and their context switches are artifacts imposed by current OS designs whose process model ties each thread to a process. This forces transactions to span multiple threads even when no concurrency is possible given the program's semantics. Therefore, allowing primary threads to

securely cross process boundaries would improve performance and programmability.

3.4 A Case for Configurable Isolation Policies

Processes enforce full isolation even when that is not needed. For example, the OS kernel assumes processes' state must be isolated from each other and memory copies are used in IPC primitives to communicate. Such level of isolation is often not necessary. Therefore, allowing programmers to express their isolation policy requirements can minimize programming complexity and increase performance without violating the isolation properties required by the programmer.

This is because software components often have *asymmetric* relationships, while processes enforce a single *symmetric* isolation policy. For example, application extensions (e.g., a web server PHP interpreter or a web browser plugin) can be fully sandboxed while the master application is allowed to directly interact with the extension's memory without going through IPC. In addition, state isolation can be eliminated when faults are merely forwarded. For example, if a database crashes while a PHP interpreter performs a query to it, the error is reflected to the PHP interpreter. If the interpreter does not contain code to explicitly recover from such errors, it is left in an inconsistent state, rendering state isolation useless. In this case, it would be thus more efficient to eliminate state isolation from the PHP interpreter, and instead forward faults to the caller web server, who will centralize the complexity of handling faults.

The need for argument immutability (i.e., copies) is also dependant on program semantics; more specifically, it is not necessary when security is not affected. For example, a disk driver does not need a private copy of its input buffers; a requester could modify them while the write is in process without affecting the driver's security, just like in an asynchronous write it is safe to modify the buffer contents between requesting the write and synchronizing with its end. Conversely, a database needs a private copy of its requests, since it must parse and verify them before actually executing them. Argument immutability is thus only necessary when time-of-check-to-time-of-use (TOCTTOU) atomicity is required, and only the programmer knows the application semantics well enough to know when this is needed.

Chapter 4

Efficient and Composable Isolation Primitives

This chapter provides an overview of the hardware and OS co-design proposed in this thesis. The rationale behind this design is driven by the four goals set by Chapter 1:

- Offer efficient component isolation.
- Maintain existing programmability through a synchronous function call interface for cross-domain requests.
- Provide an expressive and composable set of isolation primitives to ease the implementation of disparate isolation policies with minimal performance overheads.
- Ease gradual adoption on existing systems.
- Be general enough to be applied across the whole software stack.

Efficiency, programmability and gradual adoption are obvious goals for any proposal that targets existing systems. Unmodified components should still run in the system, and code oblivious to isolation should maintain its original performance. The system should also provide a path for software components to gradually adopt the proposed primitives. Since paging is still useful for multiplexing physical memory, multiple isolation domains should coexist in the same virtual address space, so that protection is overlaid on top of translation. Commonly used primitives should be cheap enough to pass inadvertently in the performance profile, and their efficiency should scale with the number of cores and threads in the system. From the experiences of past work, this last point is particularly difficult to achieve for access grants and revocations. Domain switches should follow the same model provided by regular function calls to eliminate unnecessary programmability overheads: arguments can be passed by reference and (cross-domain) calls are synchronous.

The generality and composability of primitives plays a prominent role in the design of this thesis. Primitives should be able to enforce all types of isolation required by software in an efficient way; from user/kernel separation, to application extension and inter-process isolation. Furthermore, primitives should avoid semantic overload in favour of composability to foster the exploration of novel isolation organizations. Current systems have prohibitive overheads and, even worse, using them in ways that deviate from their intended

model imposes even larger overheads. RPC is a clear example of this; side-stepping the intended model of processes requires additional coding complexity that leads to performance overheads. Ultimately, these overheads deter the exploration of new designs. In this same vein, asynchronicity and argument immutability should be left to programmer discretion and not be imposed by the system primitives.

4.1 Security Model

Given that this thesis seeks to expose users to mechanisms that can be efficiently composed to build various isolation policies, a security model cannot be strictly defined. Instead, the proposed primitives serve as isolation policy building blocks. Therefore, a list of properties that the system must be able to enforce is provided instead:

Memory and privilege protection : This is the most important property that the hardware itself must enforce at all times. Without it, it is not possible to implement domain isolation.

Entry point identification : Domains must be able to identify which addresses are valid entry points for other domains. The system must enforce that cross-domain calls only go through identified entry points when requested. Failing to do so could be used by a malicious caller to force the execution of unintended paths on another domain.

Return address validation : A callee domain must be able to ensure that its return address follows the intended semantics of a function call. Failing to do so could be used by a malicious caller to misdirect the callee into unintended code when it returns.

Computation state integrity : Callers must be able to enforce the integrity of the state of their computation around a cross-domain call (i.e., forbid unintended writes). Memory protection is handled in the first point above, so computation state comprises the thread's stack and register values. Failing to do so could be used by a malicious domain to manipulate the state of another domain.

Computation state confidentiality : Callers and callees must be able to enforce the confidentiality of their computation state for anything that is not an intended call argument or result, respectively (i.e., forbid unintended reads). Failing to do so could be used by a malicious domain to gain access to leaked sensitive information like private encryption keys.

4.1.1 Asymmetric Isolation Policies

These building blocks can be used to tackle security and reliability against errors and malicious actions. Furthermore, providing a system with policy building blocks (instead of imposing a single policy on all cases), allows the construction of isolation policies that more accurately follow the programmer requirements while avoiding the overheads of unnecessary isolation properties. This level of control allows the implementation of *asymmetric* isolation policies; that is, policies that are different on each side of a cross-domain call.

Using asymmetric policies implies that isolation guarantees can be traded off for performance. The safest option is to enable all isolation properties by default, but let the programmer to select when, and which, to disable in specific circumstances. This is specially important when invoking short routines across domains. Short calls to other domains are typically avoided in current systems due to the overheads imposed by existing isolation mechanisms. If one such function is used frequently, the programmer can stop to think if the other domain can be trusted and select which isolation properties are unnecessary, therefore improving performance.

Such decisions involve two observations. First, whether the code on the other domain can be trusted to follow the system's conventions; i.e., it is not maliciously trying to exploit weaknesses on the domains it is interacting with. Second, whether the code on the other domain can be assumed to be correct; i.e., it does not contain a bug that can lead to a break of the assumptions of the domains it is interacting with.

For example, a web server could split its encryption code into a separate domain, maintaining encryption keys outside the reach of the rest of the application in case it is compromised. Since both domains are part of the same application, they can be trusted to not be malicious to each other. It does not make sense to assume the code malicious, since the programmer would then be defeating herself. In this example, the web server can safely decide to forego integrity and confidentiality. Likewise, the isolated encryption code can forego integrity, but maintain confidentiality in case the web server is compromised (to avoid partially leaking the encryption key; e.g., through stale register values [88]). Similarly, the web server can call into an isolated plugin while maintaining both integrity and confidentiality. Since the plugin is exposed to external inputs, it could be subject to vulnerability exploitation attacks. At the same time, the plugin does not need to maintain integrity nor confidentiality with the web server; in this case, the web server can be fully trusted by the plugin, since doing otherwise would defeat the purpose of a plugin architecture.

OSs pose another common example where policy asymmetry can be used. By design, the OS kernel is fully trusted by all applications; therefore, user applications can forego integrity and confidentiality when interacting with the OS kernel.

Importantly, mis-configuring the policies on one domain can expose it to errors or malicious attacks, but will never directly expose other domains that interact with it. Of course, this conforms the system into a chain of trust; if a domain A assumed another domain B safe, but B is exposed through a mis-configuration, A's assumptions are not valid. Nonetheless, this chain of trust already exists in current systems; for example, a bug in a binary serializer application can break another application that consumes that data, or a bug in a device driver can break applications that use it.

4.2 System Design Overview

The most ambitious goal of this thesis is to substitute existing RPC primitives with highly efficient cross-process calls, without percolating the process concept nor its semantics into the architecture. That is, allow the coexistence of multiple user-level isolation domains in the pure micro-kernel style, but providing a simple synchronous function call interface and without mediating their communication through the OS kernel.

Existing capability architectures have the highest potential to achieve this goal with high performance, but suffer from numerous shortcomings and many challenges remain open. First, many capability architecture proposals concentrate on isolating components inside a single application. Extending their concepts across processes requires a careful hardware design and significant changes on the OS. Second, many capability architectures hardwire semantics and concepts of system structure, precluding the optimization of isolation properties when they are not needed (e.g., by implementing the process concept in hardware like in the Plessey System 250 [72]). Third, capability architectures are typically hard to integrate in existing codebases without large changes on the code and/or the compiler; while these can provide the best mix of isolation granularity and performance, there must exist intermediate points for gradual adoption.

This section provides an overview of the design of the different parts of the proposed design, and how they relate to each other, as illustrated by Figure 4.1. Some simple compiler support is used to allow the programmer to describe her isolation domains and desired isolation policies. In turn, the DomOS OS provides the interfaces and necessary logic to enforce the isolation of the programmer-specified policies, which are then mapped into the CODOMs architecture. As a result, programmers can perform regular function

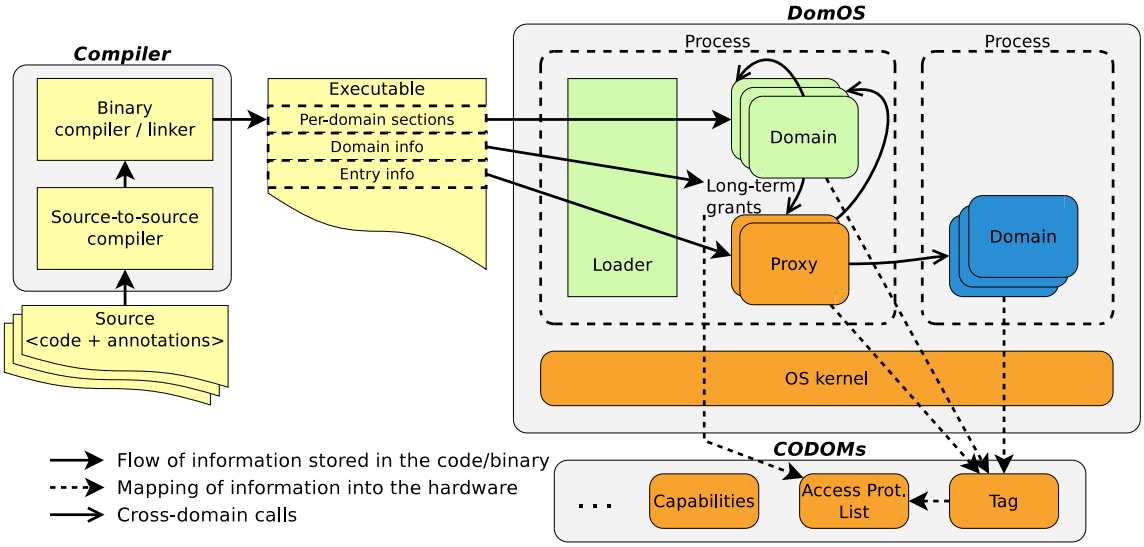


Figure 4.1: Overview of the relationship between the different parts of the proposed system. Programmers can perform regular function calls that cross isolation domains, and each domain can equally be inside the same or a different process.

calls that cross domain boundaries, and each domain can equally be inside the same or a different process. An in-depth description of the design, implementation and the finer points behind each of the parts of the proposed system is provided in Chapters 5 to 7.

4.2.1 CODOMs

The CODOMs architecture, partly shown at the bottom of Figure 4.1, provides the low-level mechanisms to enforce memory and privilege protection. Its design is inspired by key-based memory protection and capability architectures, taking the best of both worlds. On one hand, *code-centric* protection provides efficient mechanisms to define the boundaries of isolation domains and their long-term grants to other domains at page granularity, with a special emphasis on very efficient cross-domain function calls. On the other hand *capabilities* allow transient sharing of information across domains at arbitrary granularities, with a special emphasis on efficient revocation.

CODOMs provides *code-centric* isolation by deriving the active protection domain from the instruction pointer of the executing instruction; that is, the instruction pointer acts as an implicit capability. This seemingly simple property allows implementing cross-domain function calls at virtually no latency. Each domain is associated with a tag, and the page table is extended with a per-page *tag* to assign each page to a domain. An additional per-page bit indicates if execution of privileged instructions is allowed in code stored on that page. To describe which other domains one can access, every tag (or domain) T is associated with an *Access Protection List (APL)*: a list of tags in the same address space that code pages in domain T can access, along with the granted access permissions. For example, the APL for the green domain in Figure 4.1 could be configured to allow it to call into functions on the orange ("proxy") domain, but not the blue domain that exists on another process. Figure 4.2 shows a comparison of a cross-domain call in CODOMs, key-based memory protection and traditional capability architectures. The main advantages of CODOMs' code-centric

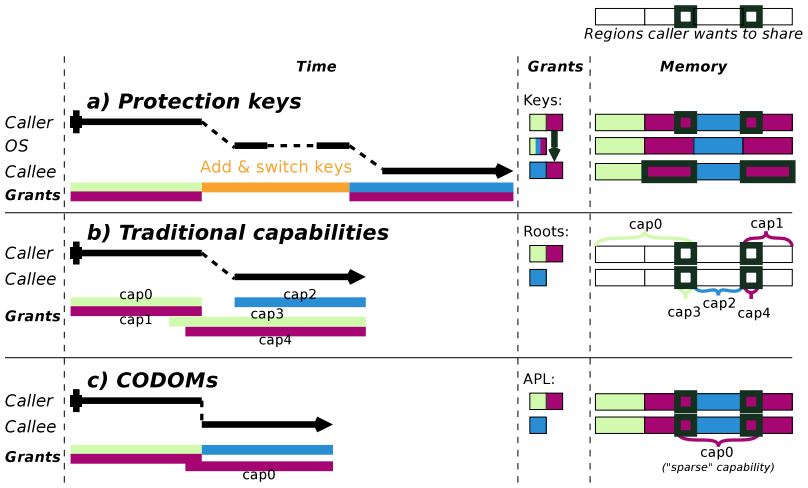


Figure 4.2: Comparison of CODOMs with other cross-domain request mechanisms shown in Figure 2.1. It compares CODOMs (Figure 4.2c) against protection keys (Figures 2.1d and 4.2a) and traditional capabilities (Figures 2.1e and 4.2b).

protection are:

Performance: Changing the current domain is implicit in control flow operations. Therefore, domain switches do not require OS kernel supervision (unlike key-based protection, as shown by the OS line in Figure 4.2a), and careful hardware design makes them have virtually no overhead even in out-of-order pipelines (unlike the overheads of changing the root capabilities in Figure 4.2b, as described in Section 2.2.4).

Transparency: Code-centric protection can be used transparently by existing code since a simple function call can transition between domains. This is unlike key-based memory protection and traditional capability architectures, where management of cross-domain calls must be explicit (e.g., using a specialized instruction, or having to explicitly manage grants at every cross-domain call boundary).

Sparsity: Domains can be "sparse", unlike in traditional capability architectures; this can be seen by the pair of capabilities *cap0* and *cap1* in Figure 4.2b which are both necessary to define the caller domain (see Section 2.2.4). That is, like in key-based memory protection, domains can be composed of non-consecutive memory pages, as can be seen in the *Grants* column of Figures 4.2a and 4.2c.

Domain count: The domain space is virtually infinite. This is unlike key-based memory protection (Figure 4.2a, described in Section 2.2.2), where per-page tags must be changed to multiplex multiple logic domains into the keys provided by the hardware (ultimately leading to costly TLB shootdown overheads).

Privilege protection: Code-centric protection also encodes access to privileged resources, unlike key-based memory protection and traditional capability architectures. Therefore, regular function calls can be used to provide zero-latency system calls.

Nonetheless, the APL is a privileged structure supervised by the OS kernel and only works at page granularity, making it more suitable to express long-term properties of the system. Therefore, CODOMs

provides *hardware capabilities* to allow efficient passing of data references as function arguments across domains. They do not require OS supervision and can express arbitrary granularities, making them suitable for short-term, or transient, grants. The main advantages of CODOMs' capabilities compared to other more traditional implementations are:

Revocation: CODOMs provides two types of capabilities to support immediate and selective revocation (explained in Section 2.2.4) in a very efficient way:

Synchronous capabilities are used to temporarily grant access rights for the duration of a function call, and are implicitly revoked after the function returns at no cost. Note that this allows further delegating a grant across nested function calls. Therefore, a special emphasis is put on zero-overhead revocation for the case of sharing capabilities in synchronous, cross-domain call/return scenarios which are, by far, the most common case.

Asynchronous capabilities are used to grant access rights that outlive the callee. They are useful during asynchronous data transfers between domains (e.g., an asynchronous disk read), or when two threads exchange capabilities through the memory. In this case, CODOMs provides an efficient implementation for immediate and selective revocation using a per-capability pointer to a "revocation counter".

Support for efficient immediate and selective revocation is specially important in the scenario where capabilities can be shared across processes. Even when immediate and selective revocation is not exposed to the programmer, capabilities must be revoked when the virtual memory used by a process that has been killed is eventually reused to allocate a new process. One could argue in favor of using a garbage collection approach, but this is simply not feasible at the whole-system level; i.e., across processes. Implementing garbage collection across processes requires involving the TCB (e.g., the OS kernel) on every single call to `malloc` and `free` to track allocations and coordinate them with uses of capabilities across all processes in the system. One could provide a more conservative garbage collection approach that treats processes as a single unit of memory, reusing virtual memory used by a process only when there is no alive process that has ever had direct or indirect call access to that process; nonetheless, this would very easily degrade to an information flow graph where all nodes (processes) keep all others alive (i.e., un-reusable).

Sparsity: Capabilities in CODOMs are "sparse", since they are derived from the code-centric protection definitions. This is shown by the single capability *cap0* required by CODOMs in Figure 4.2c, instead of the many capabilities required by traditional implementations (*cap3* and *cap4* in Figure 4.2b, as explained in Section 2.2.4).

This feature opens the door to alternative designs more efficient than the typical approach of using one capability for each pointer. For example, one can allocate an entire dynamic data structure on a separate domain, and pass a single capability to that domain. Since CODOMs' capabilities are "sparse", the allocating domain does not need to worry about whether allocations are in consecutive addresses, something that cannot be ensured in the general case. Therefore, a callee can use a single capability to grant access all elements of a dynamic data structure (e.g., a graph), instead of the more conventional (and expensive) approach of using capabilities as a pointers to grant access to each of the elements in the data structure. Furthermore, using this approach it is very simple to provide a read-only capability to the entire data structure. Doing so in a traditional capability architecture either requires having two data structures (one with read-only capabilities and one with read-write capabilities) or requires changing the hardware to transitively apply the read-only limitation (e.g.,

transform read-write capabilities into read-only ones when they are fetched as a result of chasing the pointers/capabilities of the data structure, similar to how EROS transitively enforces the *weak* property in its software capabilities [105]).

Unforgeability and integrity: Capability unforgeability and integrity is ensured without resorting to expensive memory tagging (described in Section 2.2.4). Instead, a per-page bit indicates if a page can be used to store capabilities, in which case unprivileged code cannot directly access its raw contents.

Transparency: To make gradual porting easier, capabilities in CODOMs can be used implicitly by code oblivious to them. In this case, CODOMs checks if a memory access is authorized by any of the available capabilities before raising a memory protection exception. For example, a capability-aware caller can pass arguments through capabilities to an unmodified, backwards-compatible callee.

4.2.2 Compiler Support

At the other side of the design lies some simple compiler support, shown at the left-hand side of Figure 4.1, that programmers can use to express how to take advantage of the underlying CODOMs architecture to isolate multiple components inside the same application. The compiler provides simple annotations that allow programmers to assign code and data into domains, identify domain entry points, and express domain isolation properties at the domain and entry point boundary. This information is embedded in the output binary, and later used by the program loader to setup the corresponding domains.

There is an important observation to make about the role of the compiler: many of the isolation properties described in Section 4.1 can be enforced by user level software. This is in contrast to existing systems that rely on privileged code or hardware instructions with complex semantics to enforce a fixed set of isolation properties that many times are unnecessary. The compiler can generate caller and callee stubs that enforce the selected properties at user level, therefore reducing the amount of code and complexity in the TCB. Furthermore, compiler-generated code follows an Application Binary Interface (ABI) convention that defines how each register is used across a function call. Therefore, the compiler can co-optimize these stubs with knowledge from the rest of the application; e.g., there is no need to enforce the integrity of registers that are considered dead at the call point, something that depends on both the function signature (translates into code through the ABI) and the architectural state of the code at the call site.

4.2.3 DomOS

DomOS provides the necessary primitives to let software use the mechanisms provided by CODOMs. The net result is that processes can be composed of multiple isolation domains and, importantly, domains from different processes can issue simple function calls instead of going through expensive RPC primitives.

The DomOS functionality is spread through the application loader, the run-time and the OS kernel, all shown at the top-right corner of Figure 4.1. The application loader reads the information generated by the compiler to create and configure the domains inside an application. The runtime then resolves cross-domain calls lazily the first time they are used, following the same scheme used by dynamic libraries [71]. A helper runtime library also simplifies the management of cross-process entry points, in a similar way to existing RPC libraries.

The domain and entry point configurations expressed by the additional compiler/language support are mapped into the DomOS kernel which, in turn, maps these into CODOMs' tags and APLs. Some of the isolation properties of domains and entry points refer to privileged resources unavailable to user code; e.g., changing the identity of the current process during cross-process calls. In this case, DomOS generates

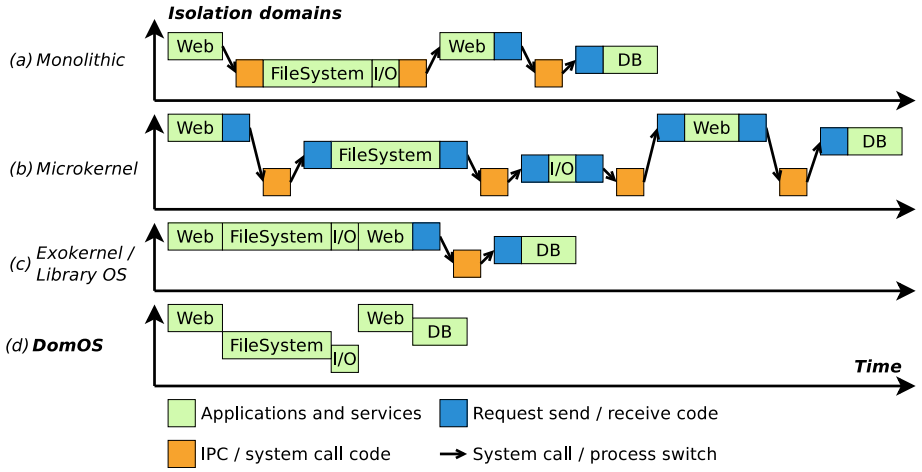


Figure 4.3: Comparison of the different system organizations in Figure 2.2 and DomOS. DomOS is able to keep the different components isolated from each other. At the same time, it does not add extraneous code to send and receive requests and does not need to go through the OS kernel on every request.

a thin trusted *proxy* routine at run-time that bridges the calls between domains and is specialized to the required isolation properties and entry point signature (orange *Proxy* domains in Figure 4.1). By producing a specialized trusted proxy at run-time, cross-domain calls do not need to go across sub-optimal generic functions implemented in the OS kernel. To achieve all these goals, DomOS also maps all processes into a global virtual address space using a single page table, and different CODOMs tags and APLs to keep processes isolated from each other (green and blue *Domain* boxes in Figure 4.1). Figure 4.3 revisits the different OS organizations models described in Section 2.3 and compares their performance with the model proposed in DomOS. Software components in DomOS can be isolated from each other, like in an idealized microkernel. Furthermore, cross-domain calls in DomOS *are* implemented as synchronous function calls at their lowest level. They do not need to go through the generic IPC primitives implemented by the OS kernel, no code is necessary in the caller or callee to bridge the semantics of function calls into those of the IPC primitives, and the system is not subject to the overheads of false concurrency (Chapter 3).

4.3 Isolation Scenarios

CODOMs and DomOS allow programmers to efficiently implement a variety of scenarios. For example:

Sandboxing: This is the typical case for an application plugin (e.g., Chrome isolates plugins by running them as separate processes), but from the point of view of the OS kernel one can also see processes as sandboxed domains. In this last case, CODOMs' efficient privilege isolation can be used to replace system call instructions with regular function calls. Thanks to CODOMs' code-centric isolation, the parent domain (the main application in the plugin case, or the OS kernel in the case of user-kernel separation) can easily have full access to the sandboxed domain without using capabilities. At the same time, the sandboxed domain cannot access its parent's resources, and implicit capability use allows running sandboxed code that is oblivious to isolation.

Privilege amplification: Is the exact opposite of sandboxing; e.g., a request from a plugin into the main application, or a request from a process to the OS kernel goes from a "less privileged" to a "more privileged" domain. In this case, the parent must export entry points to the sandbox. Nonetheless, the sandbox does not need to use capabilities to pass arguments to the parent, since the parent has direct access to the sandbox.

Mutual isolation: Is the typical scenario found in communicating processes or different subsystems and plugins inside a single application. None has direct access to the other, and thus communication in both ways must happen through entry points exported by each of the domains.

These scenarios describe isolation at the memory protection level. Nevertheless, other isolation properties (like register state or stack isolation) can be controlled separately, as described in Section 4.1, to allow implementing more efficient asymmetric isolation policies. Nonetheless, it is perfectly possible to instead use noan intermediate domain that provides a more generic (and thus less efficient) implementation that does not require annotating the applications. In fact, the IPC primitives provided by the OS kernel are examples of such generic implementations, albeit quite sub-optimal given the OS and hardware designs they support.

Chapter 5

Hardware Support

"You're very clever, young man, very clever", said the old lady. "But it's functions all the way down!" [1]

CODOMs provides the low level hardware mechanisms to enforce memory and privilege protection. The architecture is prototyped by adding and modifying some of the elements present in an x86–64 CPU, as shown in Figure 5.1. These elements provide two coupled mechanisms (overviewed in Chapter 4): *code-centric* protection defines the long-term properties and access grants of domains, while *capabilities* provide efficient short-term data access grants across domains. As a result, regular function calls can be used to switch across domains, and capabilities can be used to efficiently grant access to data passed as function arguments. The following sections describe the implementation and purpose of each of these mechanisms and the hardware elements that are involved.

5.1 Code-Centric Protection

Each domain in CODOMs is defined through an extension of the regular page table. In addition, each domain also identifies which other domains (and with which permissions) it can access. Therefore, the instruction pointer implicitly defines what is the current domain and, importantly, what other pages it can access. This is the basis for allowing regular function calls to transparently and efficiently cross domain and privilege boundaries.

5.1.1 Page Table Capabilities

Page table capabilities define the bounds of a domain (in terms of memory pages), as well as some of its properties. They are implemented as an extension of the regular page table through some of its unused bits. Their format is shown in Figure 5.2, where greyed-out areas are ignored by CODOMs:

Tag (64 bits): Identifies the domain to which a page table entry pertains. Tags are stored in the page physically contiguous to the one they describe, and occupy the same space of a page table entry (the 4 KB at

[1] https://en.wikipedia.org/wiki/Turtles_all_the_way_down

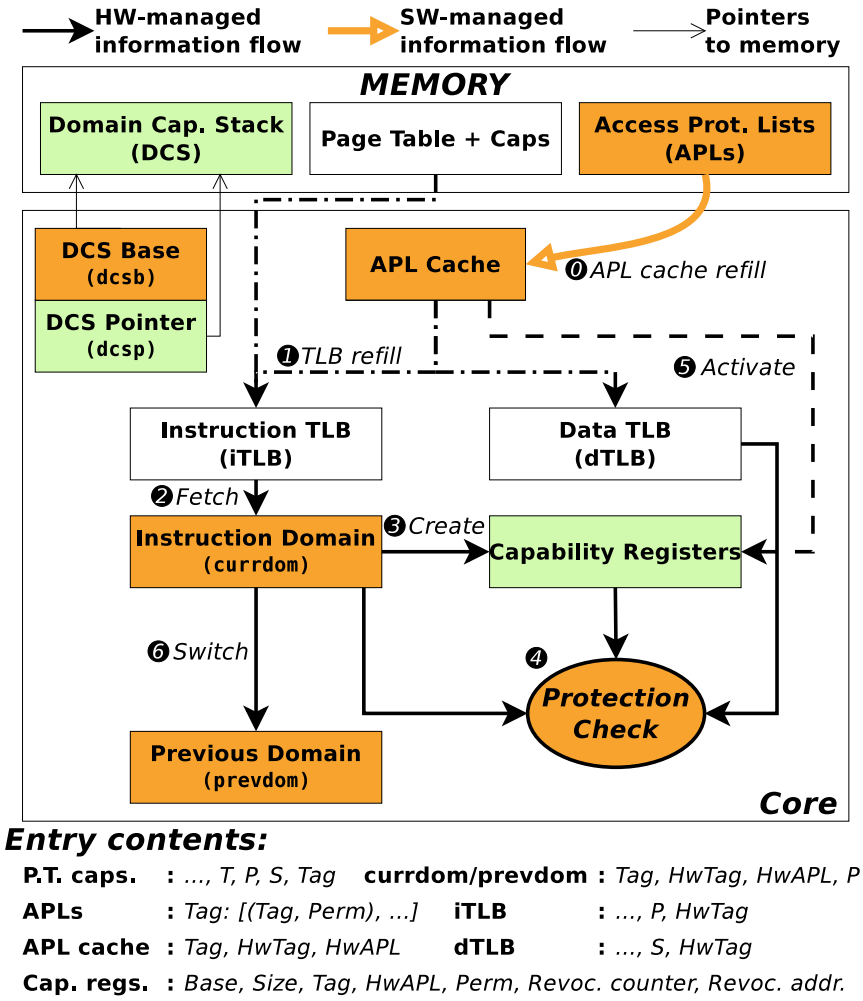


Figure 5.1: The CODOMs architecture. Added elements are colored, and extended elements are white. Light green elements are controlled by the application, while others are controlled by the TCB. Numbers are used in Section 5.4 to describe the different steps of how information flows through the architecture to perform access checks.

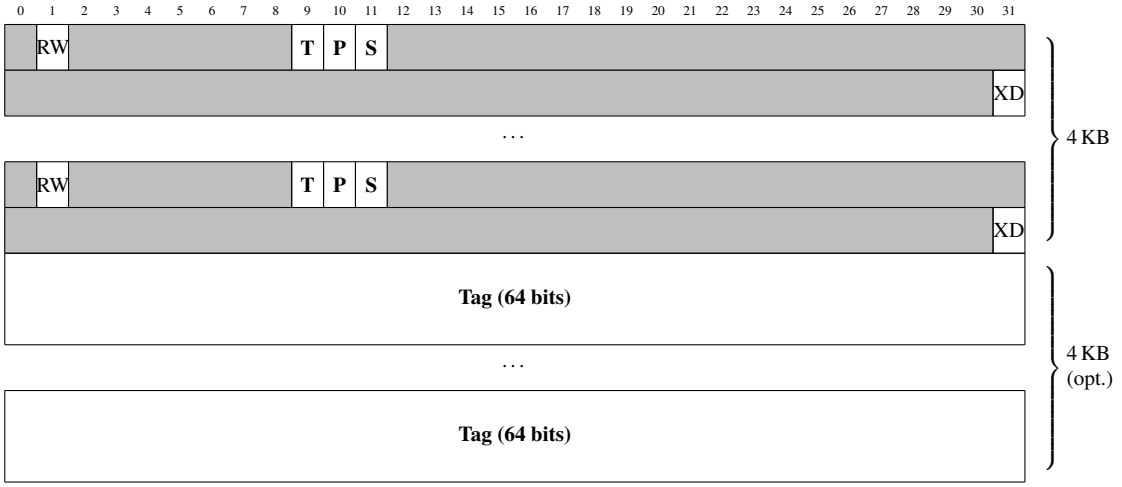


Figure 5.2: Location of added (bold face) and relevant fields in the page table structures. Greyed-out areas are ignored by the CODOMs extensions. The additional (physically contiguous) page containing tags is present only when the **T** bit is set in one of the page table entries.

the bottom of Figure 5.2). Keeping both segregated maintains backwards compatibility with the page table format.

