Object-Capability Programming Languages on the seL4 Capability-based Microkernel

Stewart Webb
Student ID: 584957
Email: sjwebb@student.unimelb.edu.au

Supervised by
Toby Murray

Thesis submitted to

The University of Melbourne

in partial fulfilment of the
requirements for the degree of

Master of Science (Computer Science)

School of Computing and Information Systems
The University of Melbourne

November 2022

Declaration of Authorship

I certify that

- this thesis does not incorporate without acknowledgement any material previously submitted for a degree or diploma in any university; and that to the best of my knowledge and belief it does not contain any material previously published or written by another person where due reference is not made in the text.
- where necessary I have received clearance for this research from the University's Ethics Committee and have submitted all required data to the School (N/A)
- this thesis is fewer than 25,000 words in length (excluding text in figures, table, bibliographies and appendices).

Acknowledgements

I would like to thank:

- Toby Murray, my project supervisor, for providing me the space within which to take on a systems project at Melb Uni, and all his time, expertise, and good conversation during our catchup meetings across the timeframe of the project.
- My fellow course-mates:
  - Matt Farrugia-Roberts, whose support, 'Shut-up-and-write' session organising, and encouragement throughout the final thesis process was extremely helpful and valuable for getting everything I had worked on down into this thesis.
  - James Barnes and Amy Mendelsohn, for their friendship and support in trying to make and support a social environment within the Master of Computer Science degree (and Amy's help in practising and preparing the presentation component of the project),
  - Eleanor McMurtry, Ben Frengley, Isitha Piyumal Subasinghe, and again James, for their classmate chummery and extensive conversations on programming and computer science topics.
- Gernot Heiser and the UNSW Trustworthy Systems group for the initial project idea, and again, a context in Australia to perform systems research, and Kent McLeod for answering so many of my questions via the seL4-external Mattermost chat.
- Timothy Roscoe, David Cock, and Daniel Schwyn from the ETH Zurich Systems group, for running the Advanced Operating Systems subject that I was lucky enough to be able to take on exchange in 2019, which served as my leg-up into systems research, and without which I would never have been practically able to work on anything covered in this thesis.

- My classmates from the Advanced OS class project in Zurich - Silvan Läubli, Cédric Neukom, and Benjamin Schmid, whose C and algorithms expertise and hard work were invaluable for making it through to the end of the subject.
- Sylvan Clebsch, whose thesis and work on Pony ended up serving as the basis for a large component of this project, and all the other Pony developers, including Sean T Allen for his very helpful responses in the Pony Zulip chat.
- Mark S. Miller, for his extensive research on object-capability languages that much of this project was based on, and to him and other members of the `cap-talk` Google Groups mailing list for discussion and resources relating to object-capability languages.
- Ryan Crosby and Harry Ramadan from Unique Micro Design for their offers of proof-reading and subsequent feedback, and keeping me employed throughout the duration of my degree - and also to Ryan for many productive chats about OS and PL topics.
- Last but not least, my family and friends for all their support throughout what was a very difficult degree and time amidst completing this project.

Contents

List of Figures **vi**

Abstract **1**

1 seL4 + OS Capabilities Background **3**

1.1 Operating System Capabilities 4
1.2 seL4 System Calls and Programming Model 7
1.3 seL4 Capabilities 9
1.4 seL4 Protection Domains 12
1.5 seL4 IPC and blocking semantics 14
1.6 seL4 boot process + root task capability bootstrap 15
1.7 'Memory Server' example 18

2 Object-Capability Language background **21**

2.1 Confined Execution 22
2.2 Vats and remote objects 23

3 Project Motivation and current seL4 tooling review **24**

4 Survey of Object-Capability Languages and appropriateness for mapping **27**

4.1 E 28
4.2 Jessie / Secure EcmaScript 30
4.3 SHILL 32
4.4 Dala 33
4.5 Pony 34
4.6 Other related art 35
4.6.1 Rust - `cap-std` and `ferros` crates 35

4.6.2 WebAssembly + WebAssembly System Interface (WASI) . 36
4.6.3 Microsoft Singularity Project 38

5 Pony Background **39**
5.1 The Pony language + execution model 40
5.2 Compilation model and `libponyrt` runtime 42
5.3 Runtime components and API 44
5.4 Pony capabilities 46
5.5 Standard Library authorities 48
5.6 Sample Pony program 51
5.7 'Causal' messaging 53
5.8 'Distributed Pony' 54

6 Comparison of related concepts in Pony and seL4 **55**
6.1 Synchronous v.s. asynchronous models 56
6.2 Message-passing message size 59
6.3 Pony allocation sizes v.s. seL4 object sizes 60
6.4 Memory Address spaces 61
6.5 Capability enforcement / Trust boundaries 62
6.6 API contracts over message-passing channels 65
6.7 Authority for memory allocation 69