Tag presence bit (T): Indicates if the page table entry has an associated *tag*. If the bit is set, it means that the associated tag must be used, which is located 4 KB forward on the physical address space.

Not setting this bit allows executing existing code without CODOMs. Since the bit exists at all page table levels, tags can be set for entire sub-trees. This minimizes space and management overheads for the page table.

Privileged capability bit (P): Indicates whether privileged operations are allowed in the code contained in the range of memory described by the page table entry.

Serves as an efficient replacement to privilege levels, ensuring regular code cannot execute privileged operations. Note that the existing *U/S* bit in x86 (for *user/supervisor*) could be repurposed instead of implementing *P* on an unused bit.

Capability storage bit (S): Indicates if the range described by a page table entry contains capabilities. Capability load and store instructions must be aligned to the capability size and must reference a capability storage page, otherwise an exception is raised. Regular load and store instructions must reference pages without the *S* bit, otherwise an exception is raised. Together, these rules ensure the integrity of capabilities stored in memory.

This bit provides a low complexity implementation that allows mixing code/data and capabilities on the same domain, as long as they are on different pages. This is unlike other systems, where one of the following is true:

- Capabilities use a separate address or indexing space.
- Capabilities contain a bit indicating whether the target segment contains data or capabilities.

- Memory is tagged at byte granularity to identify which bytes must be integrity-protected.

Nevertheless, data structures must be split between data and capabilities. Given the maturity of current compiler technology, it is safe to assume the compiler can be modified to automatically manage such structures in high-level languages. Furthermore, Section 4.2.1 describes a design where data structures (including complex dynamic data structures) do not need to keep track of associated capabilities (see *Sparsity* under *CODOMs* in Section 4.2.1).

Discussion

Traditional memory-addressing capability architectures provide a design that is very enticing in its cleanliness and simplicity. Nonetheless, they are not "sparse"; memory-addressing capabilities cannot easily support multiple isolation domains in the same address space unless all code is fully aware of their existence. That is, one can create a separate capability for every single function and memory allocation, at the expense of making the system non-compatible and less efficient. This extreme situation is externally imposed because dynamic memory allocations and dynamically loaded libraries (e.g., through `dlopen`) cannot be ensured to exist in consecutive virtual memory addresses that are not interleaved with those of other domains. One could pre-reserve large virtual memory ranges to accomodate future memory growth inside a domain; but that is not effective in the general case because it limits the number of possible domains and their maximum virtual memory size.

In contrast, using a per-page tag in CODOMs allows domains to be "sparse", and can be defined by the programmer at the software component boundary. That is, each domain can be composed of non-consecutive virtual memory addresses that can be interleaved with those of other domains. This design allows components unaware of CODOMs to be properly isolated from other domains in the same address space.

5.1.2 Domain Access Permissions

Domain access permissions specify the type of access that can be exercised on any page of a domain, and are used by both code-centric protection and capabilities (see Sections 5.1.3, 5.2 and 5.4). They are implemented as a 2-bit field with four totally-ordered values, which are combined with the per-page permissions when performing access checks:

None: Grants no access at all to any of the pages of a domain.

Call: Allows calling into code as long as the target address is aligned to a system-configurable value. If used through a capability with size zero, the alignment restriction is not applied and return instructions are also allowed.

Note that this permission does not allow regular jumps. This ensures the architecture provides the intended return address according to the call semantics (i.e., the return address is not forged; see Sections 5.3 and 7.2.2 for more details).

Read: Allows executing and reading arbitrary addresses of pages without the capability storage bit. In the latter case, it allows loading capabilities from such pages.

Write: Allows executing, reading and writing arbitrary addresses of pages without the capability storage bit. In the latter case, it allows loading and storing capabilities on such pages.

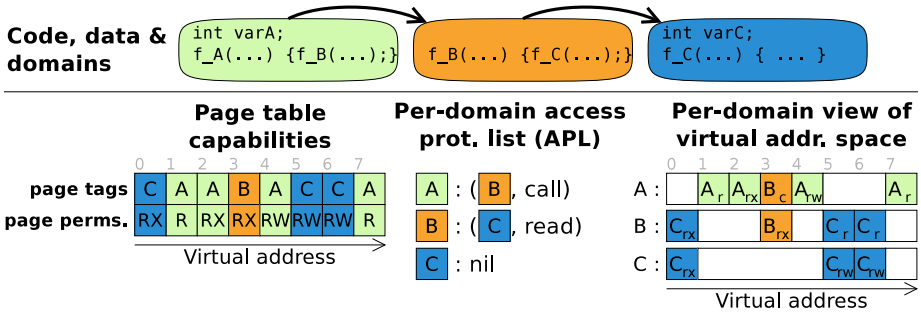


Figure 5.3: Example code-centric protection in CODOMs for three functions on different isolation domains. It shows the page table (left), APL configuration (center) and the resulting access permissions for each domain (right).

Discussion

Combining domain permissions with per-page permissions ensures that, for example, a non-executable page cannot be used to execute code even if the domain is accessed through the *read* permission (which includes support for execution).

Since CODOMs supports an unlimited number of domains, these can also be used as a "unit of sharing". Suppose domain A whants to have read access to data owned by domain B. In the case that domain B also contains code, giving A *read* permission to B would also allow it to execute arbitrary code in B. In this case, an intermediate domain D can be created to hold that data. Domain A would have *read* permission to D, while domain B would have *write* permission to D.

Alternatively, domain permissions could be easily turned into a 3-bit field with orthogonal permissions (*call*, *read* and *write*), instead of the current 2-bit field with totally-ordered values. In such a case, the intermediate domain D would not be necessary. The only limitation to do this in the current design is in the space available in hardware capabilities, but a bit could be easily borrowed for this purpose from other fields present on capabilities (e.g., the *revocation counter value* field; see Section 5.2).

5.1.3 Access Protection List

The Access Protection List (APL) describes the access permissions that a domain has over other domains. Conceptually, it can be understood as a mapping from target domains (i.e., tags) to domain access permissions; on every memory access, an entry is searched that describes the target domain (identified by the tag of the target address' page) and has the adequate domain access permission. CODOMs provides a software-managed cache to perform access checks against this structure, as explained in Section 5.4.

5.1.4 Example of Code-Centric Protection

Figure 5.3 illustrates an example of code-centric protection where three domains (A, B and C) invoke each other. Specifically, routine *f_A* in domain A invokes routine *f_B* in domain B. The latter then invokes routine *f_C* in domain C. The figure also illustrates on its right-hand side how domain access permissions in the APLs are combined with the existing per-page protection bits.

The left side of Figure 5.3 shows that pages 1, 2, 4 and 7 are associated with domain A. This is an example of a "sparse" domain, since its pages are not consecutive. Suppose for a moment that *f_A* wants to

access the variable *varC* in domain C, stored in page 5. The APL for *f_A* (domain A; center of Figure 5.3) does not allow accessing it, but it does allow calling into the entry points of domain B which can access *varC* in its behalf. After calling into *f_B* (the entry point of domain B stored in page 3), CODOMs will use the APL of domain B, which is currently executing. Since the current APL now has read access to domain C, the variable *varC* can be directly accessed by *f_B* without executing any code in domain C. Although not shown in the figure, access permission grants of a domain to oneself's pages is implicit. The example also shows on the right side of Figure 5.3 that, even though page 3 (from domain B) has read and execute permissions, the APL for domain A only allows function *f_A* to call into the entry points of B. Likewise, even through page 5 (variable *varC* from domain C) has read and write permissions, *f_B* is not allowed to write into it because the APL for domain B caps its permissions to domain C to read and execute operations.

The example also demonstrates the ability to directly invoke procedures across domains using regular function calls. The right side of Figure 5.3 shows that once code in domain A (*f_A*) calls into the entry point of B (*f_B*), CODOMs uses B's APL, since the instruction pointer now originates from a page in domain B. This, in turn, enables B to directly jump into code of domain C (pages 0, 5 and 6), which is not directly accessible to domain A. It is important to note that if B's page 3 had the privileged capability (*P*) bit set, a call into B's entry point (*f_B*) would be all it takes to execute privileged code (e.g., a system call). Note also that B can call or jump into arbitrary addresses in C (as long as they are executable), since B's APL is not enforcing any entry point (B has *read* permission to C, instead of having *call* permission). This shows how CODOMs can be used to enforce asymmetric policies where domains are not mutually isolated (e.g., B could be the OS kernel and A and B could be isolated kernel modules or user processes). Also, the type of operations that B can perform on C's pages is capped by B's APL. For example, even though C's pages 5 and 6 are writable, B's APL limits accesses to reads (i.e., B cannot write into C's pages 5 and 6), while C's APL does allow writing into them.

Discussion

Traditional capability systems make it specially tricky to have global variables, which might be shared by multiple domains. Suppose that functions *f_A* and *f_C* require access to a common global variable *v_G*. When the program is started, the loader must create capabilities for domains A and C and load the corresponding code into the memory pointed by each. The loader must also create a capability G for the global variable *v_G* and, importantly, store it twice in the memory pointed by the code capabilities, once in A and once in C. Therefore, when functions *f_A* or *f_C* execute, their respective capabilities will grant access to the appropriate code *and* to a portion of memory that contains a copy of the G capability to access *v_G*. Even though the example describes a shared global variable, the same can be applied to accessing code, for example. Therefore, capabilities A and C in this example are acting as "root capabilities".

Code-centric protection makes this much more simpler to implement, since no capabilities are needed. If variable *v_G* is stored in domain G, and G is in the APL of both A and C, the functions *f_A* and *f_C* immediately have access to the global variable without further ado.

5.2 Capability Protection

Capabilities are termed as *passive* when stored in memory, and *active* when stored in registers. Each passive capability occupies 256 bits (32 B). This is half a cache line in the target architecture, making it simpler to implement atomic reads and writes without having to lock multiple cache lines (remember that accesses to capability storage must are aligned). Each passive capability contains the following fields:

Base address (48 bits): Specifies the base virtual memory address that this capability grants access to.

Size (48 bits): Specifies the size of the grant of this capability. Therefore, the range of virtual memory addresses that can be accessed through the capability is $[\mathit{base\ address}, \mathit{base\ address} + \mathit{size})$.

Tag (64 bits): Specifies which tag's APL must be used for checking accesses performed through this capability. Therefore, a capability can be used to access all pages accessible by code in the given tag.

Access permission (2 bits): Specifies the maximum access permission granted through this capability. The memory access check logic combines this information with the domain access permission for the target tag, as encoded by the APL of the tag of this capability (see Section 5.4).

Revocation counter address (48 bits): Specifies a virtual address that is used for immediate capability revocation (see Section 5.2.5) and/or as a pointer to a protected opaque payload (see below).

Revocation counter value (46 bits): Specifies a value used to verify if a capability has been revoked. If the value is zero, the capability is said to be "synchronous" in which case it is implicitly revoked at cross-domain boundaries at zero cost (it is said to be "asynchronous" otherwise). Section 5.2.5 provides a detailed description of the two capability types and revocation.

CODOMs provides a set of capability registers to store the per-core set of *active* capabilities, and exist side-by-side with the general-purpose registers. The current implementation provides 8 of these registers, which can be managed with the following operations:

Create initializes a capability register. The application must provide all fields but the tag, which is set by the hardware from the executing instruction. Providing a non-zero revocation counter (address or value) requires the privileged capability (*P*) bit. Therefore, creation is safely implemented as an unprivileged instruction as long as a revocation counter is not provided.

Modify only allows code to weaken the access permissions of a capability and to shrink the address range it encodes, either by increasing the start address (and decreasing the size accordingly) or by decreasing the size (ensuring the size never goes below zero).

Copy allows copying capabilities between different registers.

Load and store move capabilities between memory and registers. They are only permitted if: the address is 32 B-aligned, the target page has the capability storage (*S*) bit and the address is accessible with (at least) read or write permissions (for loads and stores, respectively).

Usage performs a memory access checked against the given capability registers. That is, capabilities must be loaded into a register before being used to perform a memory access.

Two usage models are supported. *Implicit use* validates memory accesses against all active capabilities (i.e., all capability registers). This is currently the default. It simplifies compiler support and enables transparently adding capability use to existing code that is oblivious to their existence. *Explicit use* is provided trough separate instructions that identify which capability register to use to check an access. Compilers can use these instructions when they know what capability is to be used by a specific memory access. This mode of operation minimizes the number of capability register checks, improving energy efficiency.

Probe checks if a capability register allows a specific access to a given address. This can be used by code to verify that a capability is valid according to its expectations.

Revocation is allowed if the revocation counter value is not zero (an asynchronous capability) and the capability's tag is the same as the executing instruction. That is, only the domain that created a capability can revoke it.

Discussion

Implicit capability use checks accesses against all capability registers. This functionality is key to allow unmodified code to perform accesses through capability registers that have been set by another (calling) domain. If hardware resources and energy consumption of this approach were an issue, checks can be limited to a single capability in the common case by using an approach similar to the *sidecar registers* found in MMP [124], caching the recently matched capability.

An interesting characteristic of explicit capability check instructions is that they can be used to secure code that processes untrusted inputs against malicious attacks (e.g., to avoid buffer overflows). Fully integrating these with the compiler is the common way in which capability architectures achieve this goal [34]. Nonetheless, capability probing instructions can also be used instead by the programmer to secure code that is otherwise unaware of capabilities. Furthermore, functions sensitive to this type of attacks can be placed on a separate domain to enforce the use of the appropriate exit point even in code that uses implicit capability checks (see Section 5.3 for more details).

The addressing model that allows implicit capability checks also has its downsides: explicit capability use requires passing a capability to check accesses against, and another immediate or general-purpose register that indicates the address to access. Instead, capabilities on the third version of the CHERI design [34] embed an additional offset field that is used instead of the immediate/register argument in CODOMs. This proved to be useful for compiler-assisted pointer protection inside a domain by using capabilities as "fat pointers". Nonetheless, the model in CHERI deters the use of implicit capability checks on existing, unmodified code. Furthermore, adding an offset field in CODOMs would bump passive capability size over the 32 B mark, because CHERI does not support revocation and instead relies on garbage collection (saving that space on capabilities for other purposes). Therefore, 64 B should be used to keep simple memory access atomicity for capabilities, leaving a lot of unused space even if the existing fields in CODOMs' capabilities were enlarged to occupy 64 bits each.

5.2.1 Example of Capability Protection

Given the domain configuration shown in the example of Figure 5.3, assume that function *f_C* (in domain C) wants to read the contents of variable *varA* (in domain A). Since *f_C* cannot directly access *varA* (the APL for domain C does not have read access to domain A) *f_A* must grant *f_C* read-only access to *varA* through a capability.

Function *f_A* will start by creating a read-only capability that marks the extents of *varA* (say, some range inside page number 4). CODOMs will automatically store in the capability the tag of the code creating it, in this case domain A (where *f_A* resides). Having the capability in a capability register, *f_A* can now call *f_B* which, in turn, calls *f_C*, all maintaining the capability in its register. Now, *f_C* can perform a memory access to *varA* using the capability (either explicitly or implicitly). Even though the variable is not accessible through the APL of domain C, the capability uses the APL of domain A, grating access to it. Note that even though the APL of domain A allows write accesses to *varA*, the capability limits it to *read* permission.

Therefore *f_C* cannot write into *varA* using this capability. Of course, the memory range encoded in the capability only allows accesses to *varA*, and not the entirety of domain A.

5.2.2 Capability Confidentiality

The architecture does not provide instructions that allow peeking at the contents of the capability fields. Adding them would be a trivial modification, but it could open the door to attacks that use these values to infer the internal state of the domain that created these capabilities. While this might sound far-fetched, there are examples of working attacks that extract sensitive information like private encryption keys through indirect observation of the state of another computation [88]. In the current implementation, capability content confidentiality can be overridden by privileged code, which has direct access to the contents of passive capabilities.

5.2.3 Capability Unforgeability and Integrity

CODOMs ensures that the unforgeability and integrity properties of capabilities are enforced in the face of unprivileged code. Active capabilities cannot be forged because they can only be created through operations that set the capability register fields based on the tag of the instruction that creates them and, by extension, based on the APL of the creating instruction. Their integrity is also not affected, since the manipulation operations can be trivially shown to never increase permissions nor increase the address range they grant access to. Passive capabilities cannot be forged because pages with the capability storage (*S*) bit cannot be directly accessed for reads nor writes. The same applies to their integrity. These limitations do not apply to code executing with the privileged capability (*P*) bit. Therefore, privileged code has the ability to create new capabilities for domains it does not own.

5.2.4 Domain Capability Stack

Since the regular data stack cannot be used to store capabilities, CODOMs provides a per-thread Domain Capability Stack (DCS) where capabilities can be stored. Just like x86 code uses the `rbp` and `rsp` registers to manage the stack frames, CODOMs provides the `dcsb` (for *DCS base*) and `dcsp` (for *DCS pointer*) registers to manage DCS frames (see top-left corner of Figure 5.1).

In the current implementation, the `dcsb` register can only be modified by privileged code. At the same time, the `dcsp` register can only be modified through capability push/pop instructions, and an exception is raised when a pop instruction surpasses the `dcsb` address (i.e., a negative DCS frame size). The range identified by these two registers acts as an implicit capability with *write* permission, and the TCB must ensure it points to pages with the capability storage (*S*) bit set. Together, these conditions ensure that the DCS is effectively thread-local.

Discussion

In retrospective, the DCS register restrictions were a product of early optimization, when the design of system software (runtime and OS) was not completely fleshed out. First, to provide zero-cost revocation of synchronous capabilities (see Section 5.2.5) the system must differentiate between the DCS and other capability storage pages. Second, the original design envisioned using the DCS as both a stack for capabilities and as a scratch memory space for the TCB to store cross-domain call information. This information was expected to be stored right before the current DCS frame, where the user cannot reinterpret it as a capability.

Later, this proved to make sharing portions of the DCS too complicated (e.g., when capability arguments exceed the number of capability registers and need to be passed through the DCS).

The current DomOS design addressed these concerns without modifying the architecture. Nonetheless, if CODOMs were now redesigned it would be much simpler to have a special type of capabilities for the DCS: an internal privileged bit indicating if the target memory is for the DCS. This would allow user code to freely manage the DCS and its frames by exposing the DCS registers as regular general-purpose registers and by using a capability to grant access to the DCS.

5.2.5 Capability Revocation

As introduced in Section 2.2.4, capability revocation is a very tough problem in both hardware and software implementations. CODOMs has been carefully designed to provide a very efficient implementation for *immediate* and *selective* capability revocation, which is key to allow DomOS to pass capabilities across processes.

The first difference with other systems is in the optimization of capability revocation by distinguishing between *synchronous* and *asynchronous* capabilities, as described in Section 4.2.1. A capability is considered synchronous when its revocation counter value is zero, and is considered asynchronous otherwise. The implicit revocation property of synchronous capabilities is enforced by ensuring they are not shared with other threads through memory; therefore, once a callee domain returns, it no longer has access to the synchronous capabilities it received. Since the number of capability registers is limited, synchronous capabilities are allowed to be stored into the DCS, but not to other capability storage pages. Since the DCS is strictly a per-thread resource (see Section 5.2.4), storing synchronous capabilities in the DCS does not pose any problem.

The second difference with other systems is in efficiently supporting immediate and selective revocation of asynchronous capabilities. Asynchronous capabilities can be stored in any capability storage page, including those of the DCS. Software starts by creating a *revocation counter* in memory. An asynchronous capability is created by passing it the address and value of the revocation counter stored in memory. An asynchronous capability is considered valid as long as its cached version of the counter (the *revocation counter value* field) matches the one stored in memory (retrieved through the *revocation counter address* field). When a revocation instruction is executed, CODOMs first verifies that the instruction's tag matches that stored on the capability, ensuring that only the domain that created a capability can revoke it. CODOMs then immediately revokes it by incrementing the value of the revocation counter stored in memory, thereby invalidating all capabilities that use the same counter (the value cached in the capability does not match the one stored in memory). Therefore, selective revocation is possible by associating a different revocation counter to different capabilities, even if they grant access to the same memory range (i.e., the same logic object from the point of view of the programmer).

Passive (stored in memory) asynchronous capabilities are lazily invalidated when loaded from memory. Hardware will reset their access permission (set it to *None*) if the counter value does not match the one in memory.

Active (stored in a register) asynchronous capabilities that use the revocation counter being revoked are immediately invalidated. The simplest approach is to send an internal signal to all cores, which invalidates all capability registers that have the specified revocation counter address. More efficient alternatives exist, such as using a central directory to track the active asynchronous capabilities. In this case, the revoking core signals the directory, which in turn invalidates all capabilities that use the same revocation counter (similar to the approach used by DiDi for TLB shootdowns [121]). In all cases, since most capabilities are expected to be synchronous (and not asynchronous), this operation will be infrequent.

A revocation counter can be reused $2^{46} - 1$ times until it overflows (raising an exception), and 2^{48} different counters can exist in the system. When a counter is reused after an overflow, the system must ensure that all passive capabilities that use the overflowed counter are immediately invalidated. Nevertheless, the magnitude of the revocation counter ($2^{46} - 1$) makes such events extremely infrequent, and the existence of the capability storage (*S*) bit greatly reduces the number of possible addresses where capabilities can be stored (as opposed to using memory tags to identify capabilities). More importantly, the system is not involved in synchronous capabilities (the vast majority), and does not need to perform garbage collection on the memory the capabilities grant access to.

Discussion

Setting a non-zero revocation counter is a privileged operation because the capability revocation operation performs memory writes based on it. CODOMs could instead verify that the revocation counter address is writable by the requestor before allowing unprivileged code to create an asynchronous capability, or could have an additional per-page bit to identify pages with user-managed revocation counters. Nonetheless, domain switches in CODOMs are sufficiently efficient that this restriction should not severely impact performance. Furthermore, the current design allows the TCB to keep a tight control on asynchronous capabilities and revocation counters. This can be specially important since DomOS allows capabilities to cross process boundaries.

5.3 Enforcing Domain Entry and Exit Points

Entry points are enforced through the *call* permission, which limits cross-domain control flow to call instructions at addresses with a specific alignment. Nevertheless, directly granting *call* permission to a domain containing the code of a regular binary would mean the compiler must carefully lay out code to avoid callers entering in the middle of a function; remember that the only limitations are using call instructions and adhering to the target alignment restriction.

Instead, a properly aligned "proxy" routine is created for each entry point, which serves as a trampoline to the actual entry point. Therefore, the regular callee code does not need to care about alignment. Access to a subset (or all) of these proxies can be granted through the two means available in CODOMs: (1) a capability with *call* permission to the range of memory containing the selected proxies; or (2) a *call* permission to an intermediate domain with the proxies. This second approach is shown in Figure 4.1, where the caller (green) domain has access to a *Proxy* domain which, in turn, has access to a callee (blue) domain; the flow of execution is the same as in Figure 5.3, where function *f_B* (on domain B) would be the proxy for function *f_C* (on domain C). Note that proxies are not strictly necessary; one can directly create a capability with *call* permission and size zero to the address of an entry point (remember that size zero disables alignment checks).

Exit (return) points can also be enforced in CODOMs. It is important to note that a callee domain does not necessarily have access to its caller domain; for example, none of the domains in Figure 5.3 has access to its caller domains. In this case, a caller domain can create a "return capability"; that is, a capability to the return address with its size set to zero, which disables the alignment checks and allows using return instructions.

The discussed approaches are taken by the compiler and OS support described in Chapters 6 and 7 to enforce domain entry and exit points.

Discussion

Providing controlled cross-domain entry points through capabilities and additional proxy domains provides a path for gradual adoption with binary compatibility. Intermediate proxy domains allow cross-domain calls on unmodified code, while capabilities provide tighter control without the need for intermediate domains.

The use of capabilities with a zero size as described raises a very interesting question: how does the system differentiate between capabilities intended for calls and those intended for returns? For domains inside the same application, we can assume components will not willingly attack each other; this is because the programmer is linking them together, so a minimum of trust exists on code being well-behaved. Therefore, in this case there really is no need to check against willful misinterpretations. For domains on different applications, where distrust may exist, an intermediary must be used to exchange the initial access grants. That is, processes in DomOS cannot initially access each other, and the OS must setup an initial communication channel. This is exploited to enforce the use of a proxy domain whose code tracks the proper usage of return capabilities, as described in Section 7.2.2. Performing these checks in software provides the intended safety, but at the expense of a few additional instructions. A future CODOMs implementation might simplify this by adding a per-capability bit that clearly differentiates which capabilities are meant to be used for cross-domain returns.

5.4 Implementation of Access Protection Checks

Figure 5.1 shows the steps followed by the information involved in implementing access protection checks. The APL describes the access permissions that a domain has over other domains. ⓿ The hardware caches these permissions in a per-CPU *APL cache*. It is a software-managed cache that is refilled by the OS, allowing the multiplexing of the unbounded number of tags without changing the page table entries. The APL cache maps the *Tag* field found in the page table capabilities into a smaller *HwTag* field and a *HwAPL* field that encodes the access permissions of domain *HwTag* for all other domains currently cached in the APL cache. Therefore, the *APL cache* can be understood as the "access control matrix" of the domains currently cached by this core at a specific moment in time [69]. If a domain is not present in an APL, the *None* permission is used in the *HwAPL*. The APL cache exposes following privileged operations:

`aplcache_get`: Gets the contents of the given entry.

`aplcache_set`: Sets the contents of the given entry.

`aplcache_reset`: Triggers a recalculation of the *HwAPL* fields stored in other hardware structures of this core. A bitmask argument indicates which *HwTag* values need recalculation; i.e., recalculate the *HwAPL* corresponding to the selected *HwTag*s.

`aplcache_index`: Returns the index of the entry used to cache the information of a given tag (see Section 7.4.4).

`aplcache_lru`: Returns the index of the least recently used entry.

The current implementation has 32 entries in a fully-associative configuration. Therefore, each entry contains 128 bits: 64 bits for the *Tag* field (used for lookups), and 32 2-bit entries for the permissions of the 32 cached tags (the *HwAPL* field of each entry). The *HWTag* field requires 5 bits but is not stored in the cache, since it is implicit in the entry's index. Section 8.1.4 shows that 8 entries are enough in complex scenarios (99.6% hit rate for isolating all Linux kernel modules). Nonetheless, a 32-entry configuration

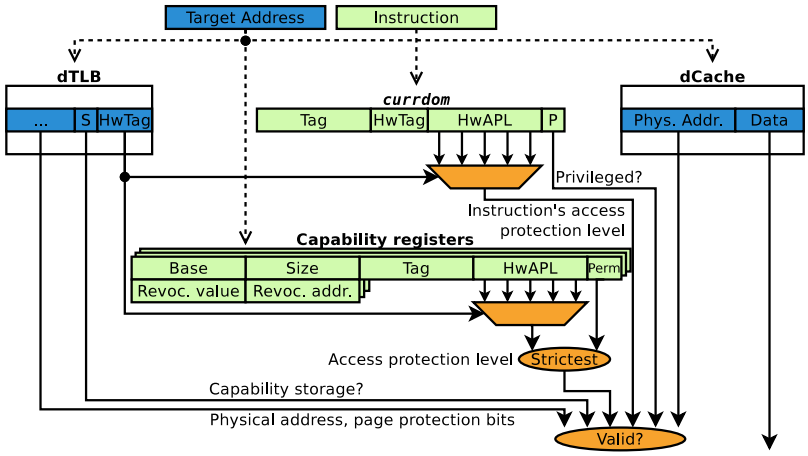


Figure 5.4: Access protection check logic for memory accesses.

provides a reasonable trade-off between hardware costs and a high APL cache hit ratio for more aggressive system configurations that use a larger number of domains (i.e., systems that provide finer-grained isolation).

Entries on the data and instruction TLBs are extended to contain the *HwTag* corresponding to the page table entry they describe. ❶ On a TLB refill, the hardware page table walker issues an `aplcache_index` operation to retrieve the *HwTag* corresponding to the *Tag* encoded in the page table entry. If the mapping does not exist, an exception is raised, in which case the OS must multiplex one of the entries with the new tag (e.g., using `aplcache_lru` to evict the least recently used tag and using `aplcache_reset` to inform of the change to other hardware structures in the core). Additionally, the instruction TLB is extended to store the *privileged capability* bit (*P*) from the page table entry. Likewise, the data TLB is extended to store the *capability storage* bit (*S*). This information can be stored on a separate structure to optimize energy and delay, since it is only needed after a TLB hit.

❷ When an instruction is fetched, the relevant information from the instruction TLB is stored in the architectural register `currdom`. This register encodes the information of the *current domain*, acting as an implicit capability to the program counter. Since many instruction sequences reside in the same page, this can be optimized to reduce the number of instruction TLB lookups and the amount of storage required to pass that information. ❻ Whenever the contents of the `currdom` register are modified (i.e., a domain switch takes place), its previous value is copied to the architectural register `prevdom`.

❸ When a capability is created, the *Tag* and *HwAPL* of the `currdom` register are copied into the capability register. This ensures a capability always has the same permissions of the instruction that created it, and avoids lookups in the APL cache when the capability register is used. ❺ When a capability is loaded from memory, the `aplcache_get` operation is used to copy the corresponding *HwAPL* field into it; this ensures *HwAPL* fields are never stored in memory, limiting tag multiplexing to changes in the state of a single core.

❹ An access check utilizes information from the `currdom` and capability registers, as depicted by Figure 5.4. The *HwTag* in the data TLB is used to index the *HwAPL* of the `currdom` register and retrieve the protection level for the target address. Conversely, the *HwTag* in the `currdom` register is used to check control flow instructions. The same applies to capabilities, except that CODOMs uses the stricter value between the selected *HwAPL* entry and the capability's permission. The protection checks are performed in

parallel to the actual cache access, effectively hiding their latency.

Discussion

The OS uses the `aplcache_reset` operation to multiplex the set of active tags. This reloads the corresponding *HwTag* and *HwAPL* values in the capability registers and TLB entries of that CPU. No TLB shootdown-like operations are required, since there is one independent APL cache per CPU, and its information does not leak into passive capabilities.

Multiplexing a *HwTag* typically requires changing the *HwAPL* of multiple other *HwTag*s (i.e., other APL cache entries that had access to the evicted *HwTag*). To make this more efficient, a second bitmask argument could be added to `aplcache_reset` to support lazy tag multiplexing. This would reset, for all APL cache entries, the *HwAPL* bits corresponding to the *HwTag*s identified in this second argument; i.e., the index'th permission of all entries is set to *None* for each index in the bitmask,

Exposing the identity of the previously executing domain (`prevdom` register) can be used by the OS to identify which domain raises a software interrupt or an exception without having to track every cross-domain call or traverse the page table to retrieve the domain of the previously executing instruction.

5.5 Efficient Execution in Out-of-Order Pipelines

A challenging design point is how to make cross-domain calls efficient in a high-ILP out-of-order processor.

Protection Domain Checks and Switches on out-of-order processors are efficiently implemented through the `currdom` register. The information provided by the instruction TLB is sent to the rename stage. The `currdom` register is renamed every time a domain switch occurs (when its contents change). Since the register is not changed beyond that stage, the rename stage itself can set the new value and mark the register as ready. Similarly, the `prevdom` register points to the physical register used for `currdom` before renaming it. This eliminates RAW hazards during domain switches and allows maintaining instructions from different domains in-flight. Importantly, this efficiency includes switching from privileged to non-privileged code, since `currdom` also contains the privileged capability (*P*) bit (i.e., zero-latency system calls). Experimental measurements show that 6 physical `currdom` registers are sufficient to eliminate all RAW hazards on a tight call/return loop. Moreover, for functions doing some actual work, 4 physical `currdom` registers are sufficient to make the domain switch completely seamless even on short routines.

Capability Registers may also generate RAW hazards when modified and used implicitly. CODOMs alleviates this by providing a 2-wide register window for active capabilities. Register `capX` is used by the current instruction sequence, and `capXn` for the "next" window. Writing into `capX` also writes into `capXn`. The windows are swapped on protection domain switches, thus allowing software to eliminate RAW hazards on `capX`. This can be used, for example, by highly-optimized proxies that interface with code running in the implicit capability use mode.

5.6 Revisiting Isolation Scenarios

The presented code-centric protection and implicit capability use provides a form of limited ambient authority for memory and privilege protection. It is ambient authority because it is not explicitly exercised

(e.g., all capabilities are implicitly checked). It is limited in the sense that it does not necessarily grant access to the entire address space. This brings its own benefits when using CODOMs on code with none to minimal changes. Hierarchical isolation (sanboxing and privilege amplification) can be easily implemented only using code-centric protection (i.e., page table capabilities and APLs). Since different domains can have interleaved pages, there is no need to partition the virtual memory space beforehand to accommodate all possible domains. Only small wrappers are necessary for calls to sandboxed domains when they receive information from their parent domain. Since all capabilities are implicitly used by default, no changes are necessary on the sandboxed code.

The same can be applied to non-hierarchical isolation when domains communicate through a common parent domain. Process-level isolation is a clear example of this. System calls can be replaced with function calls to the OS kernel domain, and each process can be allocated on a separate domain inside the same page table. In fact, this is the approach taken by DomOS (see Chapter 7), only that cross-process calls are further optimized.

Stricter isolation can also be achieved by moving into explicit capability use, at the expense of extensive compiler changes and/or code annotations. If all code running in a system arrived at such level of integration, the architecture could be simplified in favour of relying solely on capabilities. In this case the APL, APL cache and the *HwTag* and *HwAPL* fields would no longer be necessary, while the other positive aspects of the CODOMs design could be retained.

Chapter 6

Compiler Support

Experimental compiler support has been implemented to ease the management of isolation domains using simple programmer annotations. These annotations can be used to split the different components of an application into domains, define their entry points and establish the isolation policies between domains. They are synthesized into ancillary data structures in the output binary, which are later fed into the program loader in DomOS to automate the initialization of isolation domains. Annotations are sufficiently flexible that different policies can be expressed, including asymmetric isolation policies.

6.1 Language Interface

The compiler provides the following annotations:

Domain assignment: The macro `__dom("<name>")` allows assigning global symbols (code and data) into the specified domain. Any unassigned symbol is implicitly assigned into a default domain.

Domain-level permissions: The macro `__perm("<regexp>", "<perm>", "<regexp>")` can be used at global scope to indicate the access permissions (i.e., APL configuration) between domains of the same application. By default, no permission is granted. The regular expression arguments select which domains should be granted the given permissions. For example, `__perm("d1", "call", "d2")` specifies that domain *d1* should be granted *call* permission to *d2* in its APL.

Entry point identification: The macro `__entry()` can be used in function symbols to identify them as domain entry points, analogous to "extern" in C.

Entry-level isolation: A set of `__iso_*` macros define the set of isolation-sensitive actions that should be performed when calling into or receiving a call from another domain or entry point (see below). By default, no isolation-sensitive action is taken. The actions can be specified for interactions with a specific domain or entry point name.

There also are variants of the annotations with the "`_push`" and "`_pop`" suffixes to specify information for blocks of code (e.g., multiple consecutive functions), minimizing programmer hassle. The annotations account for order, so one can use regular expressions to refine permissions over a previous annotation. Each isolation-sensitive action is defined as a pair of isolation property (integrity or confidentiality) and a resource (register, data stack and DCS):

Register integrity translates into making a copy of live registers before a cross-domain call, and restoring them afterwards. Only applies to callers.

Register confidentiality translates into zeroing non-argument registers (for callers) and non-result registers (for callees) that are known to contain confidential information. Without further compiler support all these registers must be assumed to contain confidential information, but differentiating which information is confidential could further reduce the number of registers that must be zeroed.

Data stack integrity translates into creating a capability for in-stack arguments before a call, and another capability that points to the rest of the stack. Only applies to callers. This leaves previous stack frames inaccessible to a callee. As explained later in Section 7.2.2, the stack is accessible through a capability.

Data stack confidentiality translates into using separate stacks for the caller and callee. It applies to both callers and callees.

DCS integrity translates into adjusting the DCS base pointer (`dcsb` register) to exclude non-argument entries. Only applies to callers.

DCS confidentiality translates into using a separate DCS for the caller and callee. It applies to both callers and callees.

An additional action controls process switches which, in the current prototype, control switching the process identity; namely, resource accounting (e.g., CPU time and memory) and software resource isolation (i.e., file descriptor table). Therefore, it is possible to call into a function of another process with and without switching the process identity (see Section 7.2.2).

Establishing isolation actions at this very low granularity provides a very tight control of what isolation actions the programmer deems necessary. By only generating code for the desired actions, performance can be kept closer to that of a regular function call while preserving the isolation required by the programmer. Note that higher-level or simpler policies can be implemented on top as a combination of these annotations.

Most of these actions involve unprivileged resources, like registers or the data stack. Therefore, the compiler can inject caller and callee stubs on entry point call points and entry point definitions (respectively) to implement most isolation actions at user level. This removes code from the TCB, while still ensuring each domain handles its own share of the isolation work. If a programmer mis-configures the isolation actions of a caller domain (making them too unrestrictive), this will never expose the callee domain (which has its own stub). For example, if a caller does not enforce register confidentiality, the callee can still do so (since this is implemented on the respective stub).

Whenever a privileged resource is involved, the action is implemented on a privileged proxy routine managed by the TCB (see Section 7.2.2). This includes process identity, the DCS and data stack confidentiality (this last one for performance reasons). In this case, DomOS ensures that trusted proxies satisfy the actions requested by both the caller and the callee domains. Therefore, a mis-configuration of the isolation actions of a caller will never inadvertently affect the actions requested by the callee. For example, if the callee requests a process identity switch, the trusted proxy will do so regardless of whether the caller also requested it.

Discussion

Implementing isolation actions in user-level code provides two benefits. First, the size of the TCB code is reduced. Second, low-level knowledge from the compiler can be exploited to minimize the costs of operations like register integrity.

This knowledge is available to the compiler through the target architecture's ABI (and the function signature) and information from the architectural state at a call or return site. The ABI conventions define how to map arguments into registers and the data stack, and also define what registers should be preserved by a callee across function calls (i.e., must be restored to their original value before returning). For example, the ABI for x86–64 defines registers `rdi` and `rsi` as the locations for the first and second arguments, respectively. It also defines that `r11` and `r12` are never used for arguments, but only the latter must be preserved across function calls.

For a call to a function with one argument, the compiler will already safeguard the values for `rdi`, `rsi` and `r11` if they are later used on the caller function (i.e., they are live at the call site). This is because the ABI defines they are not preserved across function calls, while the caller will restore the value of `r12` before returning (in case it was modified). Now, if the target function is on a separate domain of the same application, the programmer does not need to enforce register integrity. The code, generated by the compiler, can be trusted to follow the ABI and therefore to return with the appropriate register values. The same can be applied when performing a call to some other application that the programmer trusts to be well-behaved (e.g., a system service). If the target code cannot be trusted to faithfully follow the ABI conventions, register integrity can be greatly optimized with knowledge from the call site. The caller stub only needs to safeguard register `r12`, and only if it is live at the call site. Registers `rdi`, `rsi` and `r11` are already managed at the call site by the regular ABI rules. Similarly, a caller stub providing register confidentiality does not need to zero register `rdi`, since the ABI establishes it as non-confidential by virtue of being an argument.

Stubs can be inlined into the point where they are used, further optimizing their contents with knowledge of what registers are live at the call or return site. Nonetheless, optimizations like handling register integrity only for live registers can make debugging the caller code harder in case the callee fails. In such case, it is very easy to have a compiler flag to perform less aggressive optimizations for debugging builds. Alternatively, inlining can be avoided and, intead, select between an optimized and a non-optimized stubs at run-time depending on on whether the application is being debugged.

6.2 Implementation

Compiler support is extended for the C and C++ languages using a wrapper for the target compiler program. The wrapper is a source-to-source compiler that uses LLVM's *clang* library to generate new source files from the original inputs according to programmer annotations. Internally, the wrapper injects a header using the `-include <file>` command-line argument, which uses defines to map these annotations into the parseable attribute `__attribute__((annotate(("<annotation info ...>"))))`. In the case of global annotations (i.e., the push/pop variants), a phony function declaration with the appropriate attribute is generated instead. The wrapper then replaces the original source files with the newly-generated ones to invoke the target compiler, which will generate the binary files. Domain assignment is translated into generating code and data on specific ELF sections [71] in the form `.dom.<name>.<type>`. For example, code for domain *d1* will be generated in section `.dom.d1.text` instead of `.text`. In fact, sections of the form `.<type>` are regarded as pertaining to the default domain (e.g., a non-annotated function will be placed on the `.text` section). Other annotations generate additional sections that are later parsed by the program loader. For every entry point, the number of data stack and DCS arguments must also be stored to properly handle stack switching actions (see Section 7.2.2).

The program linker is also extended with a wrapper. It replaces the linker script used by the target linker with one that is enhanced to manage the additional DomOS-specific ELF sections. For each of these sections, it generates two symbols that identify their bounds once loaded, so that the information can be

easily parsed by the loader. Both the compiler and linker also provide additional command-line arguments that can be used to manage domain assignment at the object file and ELF section level.

The callee stub names are generated in a way similar to C++ name mangling, since the same entry point can have different callee stubs depending on who is calling it. Therefore the compiler registers the addresses of entry point callee stubs in the section that describes entry points, instead of registering the entry point addresses. Uses of an entry point symbol of another domain are replaced with the name of the caller stub. In turn, the caller stub calls into an intermediate symbol that is resolved at run-time by the program loader (see Section 7.3.2).

Discussion

Since the current compiler prototype relies on source-to-source transformations, it cannot reliably determine how many registers and how much stack space is used for arguments, nor cannot determine which registers are live or contain confidential information at a call point. Therefore, the compiler is currently unable to reliably generate caller and callee stubs, relying on the programmer to write them instead. Nonetheless, the state isolation results for C++ presented in Section 3.2 suggest that working closely with the low-level layers of the compiler could provide state integrity at extremely low costs. The same applies to capabilities, even though recent works demonstrate it is feasible to have the compiler automatically manage them [34].

Nonetheless, many interfaces of existing software components try to minimize the use of pointers when designed to operate across isolation domains (e.g., across processes). This makes it more feasible to extend annotations to selectively manage capabilities for function arguments. It also makes it easier to integrate with existing code, instead of treating every single user-level allocation as a capability [34].

6.3 Example

Figure 6.1 shows a simple example of how to use the source code annotations, and how they influence the generation of the resulting binary. The target goal is the following: a function `funcA` generates some value in variable `varA`, and then performs a call into function `funcB` in some other domain that computes a result based on the value of `varA`. To make this happen, the value of `varA` must be either passed as an argument, or must be accessible by both domains (the latter case is shown in the example).

The annotation in Line 3 of Figure 6.1a assigns variable `varA` to domain *A1* (`_dom("A1")`), while Line 5 assigns function `funcA` to domain *A2*. Finally, Line 14 assigns function `funcB` to domain *B*. This results in generating the corresponding code and data on separate sections of the output binary. This can be seen in lines Lines 2, 7 and 12 of Figure 6.1b, where sections are tagged with the name of the domain they are part of (i.e., sections `.dom.A1.bss`, `.dom.A2.text` and `.dom.B.text`).

Lines 1, 2 and 12 of Figure 6.1a establish the permissions of these three domains. This generates additional information on the output binary, which is later used by the program loader to configure the APL of each domain. The first two establish the permissions to domain *A1*, shown by Lines 4 and 5 of Figure 6.1b. Line 1 lets all domains read the value of `varA` (domain *A1*), while Line 2 lets domain *A2* also write into it. This allows `funcB` to access `varA` (Line 17) without passing a capability to it. In turn, Line 12 of Figure 6.1a lets `funcA` perform the call to the entry points in domain *B*, which is encoded in Line 18 of Figure 6.1b. Note that the annotation sets a permission of *entry*, instead of the expected *call* permission. This is to differentiate between direct *call* access to a domain (where entry points must be aligned; see Section 5.3) and call access to a domain with auto-generated proxies to the actual entry points (see Section 7.3).

Line 6 of Figure 6.1a establishes the caller-side of the isolation policy when `funcA` calls into `funcB`. In this case, it establishes that calls into functions of domain *B* should ensure the integrity of registers (i.e.,

```
1  _perm(".*", "read", "A1");
2  _perm("A2", "write", "A1");
3  int varA _dom("A1");
4
5  void funcA(void) _dom("A2")
6    _iso_caller("B", "register", "integrity")
7  {
8    varA++;
9    printf("%d\n", funcB());
10 }
11
12 _perm(".*", "entry", "B");
13
14 int funcB(void) _dom("B") _entry()
15   _iso_callee("A2", "dcs", "confidentiality")
16 {
17   return varA;
18 }
```

(a) Source code annotations.

```
1  .dom.A1.bss:
2      varA
3  .dom.A1.iso:
4      dom, .*, read
5      dom, A2, write
6  .dom.A2.text:
7      funcA
8  .dom.A2.entry_ref:
9      funcB_from_funcA, funcB,
10         signature
11 .dom.B.text:
12     funcB
13     funcB_callee_stub
14 .dom.B.entry:
15     funcB_callee_stub, funcB,
16         signature
17 .dom.B.iso:
18     dom, .*, entry
19     dcs, A2, confidentiality,
20         funcB_callee_stub
```

(b) Sections and information on the binary.

Figure 6.1: Example of source code annotations.

saving them into the stack before the call, and restoring them afterwards). This generates a caller stub for `funcB`, which is inlined into the caller at Line 9 of Figure 6.1a.

The cross-domain call is possible because `funcB` is declared as an entry point for domain *B* (`_entry()` annotation in Line 14 of Figure 6.1a). In addition, Line 15 of Figure 6.1a sets the callee-side of the isolation policy for calls to `funcB`. This results in creating a callee stub (`funcB_callee_stub` in Line 13 in Figure 6.1b) that wraps a call to `funcB` to enforce the selected policy. In this case, the policy requests maintaining the confidentiality of the DCS, so the stub is empty because the DCS cannot be managed by unprivileged code (see Section 7.2.2). Instead, some additional information is generated on the output binary that can be later used by the program loader to generate a proxy that implements this part of the policy. This information includes registering the callee stub as an entry point for `funcB` (Lines 15 and 16 of Figure 6.1b) and recording the requested policy (Lines 19 and 20 of Figure 6.1b).

The actual call into `funcB` performed in the caller stub of `funcA` is substituted with a call to an unresolved symbol `funcB_from_funcA`. This symbol is later resolved by the program loader at run-time to point it to the appropriate proxy or callee stub, which is decided based on the isolation information for `funcB` embedded in the binary (Lines 9, 10, 15 and 16 of Figure 6.1b; see Section 7.3.2).

Chapter 7

Operating System Support

DomOS examines the isolation interfaces in a way that eliminates the semantic mismatch between regular functions and IPC (see Chapter 3). It removes unnecessary overheads and provides copyless and synchronous call semantics across domains (including processes) without going through the OS kernel. Applications can define their own isolation domains, wherein processes are just a group of domains. By placing all processes in a shared address space, threads can safely call into routines in other processes, and arguments are communicated without copies through the hardware capabilities of CODOMs. Instead of providing fixed isolation policies across domains, DomOS exposes a list of critical resources that programmers can "opt-in" to isolate if necessary. Non-privileged resources are directly managed by user-level code, making isolation simpler and more efficient (see Chapter 6). With this flexibility, applications can build the most adequate policy for their needs on a per-function/call-site basis, ranging from ring-like hierarchical isolation to full-blown RPC-like isolation at a fraction of the run-time costs. Furthermore, this efficiency opens new possibilities for component and process compositions that were until now unfeasible due to existing overheads.

DomOS has been prototyped on top of the Linux kernel, but the concepts presented here can be applied to other OSs as well. The aforementioned goals are achieved using the following mechanisms:

- Applications create domains through software *domain capabilities*, and configure how they can access each other through *domain grant capabilities*. The DomOS kernel then maps this information into the underlying CODOMs architecture (using tags and APLs, respectively).

- Applications identify the entry points of domains and the policies they want to enforce on callers through software *entry point capabilities*. Callers can later provide which policies they want to enforce on callees and an entry point capability to DomOS, which uses the information at run-time to generate thin trusted proxy routines that bridge calls between these domains. Proxies implement the requested policies in the most efficient way, and only manage the isolation of privileged resources (whereas isolation of unprivileged resources is managed by the application).

- The code annotations described in Chapter 6 are consumed by the DomOS program loader and runtime to automate the use of the two mechanisms above.

7.1 Processes and Threads

Processes in DomOS function as persistent resource containers, holding multiple domains and threads. Processes (and therefore domains) in the system share a 64-bit address space, similar to a Single Address Space OS (SASOS) [32, 58, 60, 94, 113].

A thread in one process is allowed to call into an entry point in another process, but UNIX systems use user identities to control when a thread can be killed by a given user. Therefore, DomOS maintains existing thread control policies by exposing a different thread identifier for every primary thread and process pair, similar to what is provided by the concept of migrating threads on other OSs [35, 49, 57].

Discussion

Instead of processes, one could provide a new "service" abstraction to manage persistence and resource containers orthogonally to threads and processes; i.e., a service could be used as if linking against an isolated third-party persistent library. This would require registering services into the system, initializing them either when registered or the first time they are used. Persistence is simple enough to maintain (by keeping their state present in memory or swap space), but resource containers require identifying a subject to charge resource requests to, and other subjects need a mechanism to manage computations taking place inside a service (e.g., killing a thread). Since DomOS is implemented as an extension of Linux, it is simpler to reuse processes in a way that maintains backwards compatibility; resources are charged to the user that starts a process, and threads inside a process have different thread identifiers that can be controlled separately. An alternative "service" abstraction would be simpler to implement if backwards compatibility was not a concern. For example, it could be implemented similarly to the *constructor* abstraction in the object capability system EROS [105]. Migrating threads provide a similar design to that of DomOS. In the implementation found in Mach [49], processes acting as services register an *activation block* with the kernel. When a client performs an RPC with the service, an unused activation is selected, which appears as a separate thread to a user observing the system. Nonetheless, reusing activations has its own problems; an activation can appear as a different thread each time it is used (regardless of the process that triggered it), and reusing them makes it harder to keep persistent per-thread information that easily correlates to the primary thread that it is servicing.

Previous OSs implemented the migrating threads concept under the observation that dissociating the thread and process abstractions improved CPU time accounting. As a side effect, they also eliminated the need for synchronizing threads of communicating processes. The main difference with DomOS are that it: (1) maintains stable thread identifiers and per-primary-thread persistent state, (2) provides cross-process calls in a way that optimizes the steps necessary to maintain isolation, (3) eliminates data copies, and (4) does not have to go through the OS kernel.

Since DomOS uses a single addres space, components must be compiled as Position-Independent Codes (PICs) [71]. In fact, this is already commonplace to increase security through Address Space Layout Randomization (ASLR) [50, 90]. For the sake of backwards compatibility, processes are still created using the POSIX *fork* operation in DomOS, maintaining their traditional copy-on-write semantics. After a *fork*, the process uses a separate page table and loses its ability to interact with domains on other processes. When the POSIX *exec* operation is invoked with a PIC executable, the new contents are merged into the global page table and the temporary page table is destroyed. Existing systems already provide alternatives like *posix_spawn* and *vfork+exec* to initialize a child without executing code in its context, avoiding the costly copy-on-write and eliminating the need for a temporary page table.

Since threads can cross process boundaries, the common approach of using garbage collection for ca-

pability revocation cannot be easily applied here unless a large part of the language runtime and garbage collection is added as part of the TCB. A simpler approach is to forbid capabilities from crossing process boundaries, but that defeats the point of using capabilities to avoid a trusted intermediary to perform copies at domain boundaries (like conventional OS kernels do through their IPC primitives). As previously discussed, CODOMs provides a good substrate to efficiently manage revocations (see Section 5.2.5).

7.2 Low-Level Isolation Interface

The DomOS kernel provides two sets of abstractions to manage isolation: domains and entry points. They can be directly used by applications, or left to the DomOS program loader and runtime to automate their management.

7.2.1 Domain Management

Domains serve as the memory isolation unit of DomOS, composed of an arbitrary collection of code and data pages. They are managed through the operations in Table 7.1. The software *domain capabilities* (dom_x) are used as handles to the underlying CODOMs tags. In turn, domain capabilities can be used to create *domain grant capabiltiies*, which are translated into managing the corresponding APL in CODOMs; that is, which other domains can be accessed by a domain (e.g., dom_{dst} and dom_{src} in `grant_create`, respectively). Both types of software capabilities are implemented as file descriptors operated through `ioctl`, and can thus be passed around across processes (i.e., using UNIX sockets).

Domains in DomOS have the same permissions used by CODOMs (*call*, *read*, *write*), plus an additional *owner* permission. The *owner* permission is required to perform operations that modify domain's grants (see Table 7.1). By default DomOS associates memory to the allocating domain (e.g., through `mmap`). It also provides functions to explicitly allocate memory on a given domain (`dom_mmap`) and to "remap" pages from one domain to another (i.e., change their tag with `dom_remap`).

7.2.2 Entry Point and Cross-Domain Proxy Management

Entry points are used by domains to enforce where other domains can call into. They do not have a special representation in CODOMs, except that a *call* permission is necessary to enforce known addresses and call instructions are used (see Section 5.3). Therefore, it is sufficient for applications to use domain and grant capabilities. For example, suppose dom_{dst} wants to grant access to some of its functions to dom_{src}. In this case, dom_{dst} can create a "proxy" domain dom_{proxy} (with *read* permission to dom_{dst}) that contains trampolines to the functions in dom_{dst} (to avoid alignment issues in regular code). In turn, dom_{dst} can pass a copy of dom_{proxy} (with *call* permission) to dom_{src}, who will install it in its APL with `grant_create`. In fact, the functionality provided by the intermediate proxy is very similar to that of the Procedure Linkage Table (PLT) [71], and can be easily implemented by user code.

Nonetheless, there are isolation-critical resources that cannot be managed by user code, like the DCS or the set of software resource that are associated to a process (i.e., open files). To this end, DomOS provides the operations shown in Table 7.2 to create privileged proxies that can manage these resources.

Applications use the `entry_register` operation to register a set of functions that act as entry points to a domain, and returns an entry capability ($entry_e$). Each entry point descriptor contains the target's function address, its signature (argument register set and number of data stack and DCS arguments), and the set of isolation properties that it (the callee) wants to enforce on its callers. These properties are a subset of the ones used by the compiler annotations described in Section 6.1; namely, data stack confidentiality, DCS integrity,

Operation	***Effect***
`dom_default()` → $\texttt{dom}_\texttt{d}$	Return domain capability dom_d for current process' default domain.
`dom_create()` → $\texttt{dom}_\texttt{d}$	Return domain capability dom_d with *owner* permission to a new tag $dom_d.tag$.
`dom_copy(`$\texttt{dom}_\texttt{src}$`, `$\texttt{perm}_\texttt{dst}$`)` → $\texttt{dom}_\texttt{dst}$	Return domain capability dom_{dst} with permission $perm_{dst}$ and dom_{src}' tag iff $perm_{dst} \leq dom_{src}.perm$.
`dom_close(`$\texttt{dom}_\texttt{d}$`)`	Release domain capability dom_d. Tag $dom_d.tag$ is released iff there are no more domain or grant capabilities referencing it.
`dom_mmap(`$\texttt{dom}_\texttt{d}$`, ...)` → `...`	*mmap*-like allocation on specified domain iff dom_d has *owner* permission.
`dom_remap(`$\texttt{dom}_\texttt{dst}$`, `$\texttt{dom}_\texttt{src}$`, addr, size))`	Reassign selected pages from dom_{src} to dom_{dst} iff pages are in dom_{src}, and both dom_{src} and dom_{dst} have *owner* permission.
`grant_create(`$\texttt{dom}_\texttt{src}$`, `$\texttt{dom}_\texttt{dst}$`)` → $\texttt{grant}_\texttt{g}$	Return domain grant capability with $dom_{dst}.perm$ permission to dom_{dst} in dom_{src}' APL iff $dom_{src}.perm == owner$.
`grant_revoke(`$\texttt{grant}_\texttt{g}$`)`	Set permission to *none* for $grant_g.dst$ in $grant_g.src$' APL.
`grant_close(`$\texttt{grant}_\texttt{g}$`)`	Release domain grant capability $grant_g$.

Table 7.1: Domain-management operations in the DomOS kernel.

Operation	***Effect***
`entry_register(`$\texttt{dom}_\texttt{d}$`, count, entries[count])` → $\texttt{entry}_\texttt{e}$	Return entry capability $entry_e$ for the given entry descriptors iff $dom_d.perm == owner$ and all descriptors point to dom_d.
`entry_request_size(`$\texttt{entry}_\texttt{e}$`)` → `size_t`	Return the number of entries registered in the entry capability $entry_e$.
`entry_request(`$\texttt{entry}_\texttt{e}$`, count, entries[count])` → $\texttt{dom}_\texttt{p}$	Return domain capability dom_p with *call* permission to a new domain with proxies to $entry_e$ iff $\forall i < count : entries[i].signature == entry_e.entries[i].signature$. Each descriptor is set to its proxy's entry point on return. Per-entry isolation properties are $entries[i].props \cup entry_e.entries[i].props$.
`entry_alignment()` → `size_t`	Return the currently configured entry point alignment value.

Table 7.2: Entry point-management operations in the DomOS kernel.

DCS confidentiality and process switch. That is, those that are not purely enforced at user-level by the caller and callee stubs. Issuing an `entry_request` operation actually creates the proxies for each of the entry points in the entry point capability, provided that function signatures match. Entry point capabilities can also be passed across processes to establish cross-process calls, in which case the *process switch* property is implicitly set on all entries. The result is a domain capability that contains the code for each of the proxies (the domain has access to all other domains and its pages have the privileged capability bit). Also, the entry descriptors passed through the *entries* argument are modified to point to the proxy addresses, so that a caller can redirect execution to the proper proxy address. At this point, a caller domain can use `grant_create` to have access to the proxy functions and call into them.

Instead of using a generic routine in the OS kernel to manage isolation (as happens with traditional IPC), DomOS removes the OS kernel from the critical path and instead generates the optimal proxy for the selected isolation properties and function signature. The DomOS kernel contains a set of *proxy templates*, one per combination of signature and isolation properties. The appropriate template is copied into the actual proxy location, and its contents are adjusted via a very simple symbol relocation process [71]. The number of templates is relatively low, and they are generated from a single source file. Register signatures are used to select the template that makes the best use of unused registers.

All proxies make sure that caller and callee have a valid stack; they start by verifying that the caller's stack pointer and stack capabilities point to the thread's stack, and restore the stack pointer after the callee returns. This ensures a callee can use the stack to safeguard thread-specific information in its caller stub. To this end, proxies replace the callee's return address with one in the proxy and create a CODOMs capability with call permission to it (since the callee does not have access to that address). Note that checking the stack is not strictly necessary, since an additional isolation property could be added so that callers and callees can identify when stack register integrity is necessary.

Discussion

When generating proxy routines, DomOS checks that function signatures provided by the caller (through `entry_request`) match those provided by the callee (through `entry_register`) to ensure one does not trick the other on what registers or portions of the stack are really unused.

Data stack confidentiality is implemented by changing the stack pointer and transferring in-stack arguments between the caller and callee stacks, similar to GCC's split stacks [114]. This property is implemented in the trusted proxy for performance reasons, since doing it on the user code would require twice as many switches and copies unless the function call ABI was changed to support in-stack arguments existing on a separate stack.

Invoking `entry_register` and `entry_request` from the same process generates non-process-switching proxies by default. Access to this proxy can later be granted to a different process, in which case the call will not perform a process switch. Nonetheless, this does not break DomOS, since the callee is aware that access to proxies did not happen through `entry_request` (i.e., it must have been a conscious decision from the callee).

Since proxies are privileged, they must make sure the return address they receive is valid. Proxies retrieve the return address from the stack before injecting their own, and restore it before returning into the callee. This opens up two vulnerability windows where another thread could overwrite this value to subvert the proxy's return operation. The second window (restoring the return address and returning) can be fixed by using a jump to the return address in the proxy code, instead of using a return instruction. Nonetheless, experimental measurements show that this interferes with the return address stack predictor. Still, the first window (reading the original return address) remains open. Therefore, DomOS uses a private per-thread

Operation	*Effect*
`cap_create(base, size, perm, async, payload)`$\rightarrow$ `cap`$_c$	Create capability that can be asynchronous and/or have an associated payload.
`cap_get_payload(cap`$_c$`)`$\rightarrow$ `payload`	Retrieve payload of given capability.
`cap_revoked(cap`$_c$`)`	Mark capability cap_c revoked to reuse its revocation counter in the future.

Table 7.3: Capability-management operations in the DomOS kernel.

stack to ensure return addresses are always correct. This is achieved by setting up a synchronous CODOMs capability for each thread's stack. It is worth noting that architectures that store the return address in a register, like the jump-and-link instructions found in ARM or MIPS, make this much easier to enforce because the return address register cannot be concurrently modified by a third party.

7.2.3 Capability Management

Synchronous capabilities, which are the vast majority, can be directly created by user code. Nonetheless, DomOS provides the operations shown in Table 7.3 to interact with the privileged revocation counter fields. These can be used to obtain asynchronous capabilities, or capabilities that can be used as opaque pointers to user-provided addresses (see Sections 7.5 and 7.6).

If non-zero *async* or *payload* arguments are provided to `cap_create`, DomOS gets the address of an unused revocation counter and sets the revocation counter address to point to it. If there were no free counters, a new one is allocated. In the case of *async*, the revocation counter value field is set to that stored in memory (with an initial value of one, since zero implies asynchronicity is disabled). A revocation counter occupies 128 bits; 64 bits for the revocation counter value (ignoring the upper part, since the value field has 46 bits), and 64 bits for the payload. Therefore, when *payload* is provided by the user, it is stored on these last 64 bits; its usage is described in Sections 7.5 and 7.6. The rest of the fields in a capability are filled with the provided arguments, except for the capability's tag, which is set to that of the requesting domain (obtained through the `prevdom` register). This payload can then be retrieved with `cap_get_payload`, as long as the caller domain's tag (`prevdom`) matches the one in the capability. Finally, DomOS reclaims revocation counters when a domain is destroyed (after revoking all asynchronous capabilities created for it), and when `cap_revoked` is called.

The unsupervised CODOMs instructions to create capabilities set the tag field to that of the currently executing domain. Therefore, DomOS also provides variants of the `cap_create` operation that can be used to create a capability to a specific domain, as long as the requestor has access to that domain or provides a software domain capability to it. This functionality is provided to support the design pattern described in *Sparsity* under *CODOMs* in Section 4.2.1. For example, assume domain A allocates its dynamic data structure in domain B; since domain B might not contain code that can be used to create a capability to it, domain A can instead use `cap_create` to create a capability (not necessarily asynchronous nor with a payload) to grant access to domain B (i.e., domain A creates a capability to the allocation pool domain B).

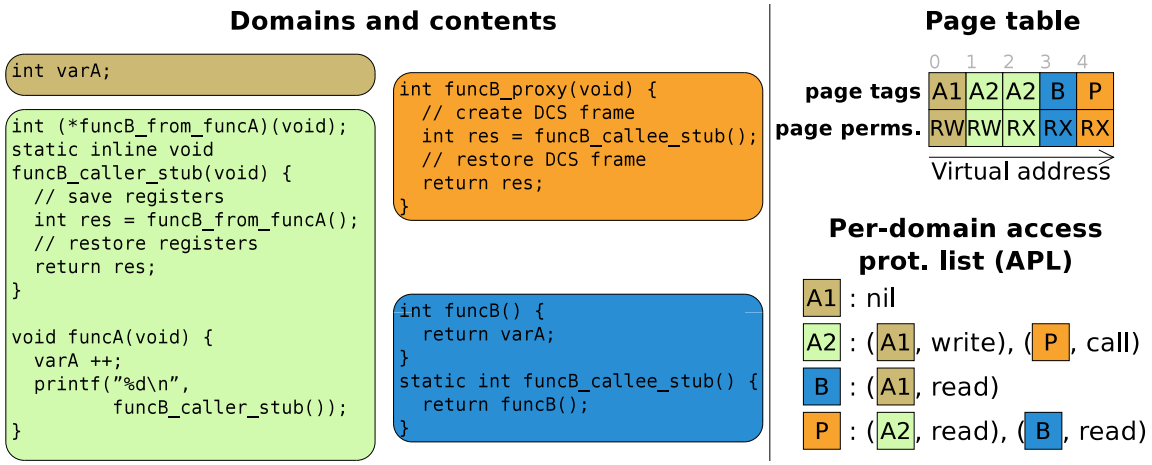


Figure 7.1: Domain contents and CODOMs configuration of the example application of Figure 6.1.