7 Possible useful Pony ocap models for seL4 programming **70**
7.1 Handing off/around seL4 IPC endpoint(s) for talking to objects . . 71
7.2 Remote actor communication through message pump endpoint . 72
7.3 Handing off physical memory that contains data 76
7.4 Embed seL4 capability types into Pony types 77

8 Porting the Pony runtime environment to seL4 **78**
8.1 Which runtime components to port first? Analysis of the `main()` procedure of a Pony program 79
8.2 Step 1: base seL4 environment 83
8.3 Step 2: porting the allocator 86
8.4 Step 3: porting the SPMC messages queues 94
8.5 Future steps: scheduling, actor heaps, garbage collection 98

9 Conclusions **99**

List of Figures

1.1 A simple example of capability-based resource control: file descriptors on Linux 4
1.2 seL4 System Call / Kernel API. Adapted from the seL4 manual [66], with simplified argument names and `out` keywords to indicate arguments that the call returns output data through 8
1.3 (Non-exhaustive) Overview of different capability types on seL4 . 9
1.4 (Non-exhaustive) Overview of various API calls that can be made against the capability types of Figure 1.3 10
1.5 seL4 CNode methods with abbreviated argument pseudo types. Adapted from §3.1.2 and §10.3.1 of the seL4 manual [66] 11
1.6 Fixed/'well-known' capability addresses for capabilities set up for the root task as part of the seL4 boot process. Sourced from `libsel4/include/sel4/bootinfo_types.h` in the seL4 kernel codebase 16
1.7 C types showing the layout of data within the seL4 'bootinfo' struct passed to the root task of the system. Sourced from `libsel4/include/sel4/bootinfo_types.h` in the seL4 kernel codebase 17
1.8 `MemoryInterface` CAmkES component and interface from Google KataOS. The format of `char request[]` is detailled in Figure 1.9 Sourced from the KataOS repository `https://github.com/AmbiML/sparrow-kata-full/tree/main/apps/system/components/MemoryManager` 19
1.9 Assorted snippets of the MemoryManager component from KataOS, showing its actual Rust-typing interface. Sourced from the KataOS repository `https://github.com/AmbiML/sparrow-kata-full/tree/main/apps/system/components/MemoryManager` 20

3.1 Sample capDL spec Sourced from the capDL documentation at `https://docs.sel4.systems/projects/capdl/` 25

5.1 Overview of Pony actor messaging at runtime 41
5.2 Overview of the Pony compiler, runtime, and output programs . . 42
5.3 `libponyrt` runtime functions 44
5.4 Components of the Pony runtime Adapted from Figure A.1 of ClebschThesis 44

5.5 Authority primitives and assorted base types for the `net` package of the Pony standard library. Adapted from the Pony standard library sources (`packages/net` in [62]) 49
5.6 Authority primitives for the `files` package of the Pony standard library. The `FilePath` class is shown with its constructor to illustrate that the `FileAuth` capability is required as an argument for interaction with the filesystem. Adapted from the Pony standard library sources (`packages/files` in [62]) 50
5.7 Authority primitives for the `serialise` package of the Pony standard library. The constructor and various methods of the `Serialised` class are shown to illustrate that capabilities are required for the various serialisation operations. Adapted from the Pony standard library sources (`packages/serialise` in [62]) 50

6.1 Pony runtime's message-queue push function, used for all message passing Adapted into psuedo-algorithmic form from `messageq_push` in `libponyrt/actor/messageq.c` in the `ponyc` repo [62] 56
6.2 Example CAmkES spec for a DHCP server, showing the C-like API defined in the `DHCP` procedure block. Adapted from `camkes/apps/dhcp` in the main / example apps CAmkES repository (`https://github.com/seL4/camkes`) 66
6.3 Examples of the infix `!` / "eventually" operator from Dr. SES and its '`Q`' library. Taken directly from [47, §2.4, page 7] 67

7.1 Example of seL4 threads and endpoints required for cross-domain message queue message-passing support for a quad-core, quad-domain Pony runtime 75

8.1 Disassembly of the C `main()` procedure of a compiled Pony program 80
8.2 Description of the various levels of memory space sources used in the Pony pool allocator Adapted from code in `mem/pool.c` in the `libponyrt` folder of the `ponyc` repo [62] 87
8.3 Existing `ponyint_virt_alloc` implementation from the release Pony runtime Taken from `libponyrt` in the `ponyc` repo [62] . . . 90
8.4 `ponyint_virt_alloc` implementation developed for the seL4 environment 92

Abstract

The seL4 microkernel provides one of the highest levels of security assurances possible for an operating system via its machine-checked proofs of properties such as integrity, authority confinement, and information-flow non-interference. The key mechanism underlying these proofs and assurances is its capability system for resource management, which is used to describe all memory and communication channels between user-level components of an seL4 system.