7.3 Runtime Support

7.3.1 Program Loading

The program loader in DomOS takes the per-domain information described in Chapter 6 to automatically create domains, assign the appropriate pages to them, and configure their APLs. Since the information refers to domains on the same application, the untrusted program loader can safely assume the information is correct.

This can be seen in Figure 7.1, which shows the layout and configuration for the domains generated by the compiler annotations in Figure 6.1. The program loader assigns each function and variable to its corresponding domain (using the `dom_create` and `dom_mmap` operations from Table 7.1). In this case it creates domains *A1*, *A2* and *B*, shown in the top-right of Figure 7.1. Note that domains *A2* and *B* also contain the caller and callee stubs `funcB_caller_stub` and `funcB_callee_stub`, respectively. The loader then reads the domain permission information from the binary (Lines 4, 5 and 18 of Figure 6.1b) and uses it to configure the APLs (bottom-right of Figure 7.1) using operations `dom_copy` (for the read-only grant from *B* to *A1*) and `grant_create` from Table 7.1.

Shared Libraries

An unannotated shared library can be used by multiple domains inside the same application. In this case, the library's state should pertain to the domain that calls it, not to the shared library itself. Note that this is different than having a shared library act as a separate domain, in which case no special action is necessary. There are various ways to handle the sharing case:

- The simplest approach is to map the library multiple times, once for each domain that uses it. This makes the domain have a private copy of the library embedded into it, while code and read-only data for the library shares the same physical memory and cache lines.
- An alternative that is also evaluated instead, places a single copy of the library on its own domain. In this case, constructor and destructor functions and writable sections are forbidden, since the domain

where the library is loaded does not have state of its own by definition. In exchange, callers must pass a capability to the library, which is used to access its dynamically allocated memory on a per-caller basis. This further reduces TLB pressure, since virtual addresses are not duplicated.

It is important to note that this second approach is key to mutually isolating threads of the same application, even when they are executing the same code. This feature is essential to efficiently enforcing critical data security policies (see Section 8.2.2 and Appendix A).

7.3.2 Entry Point Resolution

The same logical entry point can be accessed through different addresses depending on the isolation properties that the caller and callee specify. Therefore, entry points are resolved lazily, like other dynamic symbols through the PLT. This effectively provides a form of run-time entry point versioning.

The first time an entry point is called, the program loader checks if the target is for a domain defined in the same process. In such case, if the caller and callee isolation properties do not require a proxy, the call is resolved into the corresponding callee stub. If a proxy is necessary, the loader creates an entry point capability for the entry, requests a proxy for it, and grants the caller access to it.

This can be seen in the logical call from `funcA` to `funcB` in Figure 7.1. The caller stub (`funcB_caller_stub`) calls into an undefined function that the runtime must resolve (`funcB_from_funcA`). Now, the loader sees that the symbol refers to an entry point of domain *B* (Lines 9 and 10 of Figure 7.1). In turn, the loader checks the information of the `funcB` entry point. Since the call originates in domain *A2*, the loader will use the callee stub `funcB_callee_stub`. First, it will register the entry point (`entry_register` operation in Table 7.2) with the signature information (Lines 15 and 16) and the callee-side of the policy (Lines 19 and 20). Then, it will invoke an `entry_request` operation with the caller-side signature information (Lines 9 and 10) and the policy (none in this example, since it is all implemented at user-level through the caller stub). This results in a domain capability for the new domain with the requested proxy (function `funcB_proxy` in domain *P*). Finally, the loader creates a *call*-only domain capability to *P* (using `dom_copy`) and grants domain *A2* access to it (using `grant_create`).

To minimize overheads, entries are registered and requested in batches. Therefore, resolution of entries for the same domain do not need to request new proxies, but merely point to them. Note that it is possible to eagerly resolve entry points at compile time when no such proxies are involved (therefore saving a jump instruction, equivalent to a PLT entry indirection).

Cross-Process Entry Points

For a remote process call, the loader asks the callee process for the corresponding entry point capability (remember that it can be passed across processes using POSIX file descriptors). Deciding whether to return an entry point capability, and what isolation actions is should have requires application-specific knowledge. On one hand, the caller process must be able to uniquely identify the appropriate callee process; e.g., one cannot rely on some global naming service to identify processes because processes could otherwise mischievously register themselves as being someone else. On the other hand, the callee must be able to authenticate the caller process to decide on its policy; e.g., today's MySQL clients start by providing some user credentials to the database, which athorizes the connection properties based on that information.

The DomOS loader provides the base mechanisms to make this possible; applications can simply register callbacks that return the entry point capability given the target domain and entry point information. This lets applications implement their own authorization scheme for cross-process entry points.

Currently, a simple helper library is also provided to automate this process in a way similar to what other RPC systems do. Remote entry points are identified through a random binary string (i.e., a password or sparse capability), that must be known to both sides. Programmers can either provide a constant value to both sides, or implement some application-specific protocol to exchange them. When a callee process starts, it automatically maps the identifiers of its entry points to the actual entry point information (function address, signature and callee-side policy). In turn, caller processes register the remote entry point identifiers with the cross-domain symbol that the loader must resolve (`funcB_from_funcA` in the example above). Finally, caller processes provide a default callback to the loader that uses that information to automatically resolve the remote entry points.

Note that alternative more complex protocols can also be implemented on top of the support provided by the loader. DomOS can provide a trusted user-level service that serves as a global entry point request broker. Since there must be a way to authorize entry point registrations and requests, the entry point service can use a mix of sparse and software capabilities. Sparse (or password) capabilities can be used to request entry points using an unguessable "password" provided as out-of-band information (e.g., as a command line argument). Software capabilities can be used to pass access to entry points across a hierarchy of processes (using file descriptors). For example, a parent process opens a connection to the entry point service to register new entry points. From that connection, it then retrieves a new one that allows requesting access to any entry point registered through the first connection. Now, the parent process can create two children, the caller and callee processes, passing the first connection to the callee and the second connection to the caller. This effectively allows the caller and callee processes to resolve the cross-process entry point without resorting to passing "passwords" through the command line or through configuration files. In fact, this scheme shows the most typical way to organize services in systems that use software capabilities as references to objects implemented on other processes (usually termed object-capability systems [105]).

7.4 Implementation Details

7.4.1 Unified Virtual Address Space

The SASOS implementation in DomOS uses a two-level allocation algorithm. Every memory allocation starts by requesting a large fixed-size block of *virtual memory space* (currently 1 GB, minimizing the use of page tags). Actual memory is then sub-allocated from virtual memory blocks owned by the requesting process. Since memory for different domains uses different tags and addresses are non-overlapping, multiple processes can safely use the same page table. The current implementation does not use any sophisticated locking scheme, nor implements the optimal reuse algorithm of holes during sub-allocation of virtual memory space blocks. Therefore, the current implementation is expected to provide suboptimal performance due to locking contention.

TLB coherence domains

DomOS provides a unified (or global) virtual address for all processes, and uses CODOMs to isolate them on this address space. Therefore, DomOS can use a single page table for all processes in the system (see Section 7.1). This is key to provide efficient cross-domain calls, since now there is no need to switch between different page tables (an expensive operation) during critical cross-process communication operations.

Nonetheless, sharing a page table across processes accentuates the concerns regarding TLB shootdown operations [121]. Linux keeps a bitmap on each page table that indicates which CPUs *might* currently be caching entries for that page table on their TLB. This is used to calculate the set of CPUs that must be

taken into account during a TLB shootdown operation, which translates into interrupting them using an IPI. Since all processes share a single page table in DomOS, the chances of affecting more CPUs during a TLB shootdown operation increases, even when processes are not communicating with each other. Nevertheless, this already is a well-known problem in multi-threaded workloads for existing systems, and it is thus reasonable to expect that future architectures will remedy TLB shootdowns using some form of TLB coherence mechanism [95, 121].

At the same time, DomOS can alleviate this problem without additional hardware. The SASOS implementation actually allows creating multiple of these unified (or global) virtual address spaces, which are called "environments". Each SASOS environment has its own page table, and therefore acts as a separate "TLB coherence domain". Each process can be part of one, and only one, environment, and each environment can hold multiple processes. Therefore, processes that communicate frequently can be grouped into an environment, while other processes can be put on separate environments to minimize the span of TLB shootdown operations. Unfortunately, this implementation implies that different environments can never be "merged" in the future (e.g., the system detects they communicate very frequently) because they might have overlapping addresses.

Analyzing the performance and optimizing the performance of this palliative technique is out of the scope of this thesis. Nonetheless, a production implementation would ideally provide the unified (or global) virtual address space orthogonally to page table sharing. With this approach, the OS could dynamically adapt which groups of processes share a page table inside the same SASOS environment in a way that minimizes the span of TLB shootdown operations (i.e., spreading processes across multiple page tables on the same environment) while keeping frequently communicating processes on the same page table.

7.4.2 Thread Management

Since thread management is not critical to the measurements of this thesis, a very simple approach is used to support cross-process threads that minimizes kernel changes. The first time a primary thread uses a proxy routine that crosses into a different process, it creates a worker thread for the callee process. The proxy issues a kernel upcall to the DomOS runtime, which creates the worker thread. Subsequent process crossings from the same primary thread refer to the same worker thread, keeping a stable mapping from the logical (primary) thread to the actual thread identifiers in the system. These worker threads stay blocked in the kernel, and most of their storage is thus not necessary; this is similar to Mach's activations [49], which consume less resources than a full-blown thread object.

7.4.3 Thread-Local Storage

The ELF format defines Thread-Local Storage (TLS) as a per-thread array of pointers, indexed by a variable identifier [41]. In order to support dynamic module (code) loading, TLS storage is lazily allocated the first time a thread accesses a TLS variable of another module. Since threads in DomOS can cross process boundaries, domains can be treated like dynamically loaded modules that allocate TLS space on first use. To avoid modifying `libc` and the TLS-specific ABI to implement this approach, the current prototype instead changes the active TLS array base in proxy routines that cross process boundaries to that of the target worker thread (`fs` segment in x86–64).

7.4.4 Cross-Process Proxies

Conceptually, cross-process proxies implement an immediate process switch with time-slice donation. The *hot path* (the most common) is highly optimized to provide efficient cross-process calls. It gets the index of the target domain's tag on the per-CPU APL cache using the `aplcache_index` instruction (see Section 5.4); note that the target tag is known as an immediate constant when generating the proxy. This index is then used to access a small cache array on the primary thread's task structure. The proxy first checks if the entry is caching the requested target domain's tag (otherwise goes to the warm path), and then uses that entry to retrieve the information for the target process. The *warm path* searches for the target worker thread, indexed by the target domain's tag, and stores it in the aforementioned cache array. If the target worker thread does not exist (*cold path*), it asks the target process to create a new one for the requesting primary thread. A similar approach is also applied to lookup for target stacks.

DomOS also has a small Kernel Control Stack (KCS) for each primary thread, which links cross-domain call information together. Any value that must be restored by the proxy upon return is stored on the KCS, which is not accessible by regular user code.

7.4.5 Fault Notification

Fault handling in cross-process calls can be made to provide the same guarantees of existing systems; i.e., a failed callee process can be terminated and an error returned to its caller. For example, assume a scenario with three domains (A, B and C), and four numbered threads with the following call chain: *(1)* $\rightarrow A$; *(2, 3)* $\rightarrow A \rightarrow B$; and *(4)* $\rightarrow A \rightarrow B \rightarrow C$. If an unhandled fault happens in thread *(2)*, while executing code in domain B, the process that contains B will be terminated, and a fault raised to its immediate caller domain, A. Likewise, all other threads executing inside the B domain at that moment, thread *(3)*, will raise a process termination exception to their callers (domain A). Thread *(4)* shows the most interesting case. It cannot be immediately terminated, since the computation in C must rightfully continue. When the thread returns from C, since B has been terminated, an exception will be raised to the previous caller, A.

Therefore, a fault on a domain invokes a signal handler on it. If the fault is not recovered by the domain, the process might be terminated and a signal raised on its caller, which is identified through the information stored on the KCS by a cross-process proxy. By overriding the default handler with a user-provided callback (like a regular signal), a per-thread variable can be implemented to indicate when cross-domain calls fail. Therefore, after a call, one can read the variable to check if a cross-process call failed. This allows checking for errors without changing the signature of the entry point. If changing the signature were possible, fault notification could be implemented on the trusted proxies, just like POSIX functions return an error value in most cases.

7.5 Capabilities as Opaque Handles to User-Defined Objects

Many cross-domain interfaces have the notion of stateful objects. This is very clear in object-oriented languages like C++, where the structure with the object's state is the first (implicit) argument of a method. Nonetheless, this is also found on every C interface that does not operate on some system-global state (e.g., file descriptors identify the internal file state on file-management operations).

The payload functionality described in Section 7.2.3 precisely supports this pattern in a very efficient manner. A domain can create some logic object and use the address of the data structure describing it as a payload for a capability; at the same time, the capability can grant access to the functions used to operate on that object. When a call to one of these methods is received, the caller can use the capability's payload

value as a pointer to the object, almost as if it received a pointer to it as an argument. The payload cannot be changed nor inspected by other domains, effectively turning such capabilities into protected and opaque object references. Note that this pattern is implemented in software, and not imposed by CODOMs. One could instead use a capability payload to identify the user-level object data structure, and use separate means to grant access to its methods (i.e., use the APL itself or a separate capability).

For example, a capability for a read-only version of an object could grant access to different methods than a capability for a writable version of the same object. This can be used to enforce in hardware the same properties that type qualifiers provide in, for example, C++ objects (e.g., non-`const` methods cannot be invoked on `const`-qualified objects).

In both cases, this approach is more efficient than what is currently implemented in other systems to name objects from other domains. For example, POSIX systems use an integer to identify open files, which are actually implemented by another domain (the OS kernel in the case of a monolithic system). In this case, the domain implementing the object must lookup the provided object identifier in a table to retrieve the actual data structure that describes it, making sure the identifier is looked up in the object space pertaining to the caller (e.g., each process has a different file descriptor table). Note that this problem not only applies to POSIX file descriptors, but to any software capability system, since capabilities must also be looked up by the OS kernel [122] (file descriptors are, in fact, software capabilities).

7.6 Unified Resource Access Controls

For time constraints, DomOS minimizes the amount of changes to a typical UNIX system organization (Linux, in this case). This means, among others, that software resources provided by the system are isolated at process granularity (e.g., the file descriptor table). This yields two different mechanisms to manage the authorization of different resources: memory (including application software abstractions) and system software resources.

As described above, capabilities with a *payload* can be created as opaque handles to software abstractions. This could be extended to the OS interfaces, such that domain grants and hardware capabilities could be used as the sole authorization mechanisms for remotely addressed resources. On one hand, domains would not need separate processes to have their own isolated set of system software resources. On the other hand, system services could use these opaque handles to directly point to their internal resource structures, eliminating the overheads of intermediate lookups.

Chapter 8

Evaluation

8.1 Hardware Mechanisms (CODOMs)

The architectural changes for CODOMs were first prototyped on a 32-bit x86 system using QEMU [20], together with minimal changes to the Linux kernel to expose the CODOMs features to user and kernel code. Evaluation then moved to the cycle-accurate GEM5 simulator [24] running in full-system out-of-order mode with version 2.6.27.62 of the Linux kernel (the kernel version was limited by the simulator version). The simulation parameters, listed in Table 8.1, mimic an Intel Nehalem processor, which was also natively used in some of the experiments.

8.1.1 Comparison of Different Cross-Domain Call Mechanisms

This section compares the performance of domain switching in CODOMs and other mechanisms (see Chapter 2) through a set of micro-benchmarks that call a procedure on a different domain 10 K times. Every benchmark tests a combination of mechanism, number of function arguments, and randomly generated caller and callee workloads. The parameters are shown in Table 8.2, and the the workload for each mechanism was generated using the same seed for a given parameter combination. Results are compared against a regular function call/return, taking the second of two repetitions. The mechanisms evaluated are:

Mondrix [124, 125]: Implicitly switches domains using call/return instructions, as implemented by the second MMP design [125]. The benchmark optimistically approximates the cost of a domain switch using an instruction barrier. It also optimistically ignores any other costs; namely, OS intervention and TLB-shootdown-like costs associated to memory access grants and revocations, misses on the PLB and GLB hardware cache structures, and any extra delays involved in the domain reconfiguration process implemented by the hardware (e.g., adjusting the stack-protecting registers; see Section 2.2.2).

Syscall: The overhead of using a system call, which implements the callee code. This is the base mechanism for user/kernel interaction.

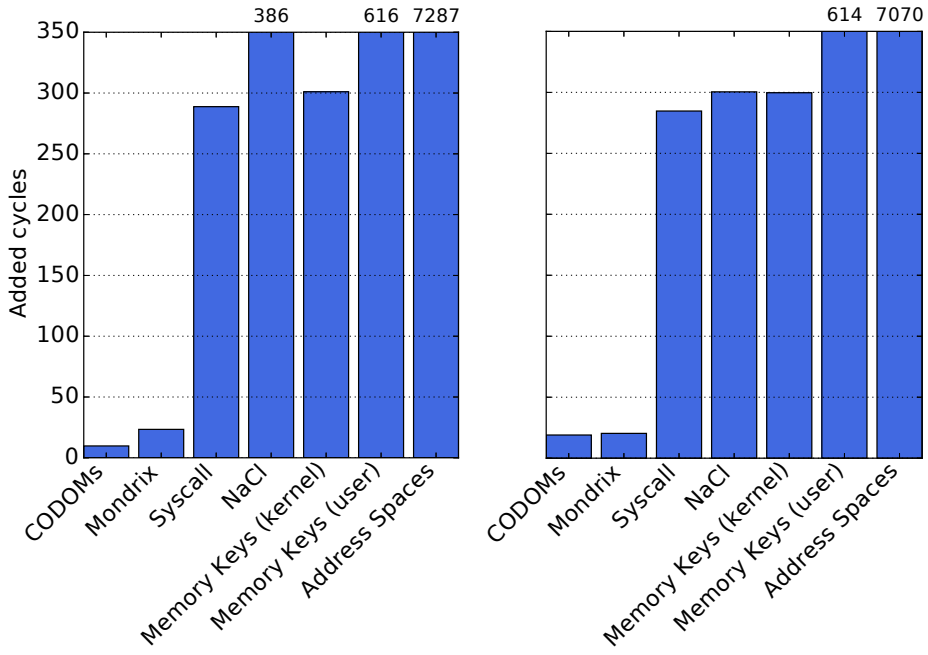
NaCl [127]: The callee switches segments at user-level before and after performing its workload, imitating a naively optimistic implementation of NaCl. This is implemented by pre-registering a segment descriptor for each of the segments that are modified: `cs`, `ds`, `es` and `gs`. Segment `fs` is not modified to avoid breaking the TLS support. Although it is not evaluated here, the same technique has also been applied at the kernel level [77].

Processor speed	2.4 GHz
Processor width	4 (insts in fetch; μops for the rest)
Register file	160 (int), 144 (float)
Load/Store/Inst. queue entries	48/32/36
ROB entries	128
i/d TLB	64 entries, 4-way
i/d Cache	32 KB, 8-way, 4 cycles
L2 cache	256 KB, 8-way, 7 cycles
L3 cache	6 MB, 12-way, 30 cycles
RAM latency	65 ns
Capability registers	8
APL cache entries	32
`currdom`/`prevdom` registers	6

Table 8.1: Configuration of the simulated architecture for CODOMs.

Parameter	**Values**
Number of arguments	0, 5, 10
Caller/callee workload insts.	0, 25, 50, 100, 1000
Workload distribution	60% integer / 20% read / 20% write

Table 8.2: Micro-benchmark execution parameters.



(a) Average of all workloads

(b) Average of empty-body workloads

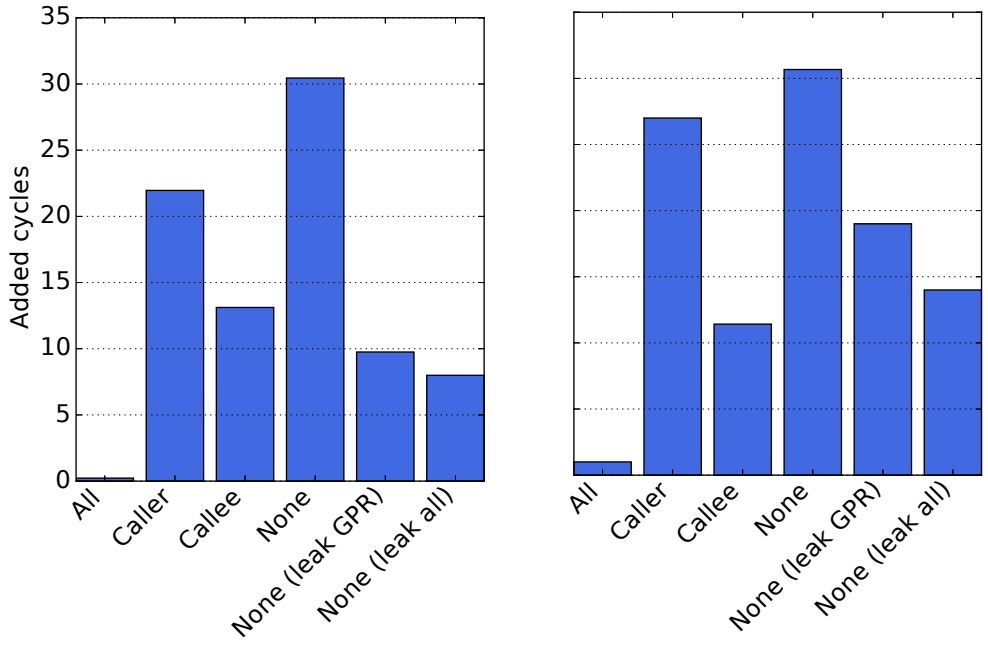
Figure 8.1: Comparison of domain switch overheads for CODOMs and other mechanisms. The overhead is depicted as additional cycles over a function call/return. "*Mondrix*" and "*Memory Keys*" are optimistically approximated by an instruction barrier. The "*Mondrix*" approximation does not simulate any other costs (e.g., grants and revocations). "*CODOMs*" uses the policy "*None (leak GPR)*" later shown in Figure 8.2.

Memory Keys (kernel) [55, 61, 67]: An approximation of a key-based memory protection switch, used to isolate kernel components. A system call implements the callee, switching keys before and after the call. Optimistically assumes that the cost of switching keys is equivalent to an instruction barrier.

Memory Keys (user) [55, 61, 67]: Like "*Memory Keys (kernel)*", but used to isolate user components. Before and after its workload, the callee invokes a system call that switches the keys. This is similar to the use of protection keys of DB2 running in IBM System *p* [36].

Address Spaces: The cost of switching address spaces by communicating data using a POSIX pipe. Web browsers such as Chrome use this to isolate untrusted plugins into separate processes.

Figure 8.1 depicts the overheads of these mechanisms as the number of cycles they add on top of the baseline function call. The figure shows that "*CODOMs*" and "*Mondrix*" provide the lowest overheads. The "*CODOMs*" results are for the "*None (leak GPR)*" policy shown in Figure 8.2, which is slightly more secure than the "*Mondrix*" policy. Note that CODOMs uses additional instructions to implement its policies; therefore, hoisting some of them into a single instruction could further reduce overheads. The results indicate that instruction barriers (used as an approximation for "*Mondrix*" and "*Memory keys*") have an order of magnitude larger overheads than CODOMs. This points to the importance of addressing pipeline stalls during domain switches, which can be very important for short isolated routines.



(a) Average of all workloads.

(b) Average of empty-body workloads.

Figure 8.2: Comparison of domain switch overheads for different isolation policies in CODOMs. The overhead is depicted as additional cycles over a function call/return.

Other mechanisms incur substantially higher overheads. All but "*CODOMs*", "*Mondrix*" and "*NaCl*" require intermediate system calls; i.e., switching between domains is a privileged operation. In addition, all mechanisms but "*CODOMs*" hinder pipeline throughput by introducing RAW hazards on a domain switch. Therefore, CODOMs is the only system that eliminates both sources of overheads, while at the same time is flexible enough to implement different policies on top.

8.1.2 Effect of Isolation Policies

Isolation policies define the overhead incurred by the proxy and caller/callee stub routines. The following configurations are evaluated[1]:

All: Both domains trust each other, meaning that resource integrity and confidentiality is not enforced by any party. In addition, the callee can access code in the caller, meaning that a return capability is not necessary. As a result, the proxy is equivalent to a resolved PLT entry.

This could be used, for example, to have two domains that can freely call each other, but whose allocations are placed on separate domains to avoid memory corruption across them.

[1] These experiments do not handle data stack confidentiality nor process switches, since they were developed before defining these properties during the development of DomOS.

Caller: The callee trusts the caller, but the caller preserves the integrity and confidentiality of its register state and integrity of its stack against the callee (e.g., kernel→module).

Callee: The trust relationship opposite to "*Caller*" (e.g., module→kernel).

None: Domains do not trust each other (≈ "*Caller*" + "*Callee*"). This can be used for mutually isolating any pair of domains.

None (leak GPR): Similar to "*None*", but general-purpose registers are not managed. Slightly more secure than Mondrix, which provides read access to the whole stack.

None (leak all): Similar to "*None*", but general-purpose and capability registers are not managed (e.g., guard against dangling pointers and stack smashing).

Figure 8.2 depicts the overhead of switches for the different isolation policies, measured as the number of cycles that each experiment adds on top of a regular function call. Note that Figure 8.2b shows the overhead of all empty-body workloads, which are the most sensitive to ILP reductions during the domain switch. The figure shows that "*All*" delivers the best performance by avoiding RAW hazards, and only incurs in the overhead of the jump in the proxy routine (cannot be appreciated in Figure 8.2a because some of the workloads perform operations before returning). In contrast, "*None*" shows the highest overhead since it implements the most restrictive policy. Still, its overhead is lower than that of other mechanisms. The rest of experiments show intermediate overheads whose main factors are the DCS frame management, the proxy's return address injection, and the safeguard of the caller's stack pointers (this last not present in "*Callee*"). These results show that separating mechanisms from policies allows tuning the performance to the desired isolation properties.

8.1.3 Area and Energy Overheads

Area and energy estimates for CODOMs were obtained with McPAT [75] using a 32nm process. The estimates show a 1.89% per-core area overhead, and Table 8.3 shows the average energy overheads of the micro-benchmarks compared to those of other mechanisms found in x86. Overheads for CODOMs are decomposed into the hardware structures and the execution of the additional policy-specific code. These results show that CODOMs energy overheads are practically negligible.

Benchmark	CODOMs	Code	Total
CODOMs: All	0.45	0.09	0.54
CODOMs: Callee	0.47	2.41	2.88
CODOMs: Caller	0.49	6.69	7.18
CODOMs: None	0.50	7.57	8.07
CODOMs: None (leak GPR)	0.48	3.54	4.02
CODOMs: None (leak all)	0.47	3.01	3.48
Address Spaces	-	-	1280.83
NaCl	-	-	40.75
Syscall	-	-	29.42

Table 8.3: Average energy overheads (%) relative to a function call/return. Includes the same CODOMs policy configurations shown in Figure 8.2, as well as other mechanisms available in x86.

Energy overheads are for the steady-state of the system when running micro-benchmarks that stress the CODOMs architecture. Therefore, they do not account for the additional energy overheads of the APL cache miss handler that is implemented by the OS kernel. Nonetheless, as Section 8.1.4 shows later, the number of APL cache misses is also negligible in macro-benchmarks.

8.1.4 Linux Kernel Module Isolation

The following experiments quantify the system-wide impact of CODOMs by considering all Linux kernel modules as separate domains. Two macro-benchmarks were measured: a parallel Linux kernel compilation, and *netperf* using the TCP bulk transfer test.

Since detailed full-system simulation is too slow, the macro-benchmark's timing was extrapolated by injecting the domain switch overheads (obtained by the micro-benchmarks) into the running time of a native execution. The number of domain switches was measured using a modified version of QEMU [20], considering every module as a separate domain by inspecting their load addresses.

Table 8.4 shows that modules typically perform short bursts of operations. Therefore, CODOMs is well suited to this environment since it provides unsupervised domain switching and access grant primitives with a low impact on ILP. Furthermore, more than 99.6% of the domain switches involve no more than 8 domains, ensuring that the APL cache hits for the vast majority of the time.

Domains		**Switches**	**Instructions**	
			→	←
Compile	kernel / ext2	6403400	1029	16
	kernel / scsi	1777834	200	19
	kernel / libata	1638960	360	26
	kernel / cfq-iosched	1187154	390	31
	kernel / unix	149170	234	13
	scsi / scsi-sd	105444	21	48
	libata / scsi	63270	111	13
	Others	114327	-	-
	Total	11439559	-	-
netperf	kernel / e1000	22098737	261	41
	Others	14048	-	-
	Total	22112785	-	-

Table 8.4: Number of domain switches (calls & returns) during the benchmarks' execution. The two rightmost columns show the arithmetic mean of instructions executed in a domain before switching into the other.

Table 8.5 depicts the modelled system slowdown. In all cases, the overheads are effectively negligible (less than 1%), although replacing system calls with CODOMs would actually improve system performance. Even though the Mondrix approximation provides similar raw performance, it still requires OS intervention and operations similar to a TLB shootdown for access grants and revocations.

Figure 8.3 shows the memory access distribution of the domains, according to the owner of that memory. Memory dynamically allocated through a pool is owned by the domain that created the pool, while other dynamically allocated memory is owned by the domain requesting the allocation. The vast majority of the non-stack accesses fall in one of two categories:

Isolation policy	Compile (%)	*netperf* (%)
None	0.10 ± 0.01	0.15 ± 0.02
None (leak-GPR)	0.03 ± 0.01	0.05 ± 0.02

Table 8.5: Runtime overheads incurred when considering each kernel module a separate domain.

- "*Self* *": Accesses to memory owned by the accessing domain. This type of accesses do not require using capabilities.
- "*Synch. **": Accesses to "remote" memory owned by another domain that is part of the current call chain; i.e., the owning domain is (indirectly) calling into the domain that performs the access. This type of access suggests that synchronous capabilities can be used to efficiently grant access, since it is possible to transitively pass a synchronous capability as an argument to all functions that lead from the owner domain to the domain that performs the memory access.

A third category, "*Asynch. **", identifies those accesses that should be performed through asynchronous capabilities, since the owner domain is not part of the call chain that led to the domain performing the access.

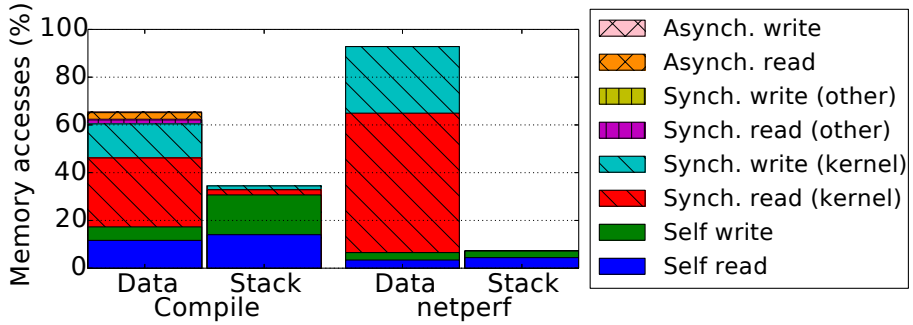


Figure 8.3: Distribution of memory accesses according to the owner domain of the accessed memory.

Of all the "*Synch. **" accesses, most go to the main kernel memory ("*Synch. * (kernel)*"). This is primarily due to structures allocated at its generic layers, which are later accessed by the device- or protocol-specific modules. A rewrite of the system could minimize these accesses, as could having the core *Kernel* domain accessible from all domains, while retaining inter-module isolation. Still, "remote" accesses are an intrinsic property of fine-grained isolation.

Importantly, most "remote" accesses can be handled with synchronous capabilities ("*Async., **" is small), showing the convenience of distinguishing between the different capability types in CODOMs.

8.2 System Performance (DomOS)

DomOS was also evaluated using a set of micro- and macro-benchmarks. The micro-benchmarks provide a quantitative comparison of different existing IPC mechanisms and DomOS, whereas the macro-benchmarks explore the performance improvements on applications that use DomOS.

Board	Dell PowerEdge R210 II
Processor	Intel Xeon E3–1220 V2 @ 3.10GHz, 4 cores
Memory	16 GB
Ethernet	Broadcom NetXtreme II BCM5716 (1Gig)
Infiniband	Mellanox ConnectX MT26428 (10GigE)

Table 8.6: Configuration of the evaluation machines for DomOS.

DomOS was prototyped on top of version 3.9.10 of the Linux kernel. Initially, the option of using open microkernel OSs (e.g., L4Linux, Minix and GNU/Hurd) was explored, but none of the available microkernels supported both a 64-bit address space (necessary to fit all processes in a single virtual address space) and all the evaluated benchmarks. The system was natively evaluated using a Dell PowerEdge R210 server, which is described in Table 8.6. This allowed executing the applications and OS in their entirety and not be limited by simulation time. Naturally, the evaluation machine does not provide the CODOMs hardware, which was instead emulated in software. Every per-CPU resource in CODOMs was emulated using per-CPU variables, and handled in the OS kernel as part of the CPU state (variables are accessed through the `gs` segment, which Linux x86–64 already uses for that purpose). Hardware capability instructions were emulated using regular loads and stores. Nonetheless, argument capabilities were assumed to be handled by the compiler. Processes share a single page table (SASOS), but of course the processor does not enforce CODOMs' APLs. The privileged capability bit in CODOMs was emulated by running all code in privileged mode using Kernel Mode Linux (KML) [83]. Note that system calls still go through Linux' standard `syscall`/`sysret` path in this prototype. Changing this could provide interesting benefits, but is not in the scope of this evaluation.

This emulation approach allows evaluating the broader effects of the DomOS design, but the hardware does not actually enforce protection. Furthermore, using memory variables is slower than a hardware implementation of CODOMs. Therefore, the described emulation approach provides a reasonable approximation of the underlying hardware.

8.2.1 Comparison of Different Domain Communication Primitives

As observed in Chapter 3 and Figure 3.1, faster hardware isolation mechanisms are necessary but not sufficient to improve IPC. Process synchronization ("*Semaphore (=CPU)*") adds very high overheads, and cross-core IPIs make the overheads even higher ("*Semaphore (≠CPU)*"). Overheads are further exacerbated when performing transfers and hiding the complex IPC interfaces under synchronous function call semantics ("*RPC*" experiments). Therefore, this section quantifies the effectiveness of eliminating these overheads in DomOS.

Figure 8.4 shows the overhead of IPC using DomOS, compared the other mechanisms shown in Figure 3.1 (and using the same methodology). Additionally, Figure 8.5 shows where time is spent on the DomOS experiments; each line shows the time spent on each of the layers involved in a cross-domain call, and the total execution time is obtained as the concatenation of these lines.

The DomOS experiments have two main sets of variants. The *High* variant ("*DomOS - High (=CPU)*") shows the performance of full isolation inside the same process: symmetric integrity and confidentiality (i.e., all elements shown in Figure 8.5, except *Switch process* and *Switch TLS*). The *Low* variant ("*DomOS - Low (=CPU)*") shows the performance of minimal isolation inside the same process, where the base proxy code only injects its return address and caller/callee stubs are empty (i.e., only the *Function*, *PLT indirection* and *Base* elements shown in Figure 8.5). The *High* variant is only slightly slower than "*Syscall*", but unlike

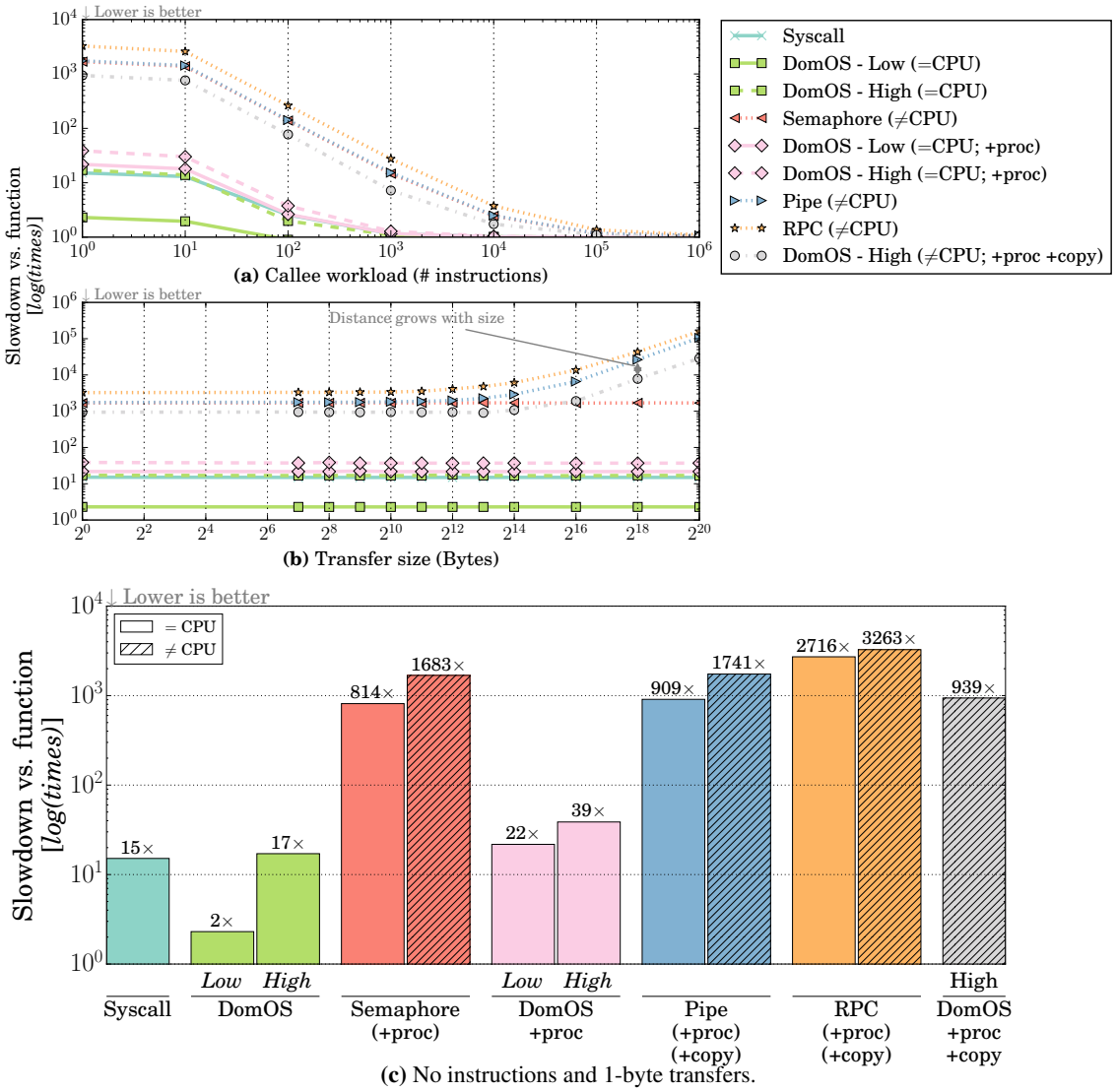


Figure 8.4: Performance of cross-domain calls using DomOS for different caller/callee isolation policies. Note that transfers in DomOS are not necessary, but are shown for comparative purposes. Also shown are other mechanisms from Figure 3.1.

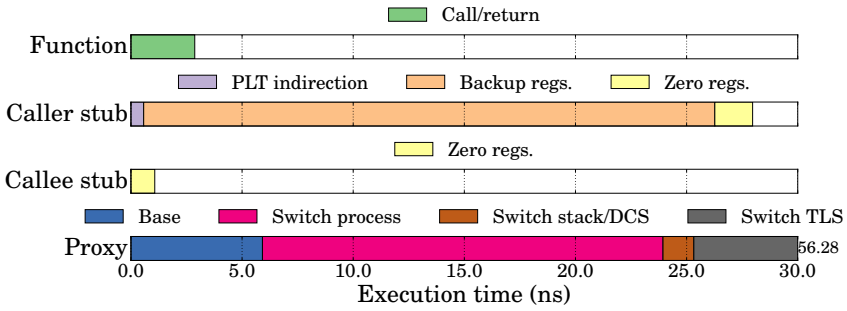


Figure 8.5: Execution time breakdown of a cross-domain call/return in DomOS. There is one line for each layer involved, where *Function* corresponds to the baseline regular function call/return. The total execution time corresponds to the concatenation of all lines.

privilege levels it is not limited to implementing hierarchical isolation. The *Low* variant is even 7.43× faster than the *High* one, mainly due to the caller saving a copy of its registers in *High*. Therefore, optimizing isolation policies yields promising performance benefits. Furthermore, a system using DomOS is simpler to program, since in the "*Syscall*" case the kernel has to explicitly handle accesses to user-level data (e.g., passed as references on the argument list) using special-purpose functions.

The variants with *+proc* show the performance of cross-process calls in DomOS, providing semantics similar to those of RPC at a fraction of the overheads. The DomOS results are substantially better than existing mechanisms, around 2 orders of magnitude faster than semaphores, pipes and RPC. Importantly, DomOS is much less complex for the programmer, since she does not need to manage the synchronization of threads on each process, nor the (de)marshaling of cross-process arguments (the same applies to RPC). Furthermore, in the worst case DomOS is only 2.55× slower than a system call, but without any of its limitations. Isolation policies make the *High+proc* variant only 1.78× slower than *Low+proc*, since both have to perform a costly TLS segment switch. A domain-aware TLS implementation in `libc` (see Section 7.4.3) would thus improve performance between 1.54× and 3.22×, depending on the isolation policy.

Finally, "*DomOS - High (≠CPU; +proc +copy)*" shows the performance of providing the exact semantics of a synchronous RPC across CPUs; that is, caller and callee execute on separate threads (and different CPUs), and argument immutability is implemented at user-level (i.e., copies, even if they are not strictly necessary with CODOMs' capabilities). The callee uses semaphores to synchronize with the thread that performs the workload on a different CPU, and then relinquishes control to its caller. Interestingly, this approach could be used to transparently implement the semantics of all other IPC mechanisms using DomOS (e.g., using library interposition). DomOS performs 3.82× better than the semantically equivalent RPC version. Importantly, performance could be easily improved by using primitives optimized for synchronous control transfers, like those found in L4's IPC, since semaphores are not optimized for this use case. Note that this experiment also performs better than "*Semaphore*", since changing between privilege levels in the original Linux implementation is more expensive. More importantly, this approach allows significant simplifications on critical synchronization code in the OS kernel, since complex cross-process memory transfers can now be implemented at user level. As an interesting side effect, user-level cross-process copies in DomOS perform better as the buffer size increases (hard to see on the logarithmic scale of Figure 8.4b). This is because kernel-level transfers must ensure pages are mapped in its address space before performing process-to-process copies.

In summary, DomOS provides leaner and efficient communication primitives, which do not require pro-

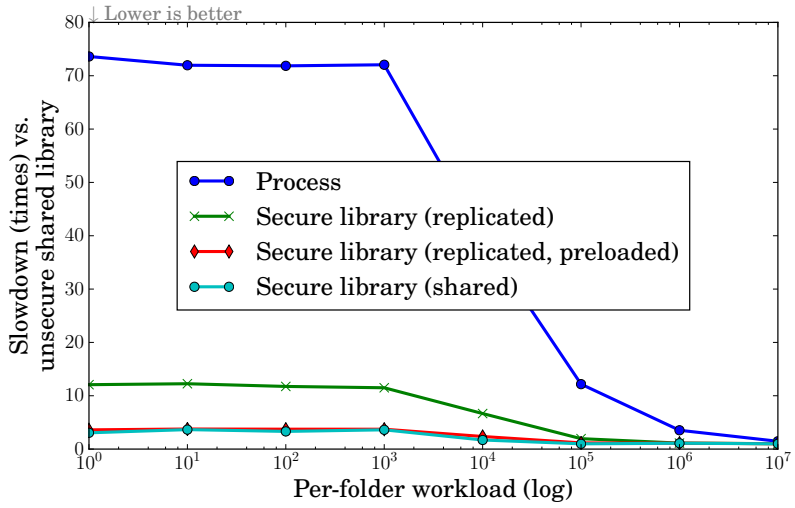


Figure 8.6: Performance of isolating computations based on the data they operate with. Results are shown for a total of 800 folders, 100 per core, normalized to the performance of an unsecured shared library. The X-axis shows the amount of work necessary to process each folder.

Processor	Intel Core i7-4770 @ 3.40GHz, 4 cores, 2-way SMT
Memory	12 GB DDR3
Disk	Seagate ST3500413AS, 500GB, 7200 rpm, 16MB cache
Network	Intel 82574L, 1Gbit

Table 8.7: Configuration of the evaluation machine for the thread isolation experiments.

cess synchronization, the RPC mapping of function calls into traditional IPC, nor cross-domain data copies. The customizability of isolation policies in DomOS also yields an extra performance boost. Finally, DomOS could transparently replace IPC in existing applications with a substantial performance improvement, freeing the kernel from the complex burden of managing cross-process copies.

8.2.2 Thread Isolation

This section explores the performance of mutually isolating threads that run the same computation on different data. A complete use-case and framework exploiting such organization is described in more detail in Appendix A. The reason for isolating threads of the same application is to enforce system-wide policies that limit the flow of information across different units of the same category (termed "folders" in Appendix A). This can be used to safely analyze sensitive information (e.g., medical records) with untrusted third-party code. This design is in stark contrast to the conventional approach of isolating computations based on which software component they execute (e.g., different users or processes), not based on what data they process. Different threads executing the same component must be isolated from each other when processing information from different folders. Therefore, each folder must be processed on a separate isolation domain, regardless of what code is used to process it.

Figure 8.6 shows the performance of isolating the untrusted code using different techniques when processing the information of 800 different folders. This experiment was run on a different machine (described in Table 8.7), and at each point in time 8 worker threads or processes were running, each processing 100 folders iteratively. For each folder, the child opens a folder-specific file and computes the factorial of its contents. The different values stored on files are shown across the X-axis to calculate how much workload is necessary to make the performance of isolation mechanisms irrelevant. In all cases, a different SELinux label is assigned to each folder to prohibit information to flow between them. The main application dispatches the processing of each folder to one of its children, and aggregates the results of computing the workload on each folder. The evaluation used the following techniques:

Unsecure shared library: A per-core thread uses the code of a regular shared library to process the per-folder files. This provides the best-performing baseline system (with no isolation).

Process: Isolation using a new process per folder on an existing system. The main application sets a per-folder SELinux label for the process (by invoking `setexeccon` before `execve`), ensuring information cannot flow across folders nor processes. This setup shows the most efficient container implementation (for executing code) of the ones described in Appendix A.

Secure library (replicated): The per-folder files are processed using the code of a shared library that is isolated using DomOS. The main application reuses the same per-core threads (8 in total) to process all folders. Therefore, a new replica of the isolated shared library is loaded for each folder, as an example of the first approach to executing shared libraries described in Section 7.3.1. Each replica is loaded on a separate domain and uses a fresh per-thread stack, which is isolated from that of other threads. To isolate threads at the OS level as well, the Linux kernel was modified to add an *iolated mode* of execution for threads. This execution mode makes the file descriptor table and the current SELinux label a private per-thread structure, effectively isolating threads of the same process. Changing the execution mode is a privileged OS operation executed by the proxy routine that bridges the main application with the isolated execution of the workload.

Secure library (replicated, preloaded): Shows a variant of "*Shared library (replicated)*" where all library replicas are preloaded into their respective domains before timing the experiment. This is not a reasonable design for a production setup, since the number of folders (i.e., domains) is potentially unbounded. Nonetheless, it factors out of the experiment results the cost of loading the secure library (`dlopen`) every time a thread enters isolated mode.

Secure library (shared): The same library replica is executed by all threads, as an example of the second approach described in Section 7.3.1. A single copy of the workload library is shared among all domains, but it only contains code and read-only data. Per-domain stack and heap space are separately allocated on their respective domains by the main application before entering isolated mode on each of the threads, and are deallocated after executing the per-folder workload. Access to them is passed to the isolated thread through a capability created by the main application.

Even if the "*Process*" experiment provides the best performance in existing systems, its slowdowns range from more than $70\times$ to 1.47x with increasing workloads. The slowdowns for "*Secure library (replicated)*" range from 12.25x to 1.01x, demonstrably better than using existing process isolation. Nonetheless, "*Secure library (shared)*" reduces slowdowns to 3.65x–0.98x, since only dynamically allocated memory and the isolated execution mode needs to be managed by the main application across folders. Finally, the "*Secure library (replicated, preloaded)*" experiment shows slowdowns lower than "*Secure library (replicated)*", in

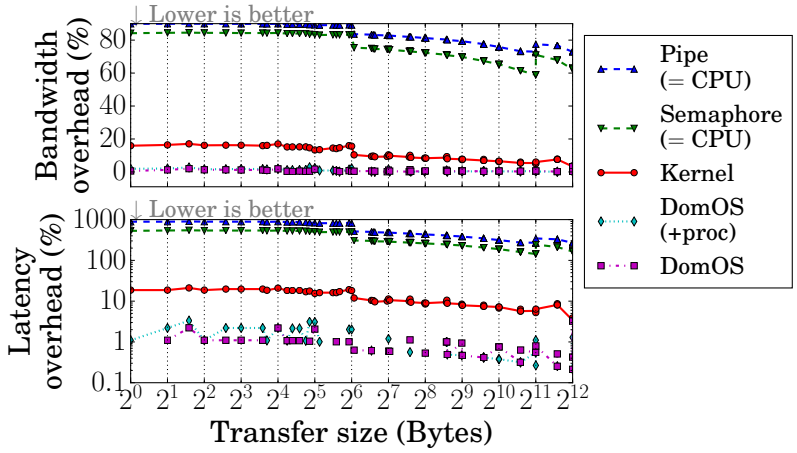


Figure 8.7: Bandwidth and latency overheads when isolating the Infiniband driver in three scenarios: DomOS, user-level processes (IPC) and a kernel driver (user/kernel isolation).

the range of 3.76x–1.01x. Unlike "*Secure library (shared)*", using library replicas does not allow an efficient sharing of micro-architectural state. These inefficiencies were already observed under the concept of "multi-execution" [25], which adds additional hardware to consolidate the resources used by multiple processes executing the same code (e.g., in server environments). Nonetheless, performance-critical code like this can be ported to the model offered by "*Secure library (shared)*" instead of adding further hardware support.

8.2.3 High-Performance Device Driver Isolation

Infiniband comes with extra hardware complexity to bypass the OS kernel overheads for its critical operations. To gain this performance, the OS must relinquish its control over many of the user's operations with Infiniband. The user is given a buffer shared with the NIC, and a user-level driver library directly operates the hardware. To ensure security, each user is given a randomly-generated stream identifier that the hardware tracks (i.e., a password or sparse capability). The current Infiniband implementation thus serves as an upper-bound performance scenario, albeit at the cost of additional hardware complexity and less OS control on the policies applied to the communications (e.g., fairness). Figure 8.7 shows the latency and bandwidth overheads of isolating the user-level driver in its own domain, where the OS could implement its policies and the Infiniband NIC would not need to provide the extra protection hardware.

Results are shown for the *netpipe* benchmark (`NPtcp`), mapped into Infiniband through the `rsocket` library. The actual driver still resides in the user application, but the Infiniband operations are interposed to make the corresponding synchronous requests to an isolated driver domain. The experiments provide results for the following configurations:

Pipe implements the driver as a separate process, interacting through a POSIX pipe.

Semaphore implements the driver as a separate process, interacting through a POSIX semaphore (`futex` in Linux).

Kernel implements the driver as a kernel module, accessible through an additional system call.

DomOS implements the driver as an isolated domain. In the "*DomOS (+proc)*" variant, the domain exists as a separate process, while the "*DomOS*" variant implements the domain as an isolated library inside the user application process.

In all cases, there are no additional copies between the application, the driver and the NIC, following the same model of shared transmission buffers found in the baseline Infiniband system.

Only DomOS can sustain the low latency characteristics of Infiniband, incurring only a ∼1% overhead. In comparison, system calls incur a 10% overhead but cannot isolate the driver from the kernel, and IPC mechanisms incur more than 100% latency overheads. Bandwidth is less affected, but still sees overheads higher than 60% for a 4 KB transfer in the IPC scenarios. Finally, the difference between the "*Pipe*" and "*Semaphore*" results show that unnecessary IPC semantics produce further slowdowns, since data copies are not needed in this case.

Unlike other IPC mechanisms, DomOS achieves low-latency isolation. This could be used in the future to regain OS control of transfer policies in cases like Infiniband, without adding complex security hardware in the NIC.

8.2.4 Multi-Tier Web Server Isolation

This section quantifies the system-wide performance benefits of DomOS using a dynamic web server macro-benchmark. The experiment uses Apache (version 2.4.16) to generate dynamic pages with PHP (version 5.6.0), that in turn retrieves data from a MySQL database (version 5.5.42). In addition, MySQL was configured to use a regular disk and an in-memory database (by storing its files in a `tmpfs` file system). The experiments also contain measurements to categorize where each core spends its execution time: user-level code, system call dispatch (assembly code in the `entry_64.S` file), thread scheduling and context switch, time spent in idle and, finally, any other time spent in the kernel. Existing performance analysis tools either add too much overhead or are not sufficiently accurate. Therefore, the Linux kernel was modified to perform these measurements with a negligible performance impact.

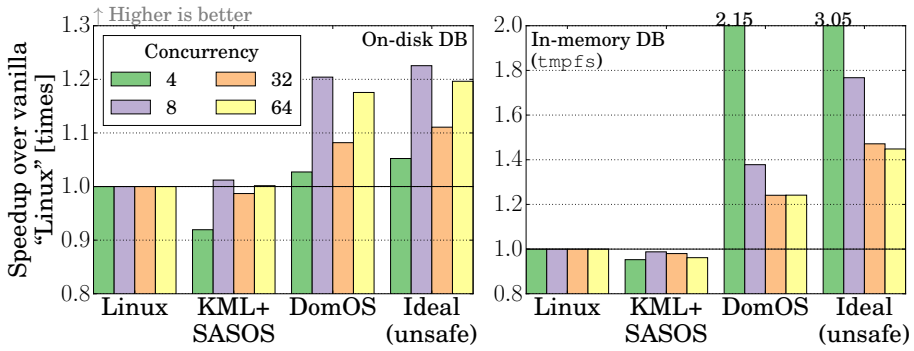
The setup was evaluated with the DVDStore benchmark [38] (version 2.1) with a 500 MB and a 1 GB input set, running for 2 min after a 1 min warmup period. All software components were executed with 4, 8, 32 and 64 threads each (or processes, depending on the case) in order to isolate the impact of server concurrency. Larger thread counts saturated the client application and the benchmark failed. Performance was compared with the following configurations:

Linux is the baseline, where all components run as isolated processes. Apache uses the multi-threaded *mpm-worker*, PHP worker processes run using FastCGI [87], and MySQL uses a separate threaded process.

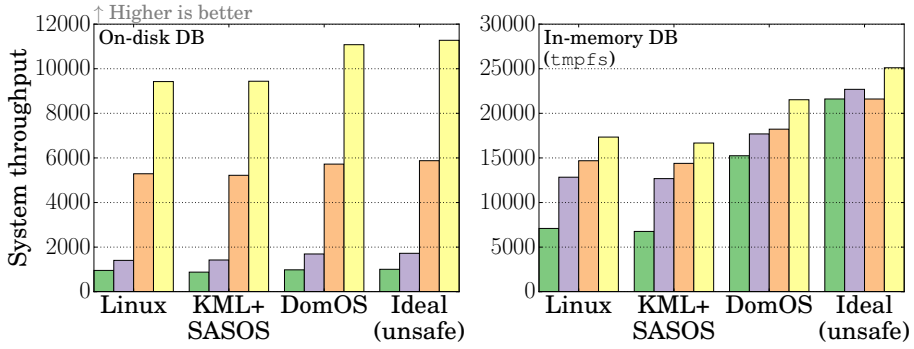
KML+SASOS shows the performance contributions of eliminating privilege level switches (although regular system calls are still used) and of sharing a global address space and page table across processes. Given that this setup does not use CODOMs, processes and the kernel are not isolated from each other.

DomOS places each component (Apache, PHP and libmysqld) on separate domains with an asymmetric isolation policy. Both Apache and MySQL do not trust PHP in any regard, but PHP completely trusts both. This includes the caller/callee stubs, as well as process-switching proxies.

Ideal (unsafe) is intended to show the ideal performance if all cross-process communication costs were eliminated. This configuration runs on the baseline *Linux* system, but embeds all components in a single process. PHP is used as an Apache plugin, and MySQL is embedded into PHP using the



(a) System speedup



(b) Absolute system throughput

Figure 8.8: Performance of different dynamic web server configurations using vanilla Linux and DomOS. To isolate communication and performance factors, the benchmark was run with 4, 8, 32 and 64 threads (1, 2, 8, 16 threads/core).

libmysqld library. The core implementation of each component is identical to that of the baseline, but without component isolation they are stripped from the unnecessary concurrency across processes and the glue code needed to manage IPC.

It is important to note that the baseline was configured in a way that minimizes the speedups of DomOS. On one hand, the setup does not use a separate bytecode cache for the PHP interpreter. Doing so would make the PHP component more IPC-bound (less computation time with the same amount of IPC calls), therefore making DomOS look faster in comparison. On the other hand, the baseline uses FastCGI instead of plain CGI; the latter spawns a new process for each request, while FastCGI (commonly used for performance) dispatches requests to multiple long-lived processes. Therefore, using CGI instead of FastCGI would provide higher benefits for DomOS since creating a process for each request is more expensive than communicating with an existing process.

Figure 8.8 shows the performance achieved by each configuration on the 500 MB database. Since the experimental compiler support cannot reliably auto-generate caller and callee stubs for all cases (see Section 6.2), their code was folded into the proxy routines. The performance difference of this approximation is below 2% of the equivalent experiments in Section 8.2.1, but shows the most conservative approximation by

assuming the worst-case function signature; i.e., all registers need to be saved and zeroed. Also, register and stack usage is unknown without compiler support. Therefore, stack switch costs are measured by performing a switch to the same stack without copying any of its arguments.

"*Ideal (unsafe)*" shows that IPC imposes large performance overheads (which are eliminated in this configuration), with speedups of up to 1.22× and 3.05× for the on-disk and in-memory versions, respectively. It eliminates around 70%–20% of kernel time, and around 20%–10% of user time (for 4 and 64 threads, respectively). Likewise, syscall dispatch and context switch times are reduced by around 50%. Most interestingly, concurrency increases performance of all configurations with an on-disk database. In comparison, the in-memory version of "*Ideal (unsafe)*" achieves close to maximum throughput with only 4 threads (1 thread per core; see Figure 8.8b): the in-memory "*Ideal (unsafe)*" is between 20×–2× faster compared to on-disk "*Linux*" (for 4 and 64 threads, respectively). This is because the on-disk version spends most of the time waiting for I/O, masking the overall improvements of "*Ideal (unsafe)*", while the in-memory database version removes most of that I/O-boundness.

The "*KML+SASOS*" experiment (on which DomOS is later built on) shows a general decrease in performance. This is contrary to intuition, since user/kernel transitions are faster when the privilege level does not change (especially in interrupts and exceptions) and context switches do not need to change between different page tables. This behaviour is explained by two main reasons. First, the memory management algorithms for the current SASOS implementation are largely unoptimized for concurrency, leading to larger contention. Second, as the number of threads increases there are more chances that TLB shootdowns will affect other cores, since they all use the same page table. Nevertheless, it is reasonable to expect that future architectures will remedy TLB shootdowns using some form of TLB coherence mechanism [95, 121]. Furthermore, the problem can also be palliated in software by using multiple TLB coherency domains (see Section 7.4.1).

The "*DomOS*" experiment follows the ideal performance very closely for the on-disk database, achieving a 1.17× speedup for 64 threads. The little difference is mainly due to slightly less improvements on user time, since it executes the caller/callee stubs and proxy routines, and the limitations of "*KML+SASOS*". It also achieves a 1.24× speedup in the in-memory database for 64 threads, but the limitations in "*KML+SASOS*" have a larger negative impact on this case. Importantly, the speedup of the in-memory database with 4 threads is of 2.15×, and as explained earlier adding more threads has a lower impact on total throughput (compared to the on-disk database version). The 1 GB database shows similar results for the in-memory database. Nonetheless, the on-disk case is even more I/O-bound, masking the speedups of DomOS to around 7%.

These results clearly demonstrate the potential benefits of the DomOS isolation model and its reduced domain crossing overheads, as opposed to the severe performance impact of existing IPC mechanisms. DomOS outperforms a standard Linux setup by up to 1.2× when using an I/O-bound setup (on-disk database), and is up to 2.15× faster when palliating I/O overheads with an in-memory file-system for the database. In all cases, it achieves more than 85% of the ideal system efficiency.

Chapter 9

Conclusions

This thesis presents a hardware/software co-design for efficient isolation of software components. The hardware substrate (CODOMs) provides a hybrid protection model that provides a high degree of performance while solving some of the shortcomings of other works targeting isolation. It makes isolation of short routines feasible by providing function calls across isolation domains in a way that does not thwart ILP in out-of-order cores. Importantly, it provides very efficient mechanisms to allow revoking cross-domain access grants; something that in previous works was either avoided or induced large overheads on every capability access.

CODOMs is also designed to provide expressive and composable isolation primitives. These properties can be used to add more efficient isolation in a gradual way that maintains backwards compatibility. Gradual adoption and compatibility are also heavily embedded into the design, up to the point that cross-domain calls are implemented using regular function call and return instructions. When combined, these features allow implementing an unlimited number of domains in ways that require from no changes to minimal changes on existing code, and eliminate the need for redundant protection mechanisms like privilege levels.

An OS design (DomOS) that takes advantage of the underlying hardware mechanisms is also provided to make cross-process communication much more efficient. Instead of using the OS kernel as an intermediary for cross-process communication, the proposed system streamlines communication through direct process-to-process function calls. This greatly simplifies programming by eliminating the implicit concurrency and communication interface complexities that processes impose in current OSs. Performance is also carefully taken care of by eliminating all isolation-related code that is not needed on a case-by-case basis. Instead of using the typical approach of bridging cross-domain calls through a generic OS kernel routine, a set of compile-time and run-time optimizations are applied to dynamically attune execution to its isolation requirements. Quantitative measurements show that the proposed primitives are up to 24× times faster than Linux pipes, and up to 14× times faster than IPC in L4 Fiasco.OC. Performance gains also hold for a large multi-tier web server benchmark that is not perceived as being limited by IPC performance. The proposed system performs up to 2.18× better than a baseline unmodified Linux system, and 1.32× on average. In all cases, it provides more than 85% of the ideal system efficiency.

Proposing changes across all layers of a system is a daunting challenge, but a necessary one to achieve larger benefits through co-optimization. Even then, there still remain lots of work to do. Integrating the code-generating compiler backend with knowledge from the added architectural features can further eliminate overheads. Just like the compiler knows which registers need to be saved across function calls, it could also know which part of the architectural state of a computation needs to be saved on cross-domain calls even

when the other end of the call is not trusted.