Object Capability programming languages make similar use of a capability security model for expressing authority propagation throughout code. In these languages, object references are viewed as the capabilities, such that having a reference to an object represents the rights to use and control that object, with the assumption that object references cannot be invented or forged (for example by dereferencing arbitrary memory addresses). Security mechanisms and policies can become expressed through object-oriented programming patterns such as object proxies, and execution contexts usually specify ways to be launched with initial capabilities, or channels to obtain subsequent capabilities via once the program starts.

Whilst many tools have been developed for building systems with seL4 such as capDL and CAmKES, they rely on static distribution of capabilities upfront at project build time, and in general do not provide for dynamic capability transfer once the system has started. In practice, for runtime/dynamic access control, many systems revert to standard POSIX layers where ambient authority for resource access returns, throwing out the fine-grained access control possibilities that come for free with a capability model. In theory, an object capability language would provide a better means for building dynamic capability systems, especially if such a language can have seL4 capabilities for kernel objects mapped into it.

Many Object Capability languages exist (E, Oz-E, Jessie/SES, SHILL, Pony, Dala) but their purposes/research goals are varied and their implementations can

rest on many layers of software. This research surveys various object capability languages from the research community and evaluates their appropriateness for mapping to seL4. The Pony language is examined as a frontrunner for an implementation due to its C-like performance and relatively minimal set of language implementation dependencies. However many trade-offs are still involved due to the minimal mechanisms provided by seL4 (compared to more prevalent monolithic kernels), which are explored with an aim to more formally detail and compare the properties and limitations of both seL4 and object capability language capability models and mechanisms.

Chapter 1

seL4 + OS Capabilities Background

As this thesis focuses primarily on the seL4 microkernel and its capability subsystem, some background on both must be given first.

1.1 Operating System Capabilities

Capabilities are an old operating systems concept for providing a security model for resource access control, stemming back to an initial definition from 1965 [22], and come with a rich lineage of associated concepts and research. They center around the idea of 'unique, unforgeable tokens' used to represent both the access to a resource, and the type of access rights granted. If the tokens are unforgeable, then security is maintained, as subjects cannot gain access to any resource unless they were explicitly provided a capability for the resource.

A simple example of how capabilities are used in an operating system context can be found in the form of file descriptors from Linux/UNIX systems. Figure 1.1 shows an example of this. Opening a file grants a process a file descriptor number, which is used as the 'token' for all operating system operations (i.e. system calls) relating to that file - e.g. the `read()` system call. This 'token' can never be forged, because the kernel always checks that the passed number is indeed a valid file descriptor number, by checking against its file descriptor table, which is a protected kernel-space datastructure.

```
#include <fcntl.h>
int fd;
char buf[50];

fd = open("test.txt", O_RDONLY);
int bytes_read = read(fd , buf, 10);
```

Figure 1.1: A simple example of capability-based resource control: file descriptors on Linux

This Linux example however also highlights another key issue often discussed in the context of capability models - the issue of *ambient authority*. In the example above, the program has an implicit ability to interact with the entire filesystem and acquire rights to files due to `open()` being an implicit global that does not operate using any capability arguments, or rather that the path argument is simply the *name* of a file, and not anything that designates *authority* - i.e. anything to do with the program's *right* to use a file. Instead, the source of authority is delegated to the user running the program, as on Linux/UNIX systems, file access permissions are dictated by Access Control Lists which are evaluated against the UID of the running process. This consequently means that the program has launched with the implicit ability to access *any* file the running user has access to. Requesting

Linux programs to operate on specific files is often done via command-line arguments that are again just object names, but these do not constitute *capabilities* that denote the *right* to use a file.

The presence of ambient authority invariably leads to the possibility of the 'Confused Deputy' problem [34], where programs can easily be tricked into performing operations that the program author or user might not have expected should be possible. The Confused Deputy paper in particular focuses on an example involving file name arguments similar to that outlined above. Capability-based systems by nature of their construction eliminate this category of problem by making the rights for resource usage inherently more explicit.

Capabilities and the avoidance of ambient authority also give way to another related security concept called the *Principle of Least Authority* - the idea that a program should only ever be given the absolute minimum amount of rights to accomplish the task it is being invoked to do, and this is a important idea central to seL4's security model. A good more common modern example of this is the advent of increasingly granular permission controls being developed within mobile operating systems such as iOS and Android - in earlier versions, apps often had to be given access to full photo libraries or even the whole filesystem [1] to access a users camera photos, whereas it is increasingly more common for photo access to be granted by a trusted OS 'file picker' component, which in the end only hands the running application the right to operate on one specific file that the user themselves selected.