The proposed hardware mechanisms are also in a very good position to be adopted for isolating I/O devices and their drivers. In such a scenario, regular user code could drive low-level devices without fear of corrupting the system on an error nor the possibility of using such capabilities to access unauthorized information. This would provide a large leap forward from the current state of I/O isolation. On one hand, current device drivers pay large performance overheads to isolate them from user applications. On the other hand, using the IOMMU to isolate I/O devices from the rest of the system also imposes large overheads and OS kernel complexity [21]. By using the proposed design, applications, device drivers and I/O devices could all coexist on a single virtual address space, where CODOMs capabilities could be used to enforce protection of all components, including both software (applications and device drivers) and hardware I/O devices (by integrating capabilities with the IOMMU).

Since all applications in DomOS share a single virtual address space, memory translation resources (like the TLB, which are on the critical path) can be removed from the core. Instead, TLBs could be placed in other levels of the memory hierarchy. This would consolidate their costs and reduce hardware complexity and performance overheads. CODOMs tags could be assigned at a much larger granularity, like the segments found in IBM Power processors [61]. By placing memory translation (TLBs) on less-critical memory hierarchy levels, their operation could be overlapped with the already large latency of accessing main memory or larger shared caches.

The efficient cross-process calls prototyped in DomOS, could also be transparently integrated with systems that expose RPC interfaces as the basis for IPC, like GNU/Mach, since they already use an RPC interface compiler to generate caller and callee stubs. Pure object capability systems like EROS [105] also use the same approach, and could take even further benefits because cross-domain calls could also be transparently applied to requests to OS kernel resources.

Finally, efficient cross-domain calls can also be applied to implement *pico-para-virtualization*. Instead of using complex hardware or emulation to virtualize machine resources like I/O devices or certain instructions, cross-domain calls could be used instead. By providing efficient cross-domain calls, one can use function calls as the interface to virtualize operations at a very small granularity. For example, to replace virtualization-critical instructions or the memory write operation that triggers an I/O device operation. If the caller is being virtualized, the call would be directed to the virtualization layer, while executing in a non-virtualized environment would simply translate into calling a function on the same domain.

Clearly, providing flexible and efficient primitives for software component isolation allows programmers to use new secure domain composition designs that were until now unfeasible due to existing overheads.

Appendix A

FlowTAS: Making Data-Centric Mandatory Access Control Practical

This annex describes a secure infrastructure for running untrusted third-party applications on security-sensitive information. The entire infrastructure was designed and developed from scratch as a result of a stay at the Spark group at UT Austin, led by Dr. Mohit Tiwari. The infrastructure is designed to work on existing systems (various isolation mechanisms are evaluated), but results from Section 8.2.2 show that infrastructures with this type of characteristics could get large benefits from the system proposed in this thesis. The evaluation of the entire infrastructure has been eliminated from this appendix for the sake of brevity.

A.1 Introduction

Platforms such as Google Drive, Dropbox, and TrueVault[1] present the simplest of privacy controls to users. Users create and share *folders* with other users, a form of user-level discretionary access control (DAC) – and then insert data such as images, text, and video into each folder. Folder-level access control is particularly attractive since users can set access controls independent of specific applications. Once a folder is shared, all PDFs, presentations, spreadsheets, images, health records, etc. are shared with only the folder's Access Control List (ACL), implying a Mandatory Access Control (MAC) policy over all applications that use a folder. What users want, thus, is *data-centric* DAC that the system translates into MAC over each application.

In practice, however, access controls are *application-centric*, and a malicious or compromised application today can easily violate users' folder-level access controls. Consider an example from healthcare in Figure A.1. TigerText handles all messages, ZocDoc handles scheduling, and Athena Health handles health records, and all store encrypted data in HIPAA-compliant TrueVault databases (the last two not shown in Figure A.1 to save space). For such software-as-a-service (SaaS) applications, users can only decide whether an application has access to a *data type* — e.g., messages, calendar events, or health records. The application then has unfettered access to all messages (TigerText) or all calendar events (ZocDoc), which opens the door for data breaches.

Data breaches can happen when an application simply copies data over to an unauthorized entity. However, enterprises, such as hospitals, can prevent such leaks by *self-hosting* third-party applications — in an on-premise server or a private-cloud [1, 14] — and by firewalling the app from communicating to outside

[1] https://truevault.com: HIPAA compliant storage

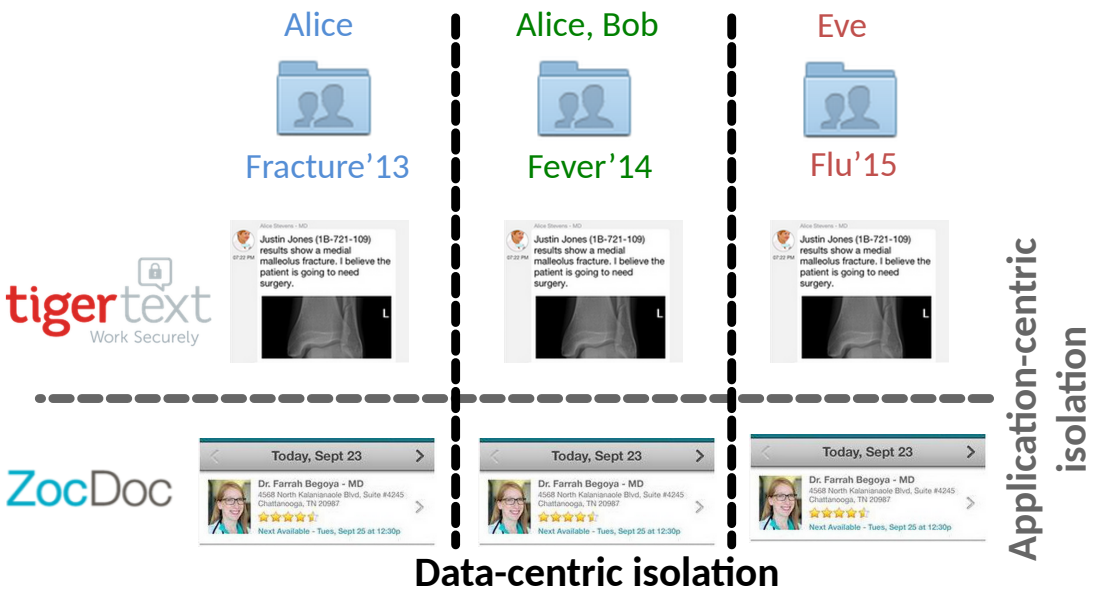


Figure A.1: Current systems provide application-centric access control, where compromised or malicious apps can cause data breaches. E.g, TigerText leaks all Alice's and Bob's messages to Eve. **Data-centric access control** matches users' expectations better — e.g., all messages and calendar events in folder Fracture'13 are only shared with Alice.

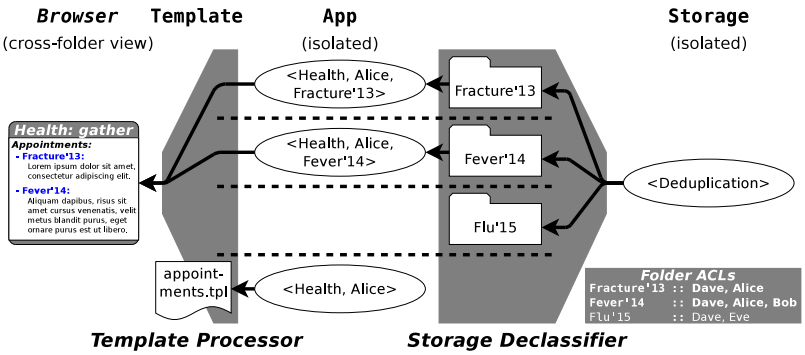
entities. A more insidious breach can occur when a malicious or compromised application copies data from one folder to another with a different ACL. For example, if TigerText is compromised, it can leak all of Alice's and Bob's messages into Eve's folders. Self-hosting and firewalling external communications cannot protect against such breaches, which are extremely harmful for an enterprise that works with sensitive healthcare [4] or financial [11] data.

Our goal is to prevent such data breaches by making data-centric access control practical. To this end, we design and build FlowTAS — a platform for self-hosted clouds to turn users' folder-level sharing decisions into a MAC policy on untrusted applications. Figure A.1 illustrates our baseline approach. FlowTAS runs *separate TigerText instances for each folder*, so that even a compromised or malicious TigerText instance cannot leak Alice's data into Eve's folders. Folders thus become units of isolation. FlowTAS enforces *non-interference across folders* by executing one isolated application instance per folder, therefore forbidding untrusted code from accessing data from two or more folders. It also provides folder non-interference on untrusted cross-folder storage components.

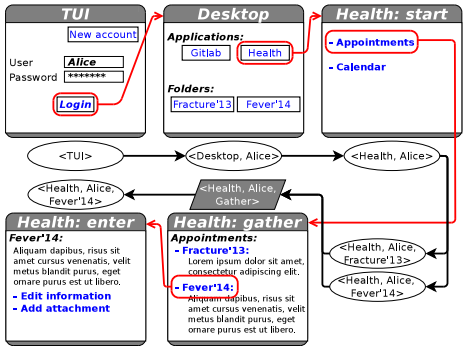
FlowTAS's design relies on "containers" as a fundamental isolation mechanism. The containers can communicate with trusted FlowTAS services, but are firewalled from all other external communication, including other containers. Interestingly, containers can be implemented as reference monitors [8, 108] that can be deployed today, or more efficiently using capabilities [120, 123] or decentralized information flow control [37, 43, 68, 115, 130] in future systems.

Data-centric per-folder MAC, however, poses two major technical challenges.

Challenge 1: Restricted functionality. Running separate application instances per folder curtails important application functionality. For example, a messaging application will not be able to show an inbox of all



(a) Developer view of the TAS pattern to produce the *"Health: gather"* view.



(b) Flow of user interaction views.

Figure A.2: High-level overview of the TAS (`template-app-storage`) pattern. Gray boxes are trusted components, and circles are isolated containers. Applications without cross-folder views can simply run on a per-folder container (`app` component). The *Template Processor* builds cross-folder views from untrusted `templates` provided by the application. The *Storage Declassifier* allows cross-folder data optimizations from untrusted `storage` components. Existing browser mechanisms ensure folder non-interference on the client.

messages, or search for a message across all folders. A calendar application will be split into one calendar per folder without a view of all events for a user. Storage services rely on deduplication on unencrypted data to minimize cost — now, deduplication can only be applied per folder and thus will be inefficient. It is evident that simply replicating the entire application per folder is not a feasible design point, even if it makes the system's access control invariant desirable and easy to describe.

Challenge 2: Slowdown. Executing several instances of an application concurrently, each in its own container and each accessing a different folder, *multi-execution* [26], puts greater pressure on CPU and main memory resources than the non-secure baseline of one application for all data. Our evaluation finds that existing container isolation systems impose 3.1× latency and 3.0× throughput slowdowns compared to a non-secured application; some of which can be eliminated using well-known web-proxying techniques. Importantly, cross-folder operations suffer from overheads in excess of 70×, which are intrinsic to the multi-execution model. Nonetheless, experiments using a prototype architecture are able to exploit the FlowTAS pattern with much lower overheads.

Making FlowTAS programmable and low-overhead is crucial to make data-centric access control practical. To this end, **we make the following specific contributions**:

1. Programmability: Application design pattern. To make applications functional in a data-centric MAC system, we propose a novel yet simple design pattern (shown in Figure A.2) that developers can use to port their applications to FlowTAS. The design pattern — `template-app-storage` (TAS) — enables developers to start with their entire application as an `app` component. This `app` component provides per-folder functionality, while the `template` and `storage` components enable cross-folder functionality. Each component runs on a separate container, making their isolation simple to enforce. We present a safe templating language that developers can use to construct views of cross-folder data, creating message inboxes, calendars, and similar pages. Further, we demonstrate that several untrusted application modules, such as deduplication, key-value stores, compression etc, can be given access to cross-folder data as long as they present a known interface to the rest of the application. Overall, the TAS pattern integrates well with Model-View-Controller (MVC) based web applications, providing developers with a simple security-agnostic method of partitioning their applications.

2. Non-interference: Robust declassifiers. We have to ensure that cross-folder functionality does not break non-interference across folders. To this end, we propose *robust declassifiers* (see Figure A.2) to ensure that (a) data from different folders cannot interfere inside a `template`-generated `view`, and (b) `storage` modules cannot leak data from one folder into another. Our templates constrain the developer to functionality that can compose data into `views` but not affect any data sent to an `app` when a user follows a URL from a `view` back into a specific `app`. This is enforced through a combination of existing browser mechanisms and a trusted *Template Processor* that composes the cross-folder `views` from untrusted `templates`. `App` components send JSON objects to the *Template Processor*, which uses the data to inflate a `template` provided by the untrusted `app`. For `storage` components, we propose a *Storage Declassifier* that integrity checks each data read operation to ensure that it only returns the value from the corresponding most recent write operation. Interestingly, this *Storage Declassifier* can ensure that even components that work with *unencrypted* data, such as deduplication, can be given access to cross-folder data without breaking cross-folder non-interference.

3. Slowdown: Efficient containers using capabilities. We have built and optimized prototypes using two off-the-shelf containers, LXC [8] and a custom SELinux [108] module, which make applications easy to port. We have also measured the performance of pulling data from several per-folder containers and composing a cross-folder `view`. It yields overheads of more than 70× in the worst case, which are intrinsic to the multi-execution model. Hence, we have designed a new container prototype based on capabilities [120] that puts many `app` instances in the same multi-threaded address space and uses capabilities for light-weight isolation. Our capability-based FlowTAS design reduces slowdown to only 3.1× in a worst case stress test. Applications such as document/presentation editors where users typically work with one document at a time fit well on FlowTAS and incur vastly lower overheads.

4. Evaluation: Seven web applications. We have developed 5 web-based applications from scratch (a mock healthcare application, programming IDE, calendar, image viewer, and PDF viewer) and ported 2 off-the-shelf applications (Gitlab and Hacker Slides) to FlowTAS. We find that across all seven off-the-shelf applications, ∼600 lines of code were modified, and ∼1.25M lines of code were removed from the trusted code base (detailed in Table A.1 and Table A.2).

A.2 Motivation

We now present a motivating example that we will use to define our problem, demonstrate threats against user-data privacy, shortcomings of previously proposed solutions, and where FlowTAS fits in.

A.2.1 Example Setting and Privacy Expectations

Alice, Bob, and Eve are patients in a hospital under the care of Dr. Dave. Dave maintains per-encounter folders that he shares with relevant patients, e.g., Fracture'13 (with Alice), Fever'14 (with Alice and Bob), and Flu'15 (with Eve). Each folder includes data generated by applications such as messages, calendar events, to-do milestones, health records, images, and other documents. Maintaining these documents in cloud-backed storage, such as Google Drive, Dropbox, and True Vault, and using web-based applications enables Dave to share folders and collaborate with his patients.

Dave's expectations about the confidentiality of his online data can be expressed for almost all applications in terms of access-control lists (ACLs) as shown in Figure A.2: e.g., Fracture'13 information should only be accessible to Alice. Unfortunately, even if Dave translates his confidentiality expectations correctly into ACLs, many opportunities exist for his information to be diverted inappropriately to recipients he has not explicitly authorized.

A.2.2 Data Leaks with Untrusted Applications

Opportunities for data leaks are primarily rooted on the fundamental difference between access control and information flow control. Access controls govern how a hosting platform grants access requests by other users to Dave's documents; they are, essentially, *perimeter controls*. They say nothing about how information flows internally within the platform.

For example, the system is not secure even if the platform implements ACLs correctly and prevents Eve's attempts to read Fracture'13 documents but allows her to see a Flu'15 photo. If the contents of a Fracture'13 X-ray are somehow copied into a Flu'15 health record, Eve would be able to read them anyway. This could happen in the cloud (say within Google Drive) if a buggy image-indexing application ends up inadvertently copying image content from one file to another. Even worse, malicious applications can exploit legitimate cross-folder accesses to exfiltrate information across folders. In the worst case, these could result in information leaking to *all* folders that an application can access.

Ultimately, granting an application access to private information implies trusting that application; if Dave allows Google Presentation to access Alice's medical record, he implicitly trusts it not to spread the information from that record all over the other documents Presentation has access to. In summary, all documents accessible by an untrusted application effectively share the same privacy policy (e.g., the same ACL), even if their owners have set that policy differently. System- or language-based confinement has been the typical response, as we see next.

A.2.3 Plugging Untrusted-Application Data Leaks

To deal with promiscuous information flows among sensitive documents, careful tracking of information-flow labels has been designed in various guises. One effective example is **programming-language support for information-flow control**. Applications built on top of such languages (e.g., JIF [99], Hails [52], Resin [128]), carefully annotate program inputs, intermediate variables, and outputs, to ensure that information does not flow between inputs and outputs that have incompatible privacy policies. Unfortunately, such

languages are notoriously complex to use in practice, they require that developers have significant security expertise, and constrain those developers to use only supported programming frameworks, libraries, and components. What is worse, these approaches only work as long as application developers are trusted [99], or when platform developers maintain the model for all third-party MVC applications [52]—which might not be the case for many mobile and web-based applications [91].

OS-level containers. A typical, more brute-force response is to put an untrusted application "in a box": confine it in an operating system container and grant access to that container only to authorized users. This ensures that, regardless of what the application does, information flowing out of a confidential document stays within the container and accessible only to authorized parties. Mechanisms for the definition of fine-granularity privacy domains have been studied extensively, including OS virtualization (e.g., LXC [8]), MAC systems (e.g., SELinux [108] and SEAndroid [107]), and decentralized information flow control OSes (i.e., DIFC systems like HiStar [130], Flume [68], and Hails [52]). Moreover, logical boxes that extend across the cloud servers and client devices have also been demonstrated (e.g., πBox [70]).

OS-level application-centric confinement as employed by Android, web-browsers, cloud applications, and πBox [70] follows a container-per-application model. Unfortunately, this makes it difficult for Dr. Dave to isolate the various privacy domains (folders) he may have access to: if he grants access to an image-manipulation application to "his files" (e.g., images on all his folders), then vulnerabilities in the application might still leak Alice's records to Eve.

As a data-centric alternative in our example, the system can have distinct containers for Fracture'13, Fever'14, and Flu'15 (ellipses in Figure A.2), and then use arbitrarily buggy or malicious applications within each separate container. All users can remain assured that no information leaks from one folder to another, eliminating the leak scenarios described above. Using existing systems to map each folder to a unique information flow label and enforce non-interference [52, 68, 130] would be sufficient — except that legitimate cross-folder functionality will be curbed due to illegal information flows.

A.2.4 Limitations of Prior MAC systems

Cross-folder views. On the user-facing front-end, Dr. Dave often needs to inspect information from multiple folders. Although he could maintain a separate calendar, email account, and search index per folder, being productive is at odds to such extreme separation. Figuring out if he is available for a 2pm meeting without mixing information from all of his calendars would require him to check the 2pm meeting slot in each calendar, and the more distinct folders he had access to, the more tedious the exercise would become. It is more convenient to have a global calendar spanning all the separate calendars from each of the distinct folders Dr. Dave can access (*"Health: gather"* view in Figure A.2b). However, aggregating all of his calendars together, finding an entry in question (e.g., an existing Flu'15 meeting on the 2pm slot), and modifying that entry subsequently (e.g., attaching an image to the Flu'15 calendar entry) violates a number of information flow rules; it is possible that some information that came from a Fever'14 calendar entry, via the aggregate calendar, flowed into the Flu'15 entry without Dr. Dave's explicit intention to do so.

Cross-folder storage optimizations. Similarly, on the back-end, developers or even platforms often need to store data from all users in a single global storage layer, such as an SQL database or Google's BigTable, due to cost or time-to-market concerns. Developers will logically attempt to keep items with distinct ACLs separate, but global operations such as deduplication will "mix" information with distinct ACLs (right-hand of Figure A.2a). So a presentation file containing Fracture'13 data may be deduplicated with identical files from Fever'14 (both of which are accessible to Alice but otherwise distinct), or even worse from completely unrelated folders like Flu'15 with the same file, before being written to disk. This means that a potentially buggy operation on the storage platform can leak Alice's information to unauthorized users like Eve (e.g., a

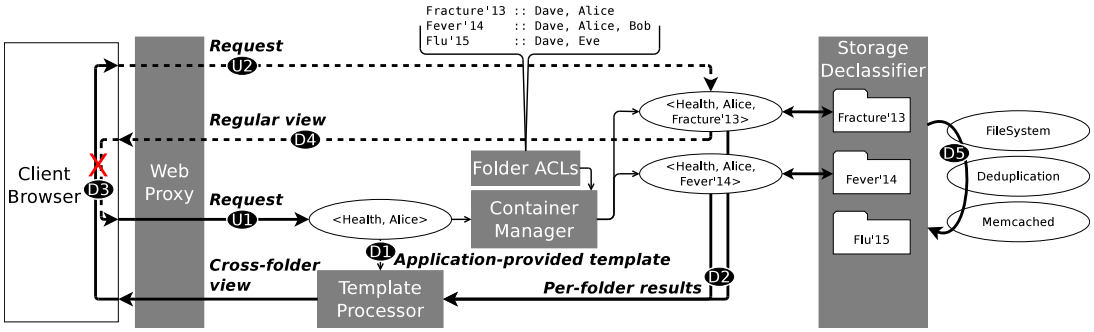


Figure A.3: FlowTAS design pattern and infrastructure overview; the trusted code components of FlowTAS are shown as gray boxes. **User flow**: The user (*"Alice"*) logs into the *Trusted User Interface*, which redirects her to the *Trusted Desktop Interface* (not shown). After selecting an application (*"Health"*), U1 the user requests an operation that triggers a cross-folder `view` (*"Fracture'13"* and *"Fever'14"*). U2 From that `view` she can select to operate with the data on a specific folder (*"Fracture'13"*). **Developer flow**: Untrusted developers split applications into `template`, `app`, and `storage` components, which are executed inside containers (ellipses). FlowTAS executes multiple instances of the `app` components (*"Health"*), each on a separate container and running on behalf of a user (*"Alice"*). An `app` instance can trigger a cross-folder operation when it does not have access to any folder (container *"<Health,Alice>"*; bold solid arrows). D1 The programmer provides an untrusted `template`. D2 The trusted *Template Processor* then uses it to render a `view` with information gathered from multiple `app` instances, each with access to a separate user's folder (*"Fracture'13"* and *"Fever'14"*). D3 FlowTAS ensures folder non-interference (even with malicious programmers) by prohibiting information to flow (directly or indirectly) between containers. D4 `App` instances with access to a folder can directly interact with the client (bold dashed arrows). D5 Developers can also specify cross-folder `storage` optimizations — the *Storage Declassifier* ensures data non-interference across folders.

faulty modification by Alice on the copy in Fracture'13 is seen by the deduplicated copy in Flu'15).

Such back-end mixing is common not only with deduplication, but also with other storage optimizations that benefit in performance from operating over larger collections of (unencrypted) data, such as in-memory key-value stores, compression, erasure coding, striping, etc. This back-end mixing of information, again, strictly violates information flow rules and existing work on declassifying data from an untrusted process does not *robustly* support such operations [52, 99]. Furthermore, as with PL-based DIFC, DIFC-enabled storage layers (e.g., IFDB[103]) can help developers protect their users' data from information leaks, but they require sophistication in setting privacy labels, as well as porting existing applications to a new storage platform.

Declassification. The way out of such information-mixing conundrums, legitimate violations of strict information-flow confinement, is to *declassify*. Declassification mechanisms are custom application logic that decides which cross-folder information flow is actually allowed by the intent of the policy; such logic must be trusted, since it can violate strict information-flow rules. But declassification logic tends to be complex and highly critical to security; if it is incorrect, it can compromise the confidentiality of the system, and it is strictly harder to write correctly than the applications themselves. Prior OS-based MAC systems [68, 130] would mark such cross-folder views and storage modules as trusted declassifiers. The downside is that writing such application-specific declassifiers falls upon the same application developers who, ultimately, can be expected to be neither trustworthy nor security experts.

A.2.5 Problem Definition

We seek to (continue to) enable users to share their sensitive data, using privacy policies as simple as ACLs on folders, on web-applications that run in private clouds. We target a threat model in which the platform is itself trusted (i.e., a private cloud), but the applications hosted within the platform (and their developers) are not. This private cloud requirement can be relaxed, however, and an enterprise could also use other Infrastructure-as-a-Service providers using attestation and trusted boot mechanisms [62, 89]. Further, our design is not limited to web-applications — any application that can interface with user-facing views can work by replacing our `templates` with, e.g., Android layout templates.

We are particularly interested in simultaneously achieving the following properties:

P1 Fine-grained, cross-application privacy domains. Definition of privacy domains, who can access which folders, should be independent of application-specific constructs like messages, events, health records, etc.

P2 MAC over applications. Policy definition should be enforced in depth, i.e., on all applications and not just on users, to prevent cross-folder information flows.

P3 Untrusted applications. Applications should not be assumed to be correct nor benign.

P4 Development freedom. Applications should be allowed to use any language, library, or framework.

P5 Safe cross-folder user views. Per-user global views across all privacy contexts (folders) should be possible, without leaking information across contexts.

P6 Safe cross-folder storage optimizations. Global storage optimizations across privacy contexts (folders) should be possible, without leaking information across contexts.

We limit the scope of our problem by taking integrity out of the discussion. Specifically, although it is our goal to prevent untrusted applications from violating users' privacy policy, we do not prevent them from

mangling user data or destroying it. A system such as Frientegrity [46] may compliment our approach in that respect.

We also place out of scope information leaks via timing or termination channels. We assume complementary techniques that either randomize (through fuzzing [117]) or normalize (through deterministic execution [17]) the timing of outputs.

Finally, users fully delegate trust in handling sensitive data to other users they share folders with. We do not seek to protect against poor judgment when sharing a sensitive file with an untrustworthy friend; that friend is free to take that sensitive file and communicate it to the world. Plugging the "analog hole" is out of scope for our work.

A.2.6 FlowTAS Overview

FlowTAS's design is founded on the following principles, which are exemplified in Figure A.3. Users work with folders as the simple abstraction for access control (P1; *Folder ACLs* component in Figure A.3). Folders are finer-grained than Home vs. Work virtual machine based control [98, 101], simpler than defining long-term security label lattices [39, 64], and are popular in platforms like Google Drive and Dropbox.

An application instance is launched within a container that is tied to a specific folder. This mandatory access control over applications ensures that information does not leak across folders (P2; different containers in Figure A.3). Since containers are OS-based, each application instance gets a standard system view, including system calls, libraries, and runtimes for any convenient development framework. For example, programmers may use standard frameworks like node.js, Django, PHP, and Rails, or other completely custom and buggy frameworks (P3 and P4). This is important for adoption given the diversity in toolkits for developing complex applications.

Applications that require no declassification trivially run within a per-folder container (P4; bold dashed arrows for Fracture'13 in Figure A.3). Those that do require cross-folder viewing or cross-folder back-end functionality, however, must be refactored into the TAS pattern such that they provide one or more `views` (*"Health: gather"* in Figure A.2b), an `app` component to provide the bulk of the user-facing functionality (*"Health"*), and `storage` components (e.g., *"Deduplication"*). This refactoring task is privacy-insensitive and developers who are security-novices only need to work with functionality-based decisions. In contrast to refactoring to TAS, assigning security labels to program variables [99] or the Model [52] (in a MVC application) requires security expertise.

A cross-folder view, such as listings of notifications, encounters, prescriptions, etc., in an untrusted application is expressed in terms of a `template` (bold solid arrows in Figure A.3). FlowTAS's trusted *Template Processor* "gathers" data (JSON objects in our prototype) generated by each per-folder `app` component and inflates the `template` to create `views` for the user. In this system, Dr. Dave can view all encounters and transition into a specific `app` component to create new or update existing encounters. The *Template Processor* ensures that data (JSON objects) from one folder's `app` are used to create URLs (i.e., HTTP requests) to only that same `app` component (i.e., accessing the same folder). Hence, HTTP requests and arguments sent from a `view` to each `app` can be declassified safely to the client (P5). Incorrect or otherwise malicious templates cannot send one folder's JSON to another since the template processor creates the URLs on a per-JSON-object basis. Thus, the template processor fulfills the requirement for *robust declassification* [129].

The template language also includes special tags for a `view` to trigger search and sort operations across data gathered from all folders. Interestingly, such minimal functionality aligns well with popular template languages [7, 10] and supports many common use-cases (P4).

Back-end cross-folder functionality (application-specific or provided by generic storage layers) is executed in separate global back-end `storage` containers (right-most containers in Figure A.3). Their in-

formation flows back to the rest of the `app` containers, and is thus declassified via a *Storage Declassifier* (right-hand side of Figure A.3). The *Storage Declassifier* keeps track of all data objects (at units such as files, disk blocks, or key-value pairs) flowing from an `app` to a `storage` component, and only allows the same, seen data objects to return back to that `app` (P6). Suppose an X-ray image is pushed by a Fracture'13 `app` to a deduplicating `storage` component (and its hash recorded by the *Storage Declassifier*); when the same file is fetched by the Fracture'13 `app`, the *Storage Declassifier* ensures that the file returned from `storage` has the same content digest and only then "declassifies it", allowing it to flow back to the `app`. By this storage and declassification, back-end information mixing cannot expose information across folders, since `apps` explicitly cannot obtain new information from storage (recall we have placed implicit channels such as timing information leaks out of the scope of this work).

Mixing information across folders. Functionality such as analytics (e.g., clustering, training classifiers, etc.) on data across folders is a valuable component of "big data" applications. Such functionality fundamentally violates the access control policies set by users, and instead requires complementary approaches such as differential privacy [42, 84, 85, 93, 96] or quasi-identifier based privacy [74, 81, 111]. Differential privacy literature has shown that releasing datasets is fundamentally unsound. Instead, the privacy preserving approach is to bring analytics functions into a container with all private data and only release the perturbed output of the function on the private dataset. This approach, as exemplified in GUPT [85], integrates directly with FlowTAS as the differentially private container reads all folders' data, receives analytics functions from the developer, and releases declassified output to the developer if the privacy budget is not exhausted. Similar declassifiers can be built for advertisement impressions [70] and debugging output [29].

A.3 Design of the FlowTAS System

FlowTAS is comprised of five primary software components, which are shown in Figure A.3: (A) the *Trusted User Interface*, (B) the *Web Proxy*, (C) The *Container Manager*, (D) the *Template Processor*, and (E) the *Storage Declassifier*. The design and interaction between each of the components can be best explained by examining a typical user workflow with the FlowTAS system. Importantly, our design is not limited to web-applications; any application that can interface with user-facing views may work by replacing our `templates` with, e.g., Android layout templates.

A.3.1 Trusted User Interface

A user begins an interaction with FlowTAS by authenticating through the *Trusted User Interface (TUI)*. Similar to Google Drive or Dropbox, the TUI provides functionality such as creating folders, uploading data to folders, and launching applications. Additionally, the TUI provides users a simple way to manage ACLs through folder sharing decisions. In fact, the TUI is a regular containerized application that has additional rights to request changes to the folders' ACLs of the user it runs on behalf of. Since `app` instances run inside containers, clients interact with a *Web Proxy* that dispatches requests to the appropriate container.

A.3.2 Web Proxy

The *Web Proxy* serves two roles. First, it dispatches client requests to the appropriate container, and forwards their responses to the clients. Secure dispatching to containers is implemented through a mix of cookies (for authentication) and randomly-generated URLs (for authorization). Second, it provides all FlowTAS abstractions (users, folders, `apps`, ports, etc.) to developers through a RESTful API accessible via secure HTTP. Authentication and authorization of each function on the API is implemented through a secure random token

argument (i.e., a sparse capability), a technique widely used in existing web service technologies [12, 63]. The set of capabilities (e.g., create folders, share folders, launch applications, etc.) given to an `app` instance (i.e., container) is established according to the user's role and the application category, and is passed to the `app` instance when started. Therefore, the TUI is trusted in that it receives more capabilities than other applications to the system operations, but otherwise uses the same mechanisms as any other application. The folder sharing decisions (*"Folder ACLs"* in Figure A.3) are later used by the *Container Manager*.

A.3.3 Container Manager

Isolated execution environments (containers) are at the core of FlowTAS. They are managed by the *Container Manager*, shown in Figure A.3, which ensures the non-interference property of all folders when creating containers. This property is trivially enforced by the *Container Manager* as it creates a new container for each folder, and uses a MAC system to ensure complete container isolation (see Appendices A.4.2, A.4.3 and A.6). This implies using a separate `app` instance for each folder (since they run on separate containers), but optimizations are also explored in Appendix A.6. To avoid cross-application interference (e.g., safeguard trade secrets on application code), FlowTAS runs `app` and folder pairs on separate containers.

As explained in Appendix A.2, container non-interference enables P2, but makes P5 impossible by design. As we discuss next, FlowTAS facilitates cross-folder `views` through the *Template Processor* component.

A.3.4 Template Processor

The *Template Processor* component, shown in Figure A.3, renders `templates` provided by applications that request cross-folder information. First, the `app` returns a `template` to the *Template Processor* which, in turn, requests that per-folder data be collected. This data is collected using a per-folder container, trivially maintaining the non-interference property. Since `app` instances run on behalf of a user, the *Folder ACLs* component (Figure A.3) limits through what folders the operation can take place. The per-folder data is then aggregated by the *Template Processor*, which renders it according to the provided `template`. Therefore, `templates` are a key element of FlowTAS to provide cross-folder functionality (P5; see Appendix A.4.3 for more details).

Robust Declassification: Data from different folders must be declassified to render it on a single page on the user's browser, while ensuring it cannot flow to other folders. To prevent this information from flowing to unauthorized subjects, FlowTAS executes a simple implementation of Information Flow Control (IFC) during the `template` rendering phase and on the client side (see Appendix A.4.4 for more details). The rendering of `templates` is carefully controlled through a stateless templating language. On the client side, it is controlled through a mix of the Same-Origin Policy (SOP) functionality (already implemented by browsers) and controls over the JavaScript on the client (which can otherwise observe and manipulate cross-folder data to exfiltrate data). Several JavaScript sanitation and parsing libraries exist, but the syntax of HTML and JavaScript make it difficult to prevent all possible malicious or buggy JavaScript injections from being executed. Therefore, rather than implementing an IFC policy on client-side JavaScript, FlowTAS forbids using arbitrary JavaScript in `templates`, except for trusted snippets provided by the template language (see Appendix A.4.4 for more details). Note that regular `views` that do not go through `templates` are not affected.

A.3.5 Storage Declassifier

The *Storage Declassifier* component, the right-most gray box in Figure A.3, allows untrusted `storage` components to apply optimizations across folders (e.g., to consolidate storage resources through cross-folder deduplication; P6).

FlowTAS allows applications to use untrusted *Storage Backend* components with access to cross-folder data while maintaining folder non-interference through a trusted *Storage Declassifier* component that keeps track of data integrity (one component per supported interface). The current prototype provides a trusted *Storage Declassifier* component that implements a simple key/value storage interface with *put* and *get* operations. It keeps track of the integrity of data that is put into it, and drops the results of get operations whose data integrity has been compromised. Therefore, the trusted component acts as a transparent proxy to the real untrusted *Storage Backends*, which can safely operate with data from multiple folders (right-most boxes in Figure A.3). Note that in a production system, the file-system itself could be used as the trusted *Storage Declassifier* interface, transparently redirecting operations to untrusted *Storage Backends* [109].

Another legitimate use-case for declassification is using existing databases for compatibility with large code-bases. Databases like MySQL are already deployed in privacy-sensitive environments. In this case, FlowTAS provides a small trusted *Service Declassifier* component that configures the existing authorization mechanisms in the *Service Backend* component (the database) to isolate the per-folder information (e.g., using different per-folder `views`). The *Service Declassifier* can then hand out per-folder authentication information to each `app` instance (i.e., each container). Note that *Service Backends* are a special case of *Storage Backends*, since they can be trusted to maintain folder non-interference.

All component instances (both trusted and untrusted) run isolated from each other; i.e., on different containers. Access to services is then managed through the port grant capabilities provided by the RESTful API implemented by the *Proxy*. For example, regular `apps` are given capabilities to access the trusted declassifiers; in turn, these trusted components have capabilities to access the untrusted backends.

A.4 FlowTAS Implementation

FlowTAS exposes a simple ACL-based system (which is data-centric by nature) to its users to manage the sharing of folders (P1). Additionally, FlowTAS imposes the non-interference property between folders (P2). Intuitively, one would translate these into IFC policies, since they provide a convenient way to express data-centric policies. Instead, FlowTAS uses a MAC system to enforce its properties. MAC systems are readily available on most environments, and can be generally enforced at lower costs than IFC. Nonetheless, MAC is eminently application-centric (i.e., the application acts as a subject for controlling access), while FlowTAS's policies are data-centric (i.e., information flow is controlled using the data as a subject for control).

FlowTAS turns MAC controls from application-centric into a data-centric policy-enforcement mechanism by running a separate instance of each application, and allowing each instance to access a single folder. Internally, FlowTAS makes a few exceptions to this simple rule in order to make writing applications practical (see below).

A.4.1 Object Categories for Data-Centric MAC

FlowTAS uses folders as the core storage abstraction to control all information, regardless of whether they are generated by the user or not. Therefore, it internally distinguishes between the following folder categories:

Application Are automatically created when installing an application (shown as *"<App>"*). When an application runs, it always has read-only access to its folder.

Settings Are automatically created the first time a user starts an application (shown as *"<App,User> Settings"*). They store user-specific settings for the application.

Data Are managed through desktop applications (see below), and contain regular user data. The term "folder" without a category usually refers to data folders.

Containers also have an internal category property, which is used to distinguish between different application phases:

Start Have read-only access to the corresponding application folder, and read-write access to the corresponding settings folder. This is the phase applications start on, where users can set their own settings (see Appendix A.4.3).

Gather Have read-only access to the corresponding application and settings folders. They also have read-write access to a single data folder. This phase is used to construct cross-folder `views` (see Appendix A.4.3).

Enter Have the same access rights as the *gather* phase/category. This phase is used to spawn application instances with access to a specific data folder (see Appendix A.4.3).

These simple rules ensure that data never flows across data folders: application folders can never be written to (except during application installation, which does not have access to data folders); and settings folders can only be written into from start containers, which do not have access to any data folder. The only place where data could flow across data folders is in the client's browser when it is running untrusted code from the application (e.g., JavaScript). As explained later, this is taken care of by randomizing URLs and ensuring the safety of cross-folder views (see Appendix A.4.3).

All applications in FlowTAS (trusted and untrusted) use the same mechanisms to run as isolated instances on containers. Therefore, the system distinguishes between the following categories to grant them different capabilities to interact with the system abstractions (see also Appendix A.3.5):

TUI FlowTAS runs a single instance of the TUI application, which has capabilities to manage and authenticate users.

Desktop Is the application that is started after a user logs in, which has capabilities to manage the applications and folders of that user.

Storage Backend FlowTAS runs a single instance of each untrusted storage backend. Each backend is tied to a single storage interface (see next).

Storage Declassifier FlowTAS runs a single instance of each trusted storage declassifier interface. Has capabilities to access all untrusted backends for that specific interface.

Service Backend FlowTAS runs a single instance of each "trusted" service backend (e.g., a shared database).

Service Declassifier FlowTAS runs a single instance of each trusted service declassifier. Has capabilities to access the target service backend.

Regular Instantiated for each user and folder pair; has capabilities to access trusted storage and service declassifiers.

A.4.2 Web Proxy and Container Manager

The *Proxy* is implemented in approximately 3,100 lines of Python, in the form of a Flask web server application [6]. It runs behind Apache, uses MySQL as a long-term database and Redis [15] as a short-term database and to implement distributed locks, and uses Celery [3] for running gather operations in parallel on the background. It currently exposes 23 different endpoints on its RESTful API, providing means to manage capabilities for applications, authentication, users, folders, files, sharing, ports, etc..

The *Container Manager* is written in around 1,600 lines of Python. It offers a RPC interface to the *Proxy* via Pyro [13]. It provides three different container implementations (LXC, SELinux, and CODOMs), which are discussed further in Appendix A.6.

A.4.3 Container and Application Instance Management

Each application is installed on its own folder through a tarball, which contains a manifest file that describes it (e.g., its name and category), and declares what other applications in the system it depends on (e.g., services). The *Proxy* can launch application instances on many hosts, each controlled by a separate *Container Manager*. Each application instance is uniquely identified through the application itself, its user and the data folder it has access to. Therefore, multiple effective containers with the same identifier are considered a single logic container where application instances can freely communicate without violating the isolation policies. This identifier is transformed into a random URL sub-domain (to avoid forging) that the *Proxy* uses to locate the target application instance.

Application lifecycle goes through three distinct phases, which trigger container and `app` instance creations. The *start* is the initial phase of all applications. When the application sends a response with the `x-flowtas-gather` header, the *Proxy* triggers a set of application instances in the *gather* phase and uses the *Template Processor* to render their results using the application-provided `template`. Finally, when the user clicks on a link generated by a gather template, the *Proxy* creates an application instance on the *enter* phase, which has access to a specific data folder. Each of these phases executes on different containers, and is therefore identified through separate sub-domains. For security purposes, cookies set by applications for any parent domain are dropped.

Start Phase: Application Spawning

Figure A.4 shows the sequence of operations involved in starting the *Health* application. The desktop application (on the `desktop` subdomain) uses the target application identifier (*Health*) to point the client to start it. ❶ The client's browser requests to start the *Health* application. ❷ The *Proxy* contacts the *Container Manager* service to request a new application instance. ❸ It creates a settings folder, and spawns the `app` instance with read-write access to the settings folder and read-only access to the application folder. ❹ The *Proxy* then registers the new *"<Health,Alice>"* sub-domain, and redirects the client to it. ❺–❻ From now on the *Proxy* simply forwards the traffic between the client and the new `app` instance.

Gather Phase: Cross-folder Data Aggregation

Figure A.5 shows the sequence of operations that allows the *Health* application to show a page with a summary of all the appointments in *Alice*'s folders. ❶ When the client requests the page at `/get-encounters` on the application, ❷ it responds with the `x-flowtas-gather` header. This header must be a JSON list (a command line), and the body of the response a gather `template`. ❸–❹ When the *Proxy* detects this header, it coordinates the creation of containers and `app` instances for each of the folders used for the gather

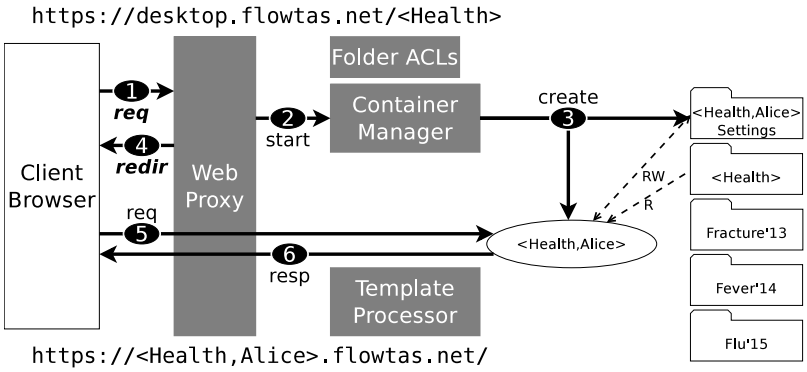


Figure A.4: Start phase: sequence of operations for starting a new `app` instance. The operation is triggered by requesting a URL with the target application's identifier (step ❶, top URL). Interaction with the new instance is triggered by a redirect to a URL that identifies the target container (step ❹, bottom URL). The application can use the "settings" folder (top-right) to store user-specific configuration.

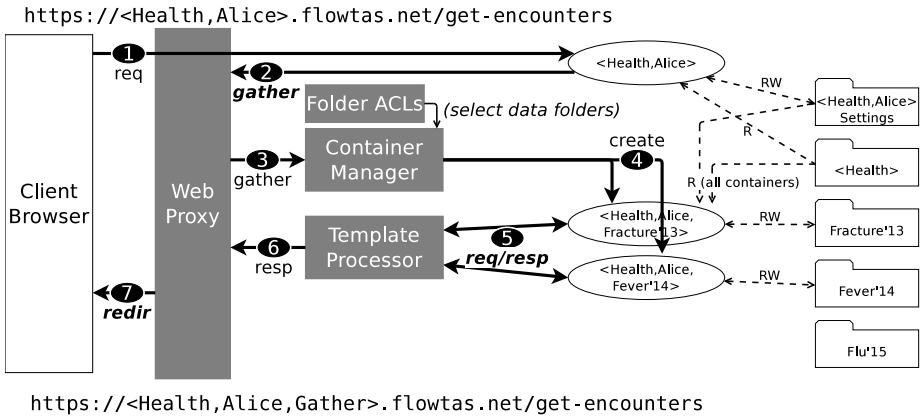


Figure A.5: Gather phase: sequence of operations for rendering cross-folder results. The operation is triggered by a *gather* request from the application running in its *start* phase (step ❷). The *Template Processor* composes the per-folder results using an `app`-provided `template` (step ❺). Finally, the *Proxy* redirects the client to the page containing the rendered `template` (step ❼).

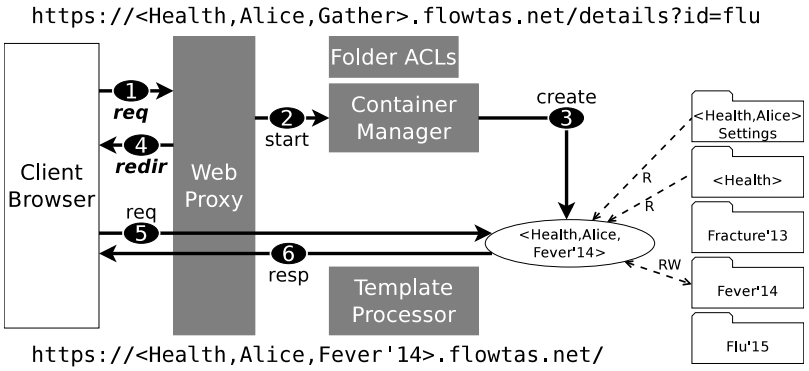


Figure A.6: Enter phase: sequence of operations for executing an application in a folder. The operation is triggered by requesting a URL that a previous *gather* phase registered to a specific folder (step ❶). Interaction with the new instance is triggered by a redirect to a URL that identifies the target container (step ❹, bottom URL). The application can still read the "settings" folder (top-right), but it cannot write into it to avoid interference across data folders.

operation. In the current prototype, all the data folders accessible by the user are selected (*Folder ACLs* component), but one could easily implement a way to select a sub set of these. Each new container runs the command provided in the aforementioned header. Each container also has read-write access to one of the selected data folders (*"Fracture'13"* and *"Fever'14"*), and all have read-only access to the application's and settings folders (*"Health"* and *"<Health,Alice> Settings"*, respectively). ❺–❻ FlowTAS then sends a request to each of the instances, which provide a JSON response, and uses these results to render the resulting page from the `template`. ❼ Finally, the *Proxy* redirects the client to a temporary sub-domain that serves the rendered page (*"<Health,Alice,Gather>"*). This enforces the use of the client browser's SOP mechanism, preventing a malicious JavaScript from analyzing the results by impersonating the client's browser.

Enter Phase: Editing folder Contents

Figure A.6 shows the sequence of operations that allows *Alice* to view the details of an appointment with the *Health* application. ❶ First, the client requests an address that was generated during a *gather* phase (i.e., `/details?id=flu`; see tag `FlowTAS.enter` below). ❷–❸ This triggers a new `app` instance on the selected folder, which is identified through its URL (i.e., `/details?id=flu`). The application is started with the command registered during *gather* (see tag `FlowTAS.entercmd` below), and its container has the same folders used by the *gather* (e.g., the data folder `Fever'14`, plus the settings and application folders). ❹ The *Proxy* then creates a new sub-domain for the new `app` instance, and redirects the client to it (*"<Health,Alice,Fever'14>"*). ❺-❻ From now on the *Proxy* simply forwards the traffic between the client and the new `app` instance.

A.4.4 Template Processor

The *Template Processor* uses the Python implementation of the Mustache templating language [10]. This templating language uses a set of tagged blocks to operate, and is state-less (it has no explicit complex

control-flow operations like loops) which makes it simpler to reason about. The *Template Processor* analyzes the `template` before and after expansion to ensure it is not used to exfiltrate information across folders. First, JavaScript is not allowed (except through special FlowTAS-specific tags; see below), since it could be used to observe and modify the resulting page. Second, links and forms that span beyond an expanded block are not allowed (they could mix information from multiple folders).

Regular blocks are expanded according to the results provided by each application instance during a *gather* operation. FlowTAS also provides two special tags. The first, `FlowTAS.enter`, is used to generate links that will be used to trigger an *enter* operation into a specific folder. The second, `FlowTAS.entercmd`, is used to specify a command to run on the new container created after the *enter* operation.

To make up for the lack of application-provided JavaScript in gather `templates`, FlowTAS provides a few additional tags that expand into trusted snippets of JavaScript. These provide functionality like searching and auto-completing across the per-folder gather results. It is important to note that none of these restrictions apply outside of gather `templates`.

A.5 Security Properties

This section summarizes the security properties of FlowTAS to clarify which piece is enforcing it and through what means.

The TCB of FlowTAS contains the trusted framework itself (gray boxes in Figure A.3), the underlying container implementation backend (e.g., LXC) and the client's browser. Therefore, from a security perspective the client's browser is part of a logic container, but the application code that runs in the client is not trusted (i.e., Javascript). Containers are isolated both "spatially" (direct communication) and "temporally" (communication across application phases).

Spatial container isolation: The *Container Manager* isolates containers at the file system and network level. Every container always has write access to a single folder, and initially can only interact with the *Proxy* (either to respond to client requests or to perform requests to the *Proxy*). Finally, the trusted *Template Processor* ensures that a *gather* `view` never expands into a page that can mix information from multiple *user* folders into a single request stemming from it (e.g., a form or a link). Untrusted client code in the browser is not allowed by the *Template Processor*, making *gather* `views` safe even if they contain information from multiple *user* folders (untrusted client code is allowed on all other `views`). FlowTAS exposes every container through a separate randomly-generated sub-domain. This ensures that client code cannot guess the sub-domain of other containers, that cookies are container-specific, and that the browser's SOP mechanism prevents client code from impersonating the user (e.g., to request a *gather* `view` and exfiltrate its results).

Temporal container isolation: Information of different application phases (Appendix A.4.3) can only cascade across the following folder categories (Appendix A.4.1): *application* → *settings* → *user*. This corresponds to information only cascading across the following container categories: *start* → *gather* → *enter*. Note that information can come from both folder accesses and data passed through application arguments. An *application* folder can never be written into by containers (it is only created at application installation time). A *settings* folder can only be written into by *start* containers. A *user* folder can only be written into by *gather* and *enter* containers. Finally, the trusted *Template Processor* ensures that information cannot flow from a *gather* container (a *user* folder) into the *start* container that triggered it (a *settings* folder). On the client browser side, exposing *gather* `views` as separate random sub-domains ensures information cannot flow back from a *user* folder (as part of the *gather* `view`) into a *settings* folder. The same applies to *enter* containers.

Storage and service isolation: The *Proxy* exposes management operations to containers, which are

secured through sparse capabilities (random tokens). Capabilities are conscientiously granted to containers to ensure they do not break the security properties of the system. For example, port grants allow applications to break the strict isolation of containers. *Regular* applications can only use port grants to containers running trusted *Storage* and *Service Declassifiers*. In turn, these trusted components are the only ones with port grants to the corresponding *Storage* and *Service Backends*. The integrity checks in a *Storage Declassifier* ensures folder non-interference even when *Storage Backends* are not trusted. *Service Declassifiers* ensure that a partially trusted *Service Backend* is properly configured to enforce folder non-interference before a *regular* application can interact with it (e.g., using a per-folder database view).

A.6 Containers Optimized for Multi-Execution

A.6.1 Baseline Container Implementations

We implemented our baseline containers using two different backend mechanisms: LXC [8] and SELinux [80, 108]. An additional process-based backend was also implemented as a baseline for performance comparisons (even though it cannot provide the required isolation). We did not try conventional virtual machines since previous studies show that LXC containers are more efficient [47, 97].

Container life-cycle goes through the following operations: (1) *create*; (2) *warmup*, or prepare it to run processes inside; (3) *execute* a process; (4) *cooldown*, or de-prepare it from running processes; and (5) *destroy*. Folder life-cycle simply consists on *creating* and *destroying* storage locations.

LinuX Containers (LXC) Provide lightweight OS-level virtualization containers. They are widely used as a core piece of Docker [5], a framework for application deployment. Each container corresponds to an LXC container, with its own OS namespaces (e.g., list of processes, file-system and virtual network interfaces). Processes from different containers can use the same port numbers without interfering with each other, since each container has its own IP. The *Container Manager* uses `iptables` to isolate containers from each other at the network level. *Creation* is implemented using a "clone" of a base file-system (`lxc-clone`). folder storage locations are then overlaid into the container's file-system using the appropriate access properties (read-only or read-write) using AUFS [2]. A *warmup* boots the container up (`lxc-start`) until it acquires its IP address. *Execution* spawns a process inside the container using `lxc-attach`. *Cooldown* simply stops the container (`lxc-stop`), freeing all its memory and CPU resources. Finally, *destroy* unmounts the overlaid storage locations and destroys the LXC container (`lxc-destroy`).

SELinux Implements flexible and fine-grained MAC mechanisms for the Linux kernel. Each *Container Manager* host has a minimal base policy that provides the necessary resources, and each container is implemented as a separate SELinux policy module. Policies limit what files can be accessed, and define which network ports can be used. Therefore `apps` cannot listen in ports that have not been assigned to them, and the *Container Manager* uses `iptables` to forbid network connections across containers. Binary policy modules are constructed in-flight from a textual template description. Folder creation creates and installs (`semodule -i`) a binary policy module, and sets the storage location labels. *Creation* creates a binary policy module that defines its labels, establishing the permissions of processes for that container, and *warmup* simply installs the policy module (`semodule -i`). *Execution* is more involved, since it must grant the main application's port to the container's label (`semanage port -a`), spawn a process with that label (`runcon`) and revoke the port when the process finishes (`semanage port -d`). *Cooldown* simply removes the policy module (`semodule -r`) and, finally, *destroy* removes the policy module file.

A.6.2 Intrinsic Container Performance

Our measurements show that both LXC and SELinux as the container primitive yield non-trivial overheads when compared to a non-isolated application (the baseline process backend). Even when effectively hiding their cost, each application phase requires separate containers; and containers, in turn, require separate processes because threads share memory but their code is not trusted. The following sections explore using future **capability-based architectures** to construct more efficient containers by isolating threads of the same process. We implemented an experimental capability-based container backend to demonstrate the benefits of isolation without multi-execution during *gather* operations — the same approach can also be applied to other phases as well.

In this design each thread of a single process can handle multiple folders by iteratively: (1) entering "isolated mode"; (2) handling the information of a single folder; (3) communicating its results to the *Proxy*; and (4) exiting "isolated mode". We have modified the OS and the architecture such that threads can efficiently switch into and from isolated mode, ensuring data is not leaked across threads nor across mode switches. Specifically, when executing in isolated mode, we assign each thread its own private SELinux label and file descriptor table, and we use the CODOMs capability architecture [120] to prevent threads from sharing memory with other isolated threads (thread-private memory contents are never reused across isolated phases of the same thread).

A.6.3 The CODOMs architecture

The CODOMs architecture [120] is a capability architecture [40] designed around code-centric protection domains, allowing multiple domains to coexist in the same page table. Code-centric domain isolation differs from traditional data-centric isolation by deriving the active protection domain from the instruction pointer of the executing instruction. A protection check thus verifies whether the *current instruction* can access an address, rather than whether *the current OS process* can access it. As a result, the instruction pointer serves as a capability that determines whether an instruction can access a memory address, and cross-domain calls are possible without OS intervention. This section highlights the key concepts of CODOMs relevant to FlowTAS.

CODOMs groups pages into domains by adding a per-page tag in the page table, and a separate structure defines long-term cross-domain grants. Access grants are totally ordered, and can be one of *call*, *read*, and *write*. For example, the grant "*A: (B, write)*" means that code in pages with tag A can execute code, read and write into any page with tag B (modulo the per-page protection bits). Additionally, code pages can be marked as having access to privileged processor resources. Finally, cross-domain entry points are enforced through *call* grants, which only allow routine calls at addresses aligned to a globally-configured value.

CODOMs also provides a set of dedicated capability registers that are managed by user code. As in all capability-architectures, capabilities cannot be forged and can be safely stored into memory, allowing domains to share memory at a fine granularity. In CODOMs, capabilities are created from the aforementioned cross-domain grants, and can be efficiently and selectively revoked.

A.6.4 Thread-level Containers using CODOMs

Figure A.7 shows an overview of our design. The *Proxy* creates as many threads as cores in the system (*T1* and *T2*). For a gather operation, each thread iteratively executes the untrusted *Health* code with each of its assigned folders in isolated mode (*"Fracture'13"* and *"Fever'14"*). Once a thread is in isolated mode, it has the same security guarantees of the containers described in Appendix A.3, and returning "de-isolates" it to continue executing the *Proxy* code normally. The application generates results in its per-thread memory (not

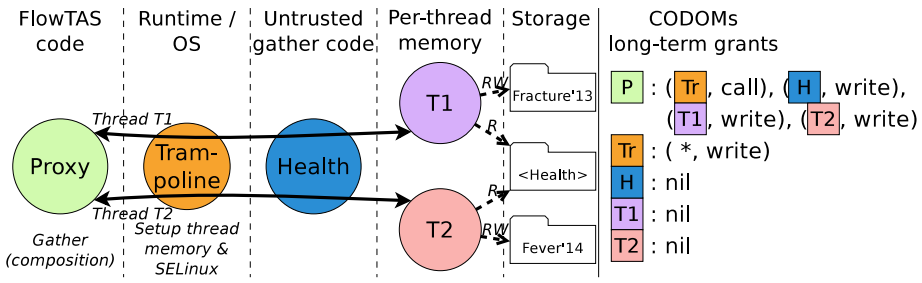


Figure A.7: Isolating two threads (*T1* and *T2*) during a gather operation using a combination of CODOMs and SELinux.

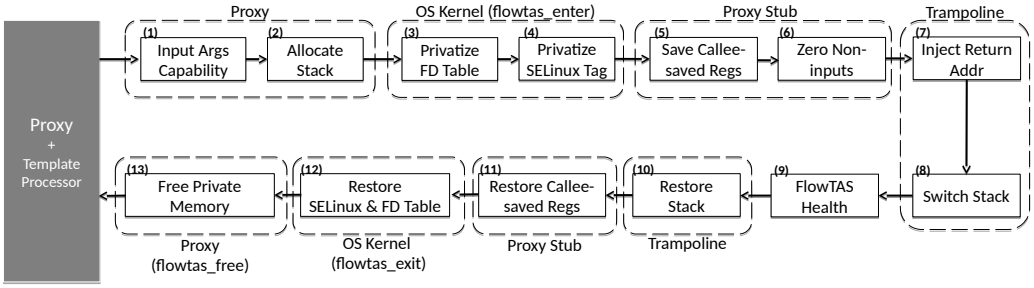


Figure A.8: Sequence of steps for a thread to execute the untrusted code of the *Health* application.

accessible by other threads), returning a pointer to these when it finishes. The *Proxy* then checks the validity of the pointer and uses its contents to render the result of the gather.

The *Proxy* manages the sequence of operations for a thread to enter and exit isolated mode through a combination of new system calls (`flowtas_enter`, `flowtas_exit` and `flowtas_free`) and a privileged *Trampoline* routine (provided by the OS). The trampoline handles the transition between executing the *Proxy* and the untrusted application code, ensuring that the latter returns with a well-known machine state. The trampoline uses a CODOMs tag that has access to all other tags (*"Tr: (*, write)"* in Figure A.7), and has access to privileged processor resources[2].

The *Proxy* is granted *call* access to the entry point of the *Trampoline* (*"P: (Tr, call)"*). The *Proxy* dynamically loads application code (using `dlopen`) into a separate tag *H* (resulting in *"P: (H, write)"*), and immediately disallows writes to these pages. Threads executing the application are thus unable to write into any domain but *H*, which has non-writable pages. The *Proxy* also creates one tag to hold the per-thread memory (resulting in *"P: (T1, write)"* and *"P: (T2, write)"*). These operations are executed only once when an application is loaded.

We now describe operations that *every thread* goes through, as depicted by Figure A.8 [3]: **(1)** Create a capability to the input arguments, which reside in the P domain, and to the per-thread memory domain ($T1$ or $T2$). **(2)** Allocate a new stack into the per-thread domain. **(3)** Privatize the file descriptor table. This ensures threads cannot share information through shared open files. **(4)** Set the per-thread SELinux label

[2] We could have implemented `flowtas_enter` and `flowtas_exit` using the same technique, but since most of their functionality is already inside the kernel, we used system calls for simplicity.

[3] The *Proxy Stub* code can be automatically generated from a declaration of the entry point function for the *Health* code.

for the target container. This ensures threads can only access resources of their assigned folders. **(5)** Save all the live callee-saved registers (and live non-output capability registers). **(6)** Zero all non-input registers. This ensures the application will not see stale values from registers (e.g., from previous iterations of the same thread). **(7)** Set the return address in the new per-thread stack to an instruction in the *Trampoline*. This ensures the application will always return through a controlled point. Since H has no access to the Tr domain, the *Trampoline* also creates a capability with *call* access to that return address. **(8)** Save the stack pointers and switch to the new stack. This ensures the application will not see stale values from the stack. **(9)** Execute the actual application code for the per-folder gather. **(10)** Restore the previous stack. This ensures the *Proxy* uses a well-known stack for its return path. **(11)** Restore the registers saved in **5**, ensuring any register tampering by the application has no effect. **(12)** Restore the previous SELinux label and file descriptor table. **(13)** Free the per-thread memory, ensuring future iterations will not read stale values left behind.

Implementation details

FlowTAS does not execute constructor and destructor functions [71] in the untrusted code *Health*, because they would run with *Proxy* privileges. It also forbids binaries with writable sections, since threads must be prohibited to write into the shared *Health* domain (H). Untrusted code must instead allocate mutable data in the heap, which is per-thread (i.e., $T1$ or $T2$). This means that untrusted code must be linked against libraries that follow that same rule. This can be applied to common libraries (e.g., `libc`), but is not feasible for every single third-party dependency.

We have also implemented an alternative version to support such cases. The untrusted code and all its dependencies are loaded into the per-thread domains (i.e., $T1$ and $T2$, we no longer use the H domain). Every thread thus gets a private replica of the target application, while the OS internally shares physical read-only memory.

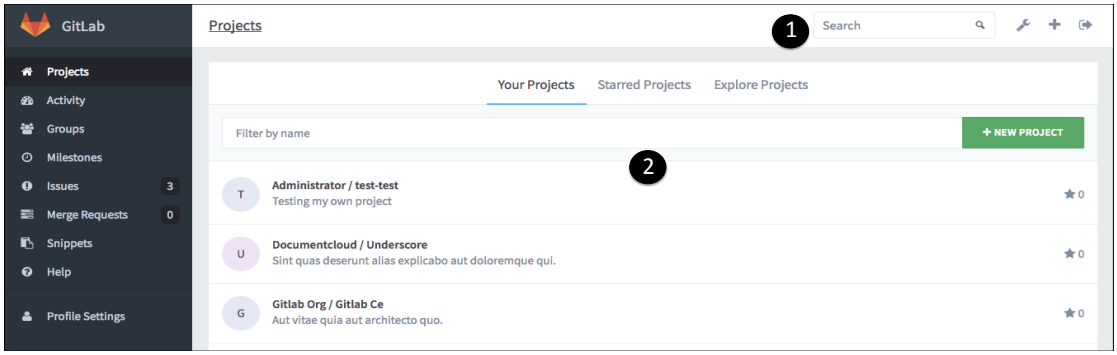
A.7 Developer Porting Experience

For the majority of applications, very few changes to the codebase are required in order to function on FlowTAS. In fact, applications that provide only per-folder functionality run on FlowTAS without any modifications. These applications, such as PDF viewers (which are notorious for vulnerabilities), document and image editors, etc. need only to provide a manifest file for launching the application (see Appendix A.4.3). These applications naturally operate on a single document in a per-folder instance.

Often times per-folder functionality is not enough, and developers will need to provide cross-folder views of data. This requires developers to make more substantial changes to their applications. Namely, developers must alter application flow, modify user management, create cross-folder templates, and use the FlowTAS storage interface. A summary of the approximate effort required to create or port each of the seven applications can be seen in Table A.1 and Table A.2. We now discuss the developer effort for porting an existing application to FlowTAS, using Gitlab as the driving example.

A.7.1 Application Design from Scratch

To best illustrate the development process to design an application from scratch for FlowTAS, we use FlowTAS Health, a mock healthcare application built upon the MEAN stack [9], as a driving example. FlowTAS Health is an encounter based sharing application for patients, physicians, and pharmacists. Encounters in



(a) A screenshot of Gitlab running on FlowTAS. This page displays the result of a gather request for all of a user's projects. The numbers correspond to the rendered HTML from template code in Figure A.9b.

1
```
<form action="/search" method="get">
    <input id="{{#FlowTAS.autocomplete}}search{{/FlowTAS.autocomplete}}">
</form>
```

2
```
{{#FlowTAS.results}} {{#projects}}
    <a class="project" href="{{#FlowTAS.enter}}/{{project_name}}{{/FlowTAS.enter}}">
    {{#FlowTAS.entercmd}}["start","--enter"]{{/FlowTAS.entercmd}}
        <div class="dash-project-avatar"> {{project_icon_letter}} </div>
        <span class="project-name"> {{FlowTAS.folder}} / {{project_name}} </span>
    </a>
    <div class="project-controls">
        <i class="fa fa-star"> {{project_star_count}} </i>
    </div>
     <div class="project-description">
        <p> {{project_description}} </p>
    </div>
{{/projects}} {{/FlowTAS.results}}
```

(b) Template code corresponding to the rendered HTML in Figure A.9a. FlowTAS provided tags are highlighted in red, while tags specific to Gitlab data are embolden in black. The code has been slightly minified for readability.

```
class Dashboard::ProjectsController < Dashboard::ApplicationController
    before_action: authenticate_user
    # This function initiates the gather and automatically returns the template
    def gather-view
        response.headers["x-flowtas-gather"] = '["start","--gather"]'
    end
    # This function responds to the gather request with the user's project list
    def gather-folder
        @projects = current_user.authorized_projects.sorted_by_activity.non_archived
        # Create the project list for the current user
        @projects_list = @projects.map do |p|
            { :project_name => p.name, :project_star_count => p.star_count, \
              :project_description => p.description, :project_icon_letter => p.name[0] }
        end
        render :json => {"projects" => @projects_list}
    end
end
```


(c) Gitlab's Project Controller for initiating and handling gather requests.

Figure A.9: Writing a cross-folder view for Gitlab.

the application correspond to either a doctor's appointment or prescription. Patients, doctors, and pharmacists have the ability to create, edit, and share encounters with one another. This type of health application was chosen as a precursor for an ongoing pilot being conducted with PATIENeT [4], a patient engagement application aimed at improving patient care.

Application By designing an application from scratch, it is possible to move as much access-control functionality out of an application to FlowTAS as possible. The first step in designing an application for FlowTAS is to reason about the minimum unit of sharing that the platform should work with. For FlowTAS Health, this could be as course grained as a patient's entire medical history, or as fined grained as a single encounter. A per-encounter level of sharing was selected because it allows an end-user to share as much, or more importantly as little, information as desired.

With the minimum unit of sharing selected, FlowTAS Health does not have to implement any of the access-control logic for the sharing of encounters. Instead, it may rely on FlowTAS to enforce a user's sharing decisions for each encounter. All requests for access to database Models in the application can pass through unchecked, trusting FlowTAS to enforce that the current user has permission to access the data. This saves developers time from reasoning about complex access-control logic, and instead allows them to focus on functionality.

User Management User authentication and password management are also no longer required to be maintained by an application. Instead, a 'filter' (a function that is called before any route handling code is run) is implemented for each URL endpoint available in the application. The filter extracts the current user passed in by FlowTAS from the headers, and sets the current session to the corresponding user. Now, users become just another attribute for an encounter or prescription instead of having their individual Model in the database (which is still possible if desired). For FlowTAS Health, this is as simple as defining an 'app.all(callback)' method with Express [5].

Gather Templates Gather endpoints and templates are required for all cross-folder views in FlowTAS Health. These pages are implemented as two overview pages for displaying encounters and prescriptions. For annotated screenshots of pages from a different application showing identical functionality, refer to Figure Figure A.9a. Navigating away from these pages requires a user to select a particular folder-backed app instance to work with, transferring control to a per-folder instance of the application.

Storage Interface Applications are free to use the backend storage of there preference. Since FlowTAS Health is written with the MEAN stack, Node.js is the database of choice. The application interfaces with the FlowTAS *Storage Declassifier* to first request access to a Node.js instance, and then finally connect (see Section Appendix A.3.5).

Development Effort In total, five applications have been developed for FlowTAS from scratch. We detail below each of the applications, and the lines of code (LOC) removed from the TCB with FlowTAS:

1. FlowTAS Health (mentioned above) was written with the MEAN stack in LOC (mostly javascript) and 25 third party libraries.
2. FlowTAS Coding is a browser based IDE written with Django in 250K LOC (mostly javascript, 1.5K of Python) and 28 third party libraries.
3. FlowTAS Calendar is written with the MEAN stack and 200K LOC (mostly javascript) with 15 third party libraries.
4. FlowTAS Images is a browser image viewing application written with the MEAN stack in 100K LOC (mostly javascript) with 16 third party libraries.

[4] https://patienet.com
[5] http://expressjs.com/api.html#app.all

Application	Framework	LOC Removed From TCB
FlowTAS Health	MEAN Stack	250,000
FlowTAS Calendar	MEAN Stack	145,000
Image Viewer	MEAN Stack	100,000
PDF Viewer	Node.js	40,000
FlowTAS Coding	Django	230,000

Table A.1: LOCs removed from the TCB with FlowTAS (including third-party libraries).

Application	Framework	LOC Added	LOC Removed	LOC Removed From TCB
Gitlab	Rails	350	50	90,000
Hacker Slides	Flask	90	100	400,000

Table A.2: LOCs we added and removed from off-the-shelf apps to enable FlowTAS functionality, and LOCs removed from the TCB with FlowTAS (including third-party libraries).

5. FlowTAS PDF is a browser PDF viewing application written with Node.js and AngularJS in 50K LOC (mostly javascript) with 1 third party library.

A.7.2 Porting Existing Applications

To examine the effort required to port an existing application to FlowTAS, we use Gitlab, a widely used web-based repository management application, as an example. We select Gitlab because it is popular among the developer community, and our closest related work [52] was only able to support plugins, rather than the full application.

Gitlab uses the Ruby on Rails Framework [16] and is based on the Model-View-Controller design pattern. The application consists of 7 primary database Models: Projects, Activities, Groups, Milestones, Issues, Merge Requests, and Snippets. It uses Redis as an in memory cache, PostgreSQL for persistent database storage, and file storage for git repositories.

Application By only adding a bash script to start Gitlab, the application component of Gitlab is usable on FlowTAS. However, functionality is curtailed due to the lack of cross-folder views. For example, a user must select the particular folder-backed application instance that contains their repository of interest. Changing to a new repository may require closing the current application instance, navigating to a new folder, and starting a new application instance. This manual switching between app instances will grow tiresome for users, making cross-folder views essential.

User Management The porting effort begins by removing user authentication from the application. For Gitlab, a simple filter is defined for all endpoints in the application. Ruby on Rails makes this task trivial by providing the 'before_action' filter for Controllers that handle HTTP requests. The application takes the current username from the request headers, creates a new user account if it does not exist, and then signs in the user as usual.

Gather Templates For maximum functionality, cross-folder templates had to be made to give cross-folder views of all database Models in the application. For example, Figure A.9a and Figure A.9b show the rendered HTML and template for the projects Model in the application. Additional gather templates are

also required when creating an instance of a new Model, to ensure the Model is created in the correct folder. Creating a gather page simply requires a slight alteration the current HTML code to include the correct Mustache tags.

Gather requests must be initiated and their gather URL endpoints specified for each of the Models mentioned in the previous paragraph. Gather requests are initiated by modifying the Model's original endpoint in the Controller to set the appropriate gather header response and returning the template (see Appendix A.4.3). A gather response is created as a function for each Model in the app, and returns a JSON list of all Model entries in the database for that particular folder. The template and application logic responsible for handling the gather operation for the projects Model can be seen in Figure A.9c.

Storage Interface Porting an application to FlowTAS means that the database must be interfaced with the *Storage Declassifier*. For storage, Gitlab uses the file system for repositories and two databases, PostgreSQL for longterm storage and Redis for an in memory cache. Repositories are simply saved in the mounted file system for each folder. PostgreSQL for the application was interfaced with FlowTAS by replacing the default PostgreSQL connection with the connection provided by the *Service Declassifier*. Redis on the other hand is not a trusted storage endpoint. Therefore, redis is accessed through the put/get interface of the *Storage Declassifier* (see Appendix A.3.5).

Appendix B

Tools

This annex describes a pair of publicly-available projects that have been developed as tools to aid in the evaluation of this thesis.

B.1 QDBI: QEMU Dynamic Binary Instrumentation

QDBI is a dynamic binary instrumentation framework built on top of QEMU [20]. QEMU is a portable emulator that is able to run code for many architectures (17 at the time of this writing) on top of almost any host system. It uses dynamic binary translation techniques, and guest code can be run as both stand-alone applications as well as full systems (a virtual machine). In fact, latest QEMU versions also serve as virtual machine managers using hardware virtualization.

Dynamic binary instrumentation is not a new topic, but QDBI offers a single interface that can perform instrumentation of code for any of the architectures supported by QEMU, as well as can instrument both stand-alone applications and full systems using the very same interface. QDBI also offers efficient mechanisms to dynamically switch between different instrumentation contents for the same guest code. This can be exploited, for example, to perform high-frequency sampling, or to instrument code only under certain conditions (e.g., instrument a specific kernel subsystem only for certain processes).

The work has still not been published academically, but the code itself is publicly available [118] and work is underway to have a large portion of it adopted in upstream QEMU.

B.2 SciExp^2: Scientific Experiment Exploration Framework

SciExp^2 (or simply *SciExp2*) stands for *Scientific Experiment Exploration*. It provides a comprehensive framework for easing the workflow of creating, executing and evaluating experiments.

The driving idea behind the framework is that of quick and effortless design-space exploration. That is, the definition and evaluation of experiments that are based on the permutation of different parameters in the design space. The framework is available in the form of Python modules, which can be integrated into larger applications, and provides a workflow broken down into three main steps: (1) defining experiments and creating the necessary files to run them, (2) facilitate experiment execution either through simple local execution or through existing job management systems, and (3) aid in the process of collecting and analyzing the results of the successfully executed experiments.

The framework started a little bit before this thesis (initially as a crude experiment automation script for a paper), but it has seen continuous improvements during the thesis and is publicly available [119].

Publications

A number of papers have been written as a result of this thesis:

- Lluís Vilanova, Muli Ben-Yehuda, Nacho Navarro, Yoav Etsion and Mateo Valero. "CODOMs: Protecting Software with Code-centric Memory Domains". In *Intl. Symp. on Computer Architecture (ISCA)*. June 2014.
- Lluís Vilanova, Marc Jordà, Diego Marrón, Nacho Navarro, Yoav Etsion and Mateo Valero. "DomOS: Fast and Secure Inter-Process Calls Using the CODOMs Capability System". *Submitted to ASPLOS 2016*.
- Lluís Vilanova, Casen Hunger, Nacho Navarro, Yoav Etsion and Mohit Tiwari. "FlowTAS: Making Data-centric Mandatory Access Control Practical". *Submitted to S&P Oakland 2016*.

Leading to this thesis, other publications have also appeared as a result of collaborations with other groups and PhD students. The following shows these publications related to the topic of this thesis:

Patents

- Yoav Etsion, Yonatan Gottesman and Lluís Vilanova. "Logical-to-Physical Block Mapping Inside the Disk Controller: Accessing Data Objects Without Operating System Intervention".

Conferences

- Carlos Villavieja, Vasileios Karakostas, Lluís Vilanova, Yoav Etsion, Alex Ramirez, Avi Mendelson, Nacho Navarro, Adrian Cristal and Osman Unsal. "DiDi: Mitigating The Performance Impact of TLB Shootdowns Using A Shared TLB Directory". In *Intl. Conf. on Parallel Arch. and Compilation Techniques (PACT)*. September 2011.

Workshops

- Isaac Gelado, Javier Cabezas, Lluís Vilanova and Nacho Navarro. "The Cost of IPC: and Architectural Analysis". In *Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA)*. June 2007.

Bibliography

[1] Amazon ec2 dedicated hosts.

[2] Another Union File-System. http://aufs.sourceforge.net.

[3] Celery. http://www.celeryproject.org.

[4] Data breach at health insurer anthem could impact millions. http://krebsonsecurity.com/2015/02/data-breach-at-health-insurer-anthem-could-impact-millions.

[5] Docker. http://docker.com.

[6] Flask. http://flask.pocoo.org.

[7] Handlebars.js: Minimal templating on steroids. http://handlebarsjs.com/.

[8] Linux Containers. http://linuxcontainers.org.

[9] MEAN Stack. mean.io.

[10] Mustache. http://mustache.github.io.

[11] Officials warn 500 million financial records hacked. http://www.wusa9.com/story/money/2014/10/21/secret-service-fbi-hack-cybersecuurity/17651619/.

[12] OpenID. http://openid.net.

[13] Pyro: PYthon Remote Objects. http://pythonhosted.org/Pyro4.

[14] Rackspace dedicated server hosting.

[15] Redis. http://redis.io.

[16] Ruby on Rails. http://rubyonrails.org/.

[17] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. In *Symp. on Operating Systems Design and Implementation (OSDI)*, October 2012.

[18] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis. Dune: Safe user-level access to privileged cpu features. In *Symp. on Operating Systems Design and Implementation (OSDI)*, Oct. 2012.

[19] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *Symp. on Operating Systems Design and Implementation (OSDI)*, Oct. 2014.

[20] F. Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conf.*, 2005.

[21] M. Ben-Yehuda, J. Xenidis, M. Ostrowski, K. Rister, A. Bruemmer, and L. V. Doorn. The price of safety: Evaluatin IOMMU performance. In *Linux Symposium*, 2007.

[22] J. Bernabeu-Auban, P. Hutto, and Y. Khalidi. The architecture of the Ra kernel. Technical report, Georgia Institute of Technology, 1988.

[23] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the SPIN operating system. In *ACM Symp. on Operating Systems Principles (SOSP)*, 1995.

[24] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Computer Architecture News*, August 2011.

[25] S. Biswas, D. Franklin, A. Savage, R. Dixon, T. Sherwood, and F. T. Chong. Multi-execution: Multicore caching for data-similar executions. In *Intl. Symp. on Computer Architecture (ISCA)*, June 2009.

[26] S. Biswas, D. Franklin, A. Savage, R. Dixon, T. Sherwood, and F. T. Chong. Multi-execution: Multicore caching for data-similar executions. In *Intl. Symp. on Computer Architecture (ISCA)*, June 2009.

[27] R. H. Campbell, N. Islam, D. Raila, and P. Madany. Designing and implementing Choices: An object-oriented system in C++. *Comm. ACM*, Sept. 1993.

[28] N. P. Carter, S. W. Keckler, and W. J. Dally. Hardware support for fast capability-based addressing. In *Intl. Conf. on Arch. Support for Programming Languages & Operating Systems (ASPLOS)*, October 1994.

[29] M. Castro, M. Costa, and J.-P. Martin. Better bug reporting with better privacy. In *Intl. Conf. on Arch. Support for Programming Languages & Operating Systems (ASPLOS)*, March 2008.

[30] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black. Fast byte-granularity software fault isolation. In *ACM Symp. on Operating Systems Principles (SOSP)*, 2009.

[31] D. Champagne and R. B. Lee. Scalable architectural support for trusted software. In *Symp. on High-Performance Computer Architecture (HPCA)*, Jan 2010.

[32] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska. Sharing and protection in a single address space operating system. In *IEEE Trans. on Computers*, May 1994.

[33] D. R. Cheriton, G. R. Whitehead, and E. W. Sznyter. Binary emulation of UNIX using the V kernel. In *USENIX Summer Conf.*, June 1990.

[34] D. Chisnall, C. Rothwell, R. N. Watson, J. Woodruff, M. Vadera, S. W. Moore, M. Roe, B. Davis, and P. G. Neumann. Beyond the PDP-11: Architectural support for a memory-safe C abstract machine. In *Intl. Conf. on Arch. Support for Programming Languages & Operating Systems (ASPLOS)*, March 2015.

[35] R. K. Clark, E. D. Jensen, and F. D. Reynolds. An architectural overview of the Alpha real-time distributed kernel. In *USENIX Workshop on Micro-kernels and Other Kernel Architectures*, Apr 1992.

[36] B. Cobb, C. Garcia-Arellano, and S. Kamath. *IBM DB2 Version 9.5 memory protection using System p storage protection keys*. IBM, Nov 2007.

[37] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: a flexible information flow architecture for software security. In *Intl. Symp. on Computer Architecture (ISCA)*, May 2007.

[38] Dell. Dell DVD store database test suite. `http://linux.dell.com/dvdstore`.

[39] D. E. Denning. A lattice model of secure information flow. In *ACM Symp. on Operating Systems Principles (SOSP)*, November 1975.

[40] J. B. Dennis and E. C. V. Horn. Programming semantics for multiprogrammed computations. *Comm. ACM*, March 1966.

[41] U. Drepper. *ELF Handling For Thread-Local Storage*. Red Hat Inc., Feb 2003.

[42] C. Dwork. Differential privacy. In *International Colloquium on Automata, Languages and Programming (ICALP)*, 2006.

[43] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. Labels and Event Processes in the Asbestos Operating System. ACM, 2005.

[44] D. Engler, M. Kaashoek, and J. O'Toole. Exokernel, an operating system architecture for application-level resource management. In *ACM Symp. on Operating Systems Principles (SOSP)*, 1995.

[45] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: software guards for system address spaces. In *OSDI*, 2006.

[46] A. J. Feldman, A. Blankstein, M. J. Freedman, and E. W. Felten. Social networking with frientegrity: privacy and integrity with an untrusted provider. In *USENIX Security Symposium*, Security'12, pages 31–31, 2012.

[47] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An updated performance comparison of virtual machines and linux containers. Technical report, IBM Research Division, Jul 2014.

[48] B. Ford and R. Cox. Vx32: lightweight user-level sandboxing on the x86. In *USENIX Annual Technical Conf.*, 2008.

[49] B. Ford and J. Lepreau. Evolving Mach 3.0 to a migrating thread model. In *USENIX Technical Conf.*, 1994.

[50] S. Forrest, A. Somayaji, and D. Ackley. Building diverse computer systems. In *Hot Topics in Operating Systems (HotOS)*, May 1997.

[51] J. Fotheringham. Dynamic storage allocation in the Atlas computer, including an automatic use of a backing store. *Comm. ACM*, October 1961.

[52] D. B. Giffin, A. Levy, D. Stefan, D. Terei, J. Mitchell, D. Mazières, and A. Russo. Hails: Protecting data privacy in untrusted web applications. In *Symp. on Operating Systems Design and Implementation (OSDI)*, October 2012.

[53] D. Golub, R. Dean, A. Forin, and R. Rashid. Unix as an application program. In *USENIX Summer Conf.*, June 1990.

[54] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[55] C. Gray, M. Chapman, P. Chubb, D. Mosberger-Tang, and G. Heiser. Itanium — a system implementor's tale. In *USENIX Annual Technical Conf.*, April 2005.

[56] M. Guillemont. The Chorus distributed operating system: Design and implementation. In *ACM Intl. Symp. on Local Computer Networks*, 1982.

[57] G. Hamilton and P. Kougiouris. The Spring nucleus: a microkernel for objects. In *USENIX Summer Conf.*, Jun 1993.

[58] G. Heiser, K. Elphinstone, S. Russell, and J. Vochteloo. Mungi: A distributed single-address-space operating system. In *Australasian Computer Science Conf. (ACSC)*, Jan 1994.

[59] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. MINIX 3: a highly reliable, self-repairing operating system. In *SIGOPS Operating Systems Review*, Jul 2006.

[60] G. C. Hunt and J. R. Larus. Singularity: rethinking the software stack. *SIGOPS Operating Systems Review*, 2007.

[61] IBM. *Power ISA ™*, version 2.06 revision B edition, July 2010.

[62] Intel. *Intel Software Guard Extensions Programming Reference*, October 2014.

[63] Internet Engineering Task Force. *The OAuth 2.0 Authorization Framework*, Oct 2012.

[64] M. L. Johnson, S. Egelman, and S. M. Bellovin. Facebook and privacy: it's complicated. In *Symp. On Usable Privacy and Security (SOUPS)*, July 2012.

[65] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. B. no, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on exokernel systems. In *ACM Symp. on Operating Systems Principles (SOSP)*, 1997.

[66] A. Kivity, D. Laor, G. Costa, P. Enberg, N. Har'El, D. Marti, and V. Zolotarov. OSv — optimizing the operating system for virtual machines. In *USENIX Annual Technical Conf.*, Jun 2014.

[67] E. J. Koldinger, J. S. Chase, and S. J. Eggers. Architectural support for single address space operating systems. In *Intl. Conf. on Arch. Support for Programming Languages & Operating Systems (ASPLOS)*, 1992.

[68] M. Krohn. *Information Flow Control for Secure Web Sites*. PhD thesis, MIT, 2008.

[69] B. W. Lampson. Protection. *SIGOPS Operating Systems Review*, January 1974.

[70] S. Lee, E. L. Wong, D. Goel, M. Dahlin, and V. Shmatikov. πBox: A platform for privacy-preserving apps. In *USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, April 2013.

[71] J. R. Levine. *Linkers and Loaders*. Morgan Kaufmann, 1999.

[72] H. M. Levy. *Capability-Based Computer Systems*. Digital Press, 1984.

[73] C. Li, C. Ding, and K. Shen. Quantifying the cost of context switch. In *ACM Workshop on Experimental Computer Science (ExpCS)*, Jun 2007.

[74] N. Li, T. Li, and S. Venkatasubramanian. t-closeness: Privacy beyond k-anonymity and l-diversity. In *Intl. Conf. on Data Engineering (ICDE)*, April 2007.

[75] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. 2009.

[76] J. Liedtke. A persistent system in real use — experiences of the first 13 years. In *Intl. Workshop on Object Orientation in Operating Systems (IWOOOS)*, 1993.

[77] J. Liedtke. Improved address-space switching on Pentium processors by transparently multiplexing user address spaces. Technical report, German National Research Center for Information Technology, 1995.

[78] J. Liedtke. On microkernel construction. In *ACM Symp. on Operating Systems Principles (SOSP)*, 1995.

[79] W. Lonergan and P. King. Design of the B5000 system. *DATAMATION*, May 1961.

[80] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the linux operating system. In *USENIX Annual Technical Conf.*, 2001.

[81] A. Machanavajjhala, D. Kifer, J. Gehrke, and M. Venkitasubramaniam. L-diversity: Privacy beyond k-anonymity. *ACM Trans. Knowl. Discov. Data*, 2007.

[82] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: library operating systems for the cloud. In *Intl. Conf. on Arch. Support for Programming Languages & Operating Systems (ASPLOS)*, Apr 2013.

[83] T. Maeda. Kernel mode linux. *Linux Journal*, May 2003.

[84] F. McSherry. Privacy integrated queries. In *SIGMOD*, June 2009.

[85] P. Mohan, A. Thakurta, E. Shi, D. Song, and D. E. Culler. GUPT: Privacy preserving data analysis made easy. In *SIGMOD*, May 2012.

[86] S. J. Mullender. The Amoeba distributed operating system: selected papers, 1984-1987. Technical report, Centrum voor Wiskunde en Informatica, 1987.

[87] Open Market. FastCGI. http://www.fastcgi.com.

[88] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: the case of AES. In *RSA conference on Topics in Cryptology*, February 2006.

[89] B. Parno, J. M. McCune, and A. Perrig. Bootstrapping trust in commodity computers. In *IEEE Symp. on Security and Privacy*, 2010.

[90] PaX Team. *PaX Address Space Layout Randommization*. http://pax.grsecurity.net/docs/aslr.txt.

[91] C. Pollock. The mobile app top 10 risks. https://www.owasp.org/images/9/94/MobileTopTen.pdf. page 21.

[92] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. Hunt. Rethinking the library OS from the top down. In *Intl. Conf. on Arch. Support for Programming Languages & Operating Systems (ASPLOS)*, Mar 2011.

[93] V. Rastogi and S. Nath. Differentially private aggregation of distributed time-series with transformation and encryption. In *SIGMOD*, June 2010.

[94] D. Redell, Y. Dalal, T. Horsley, H. Lauer, W. Lynch, P. McJones, H. Murray, and S. Purcell. Pilot: An operating system for a personal computer. In *Comm. ACM*, Feb 1980.

[95] B. F. Romanescu, A. R. Lebeck, D. J. Sorin, and A. Bracy. UNified Instruction/Translation/Data (UNITD) coherence: One protocol to rule them all. In *Symp. on High-Performance Computer Architecture (HPCA)*, Jan 2010.

[96] I. Roy, S. T. Setty, A. Kilzer, V. Shmatikov, and E. Witchel. Airavat: Security and privacy for mapreduce. In *USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, April 2010.

[97] B. Russell. KVM and Docker LXC Benchmarking with OpenStack. http://bodenr.blogspot.com/2014/05/kvm-and-docker-lxc-benchmarking-with.html.

[98] J. Rutkowska and R. Wojtczuk. Qubes OS architecture. Technical report, Invisible Things Lab, January 2010.

[99] A. Sabelfeld, A. C., and Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, January 2003.

[100] J. H. Saltzer. Protection and the control of information sharing in Multics. In *Comm. ACM*, July 1974.

[101] Samsung. Knox. https://www.samsungknox.com.

[102] M. D. Schroeder and J. H. Saltzer. A hardware architecture for implementing protection rings. *Comm. ACM*, March 1972.

[103] D. Schultz and B. Liskov. IFDB: Decentralized information flow control for databases. In *European Conference on Computer Systems (EuroSys)*, April 2013.

[104] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen. Adapting software fault isolation to contemporary CPU architectures. In *USENIX Security*, 2010.

[105] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: a fast capability system. In *ACM Symp. on Operating Systems Principles (SOSP)*, December 1999.

[106] A. Shriraman and S. Dwarkadas. Sentry: Light-weight auxiliary memory access control. In *Intl. Symp. on Computer Architecture (ISCA)*, June 2010.

[107] S. Smalley and R. Craig. Security enhanced (SE) Android: Bringing flexible MAC to android. In *NDSS Symposium*, A 2013.

[108] S. Smalley, C. Vance, and W. Salamon. Implementing SELinux as a Linux Security Module. Technical report, NAI Labs, Dec 2001.

[109] E. Stefanov, M. van Dijk, A. Juels, and A. Oprea. Iris: a scalable cloud file system with efficient integrity checks. In *Annual Computer Security Applications Conference (ACSAC)*, 2010.

[110] R. Strackx and F. Piessens. Fides: selectively hardening software application components against kernel-level or process-level malware. In *ACM Conf. on Computer & Communications Security (CCS)*, 2012.

[111] L. Sweeney. k-anonimity: A model for protecting privacy. *International Journal on Uncertaint, Fuzziness and Knowledge-based Systems*, 2002.

[112] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. In *ACM Symp. on Operating Systems Principles (SOSP)*, 2003.

[113] D. Swinehart, P. Zellweger, R. Beach, and R. Hagmann. A structural view of the Cedar programming environment. In *ACM Trans. on Programming Languages and Systems*, Oct 1986.

[114] I. Taylor. *Split Stacks in GCC*. http://gcc.gnu.org/wiki/SplitStacks.

[115] M. Tiwari, H. Wassel, B. Mazloom, S. Mysore, F. Chong, and T. Sherwood. Complete information flow tracking from the gates up. In *Intl. Conf. on Arch. Support for Programming Languages & Operating Systems (ASPLOS)*, March 2009.

[116] D. Tsafrir. The context-switch overhead inflicted by hardware interrupts (and the enigma of do-nothing loops). In *ACM Workshop on Experimental Computer Science (ExpCS)*, Jun 2007.

[117] B. C. Vattikonda, S. Das, and H. Shacham. Eliminating fine grained timers in Xen (short paper). In T. Ristenpart and C. Cachin, editors, *Proceedings of CCSW 2011*. ACM Press, Oct. 2011.

[118] L. Vilanova. QDBI: QEMU dynamic binary instrumentation. https://projects.gso.ac.upc.edu/projects/qemu-dbi.

[119] L. Vilanova. SciExp^2. https://projects.gso.ac.upc.edu/projects/sciexp2.

[120] L. Vilanova, M. Ben-Yehuda, N. Navarro, Y. Etsion, and M. Valero. CODOMs: Protecting software with code-centric memory domains. In *Intl. Symp. on Computer Architecture (ISCA)*, June 2014.

[121] C. Villavieja, V. Karakostas, L. Vilanova, Y. Etsion, A. Ramirez, A. Mendelson, N. Navarro, A. Cristal, and O. Unsal. DiDi: Mitigating the performance impact of TLB shootdowns using a shared TLB directory. In *Intl. Conf. on Parallel Arch. and Compilation Techniques (PACT)*, September 2011.

[122] N. H. Walfield. Fast capability transfer using existing commodity hardware. Unpublished (http://walfield.org/papers/200804-bachmann-fast-capability-transfer.pdf), 2008.

[123] R. N. M. Watson, J. Anderson, B. Laurie, and K. Kennaway. Capsicum: practical capabilities for UNIX. In *USENIX Security Symposium*, 2010.

[124] E. Witchel, J. Cates, and K. Asanović. Mondrian memory protection. In *Intl. Conf. on Arch. Support for Programming Languages & Operating Systems (ASPLOS)*, October 2002.

[125] E. Witchel, J. Rhee, and K. Asanović. Mondrix: memory isolation for Linux using mondriaan memory protection. In *ACM Symp. on Operating Systems Principles (SOSP)*, 2005.

[126] J. Woodruff, R. N. M. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe. The CHERI capability model: Revisiting RISC in an age of risk. In *Intl. Symp. on Computer Architecture (ISCA)*, June 2014.

[127] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: a sandbox for portable, untrusted x86 native code. *Comm. ACM*, January 2010.

[128] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek. Improving application security with data flow assertions. In *ACM Symp. on Operating Systems Principles (SOSP)*, October 2009.

[129] S. Zdancewic and A. C. Myers. Robust declassification. In *IEEE Computer Security Foundations Workshop*, 2001.

[130] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making Information Flow Explicit in HiStar. In *Symp. on Operating Systems Design and Implementation (OSDI)*, 2006.

Acronyms

ABI Application Binary Interface.

ACL Access Control List.

APL Access Protection List.

ASLR Address Space Layout Randomization.

CAM Content-Addressable Memory.

DCS Domain Capability Stack.

HPC High-Performance Computing.

IFC Information Flow Control.

ILP Instruction-Level Parallelism.

IPC Inter-Process Communication.

IPI Inter-Processor Interrupt.

KCS Kernel Control Stack.

KML Kernel Mode Linux.

MAC Mandatory Access Control.

MMP Mondriaan Memory Protection.

MVC Model-View-Controller.

OS Operating System.

PIC Position-Independent Code.

PLB Protection Lookaside Buffer.

PLT Procedure Linkage Table.

RAW Read-After-Write.

RPC Remote Procedure Call.

SASOS Single Address Space OS.

SFI Software Fault Isolation.

SOP Same-Origin Policy.

TCB Trusted Computing Base.

TLB Translation Lookaside Buffer.

TLS Thread-Local Storage.

TUI Trusted User Interface.