Several capability-based operating systems / kernels have been created throughout the years, including KeyKOS, EROS, seL4, and Barrelfish. There are also a few different ways capabilities can be implemented, as outlined in [49, §1.2]:

- **Tagged Capabilities**: where the hardware or execution environment maintains capability data within actual user-level references, but mandates through controlled access that the capability-related metadata can never be altered in an unauthroized manner
- **Partitioned Capabilities**: where the kernel stores the actual data associated with capabilities in a protected storage area, and acts as the trusted arbitrator over them, checking that user-level references are correct every time they are invoked. This is the model that Barrelfish and seL4 use, and that is shown by the Linux file descriptor example above (**??**)

[1] see notes on `READ_EXTERNAL_STORAGE` and `WRITE_EXTERNAL_STORAGE` deprecation in [52]

- **Sparse Capabilities**: where capabilities are essentially just random numbers in an very large number space, and 'unforgeable' in the practical sense that it is infeasible to guess any valid capability number.

The Barrelfish OS in particular contains a capability system derived from seL4 itself [49, Abstract, §1.3], that is also more generalised, as capability types themselves are defined using a domain-specific language called Hamlet [21]. On seL4 and Barrelfish, as part of their style of being 'microkernel' operating systems where as much implementation logic is pushed out to userspace as possible, capabilities are used as the basis for describing *all* resources in the system, including physical and virtual memory.

For this project, the focus is specifically on the current seL4 microkernel and its particular capability system. Background on seL4 relevant for the rest of the thesis will be covered in the next sections.

1.2 seL4 System Calls and Programming Model

For any operating system kernel, the main interface through which the system is controlled is the *system call* interface. For seL4, the system call API is modelled such that almost *every* call is a *capability invocation* - and thus the system call API is the primary means for interacting with seL4 capabilities.

Kernels typically start up by executing some one-time initializing code that will scan and initialize the available system hardware as required, and set up kernel datastructures in main memory, to support the kernel's operation and execution environment, before finally constructing a first 'root' userspace process/task/thread that control is finally handed over to. At this point the kernel can be considered 'booted', and what happens next will mostly depend on the code of the first process and how it interacts with the kernel through the system call interface, with interrupts from hardware (including any hardware timers used for pre-emptive scheduling) being the only main other thing that can cause any kernel behaviour to kick in again.

System calls can be implemented in a number of different ways, but are usually performed by setting values into a few specific CPU registers to identify the particular system call wanted, and the arguments to pass to it, before issuing a 'trap' or 'syscall' CPU instruction, to cause the CPU to switch into kernel mode and call the appropriate syscall handling code within the kernel that was set up at boot time.

A survey of the Linux system call API at the time of writing shows 313 different numbered system calls.

seL4 in contrast has only *eight* system calls, as part of its minimalist microkernel design (see Figure 1.2).

Whilst the seL4 system call API only lists a small number of calls, control of the system is described more by the fact that `Send()` / `Call()` family of calls are handled differently depending on the type of capability being addressed. Using `Send()` or `Call()` on a capability is termed 'invoking' the capability, and 'methods' for capability types are defined as part of the programming model, which are addressed / invoked on the basis of the contents of the `message` argument delivered when making the system call. The `message` argument can thus be considered an implementation detail of sorts for this capability invocation API.

Most interaction with the seL4 kernel and kernel objects is done using the `libsel4` convenience library, that takes care of the marshalling work required for

1. `seL4_Yield()`
2. `seL4_Send(capability, message)`
3. `seL4_Recv(endpoint_or_notification_cap, out sender_info)`
4. `seL4_NBSend(capability, message)`
5. `seL4_NBRecv(endpoint_or_notification_cap, out sender_info))`
6. `seL4_Call(capability, message)`
7. `seL4_Reply(message)`
8. `seL4_ReplyRecv(recv_endpoint_or_notification_cap, reply_message, out recv_sender_info)`

Figure 1.2: seL4 System Call / Kernel API.
Adapted from the seL4 manual [66], with simplified argument names and `out` keywords to indicate arguments that the call returns output data through

making the underlying syscall, and provides a function for each 'method' that can be invoked on the types of capabilities involved in the system. These capabilities and methods are described in the next section.

1.3 seL4 Capabilities

The number of types of capabilties in an seL4 system depends on the particular target architecture/platform the kernel is being built for, and also on whether certain kernel features are enabled. However in general there are 10 or so main capability types that all systems will have and make use of: 'Untyped's to represent usable areas of physical memory, 'CNode's for storing capabilities, Thread control blocks for defining execution contexts, Endpoints for IPC, Notifications for signalling, 2 types for controlling and handling interrupts, and 3 or so types representing pagetable data structures for the target platform in question. An example, non-exhaustive overview of these types is given in Figure 1.3, and some of the associated 'methods' available on them are shown in Figure 1.4.

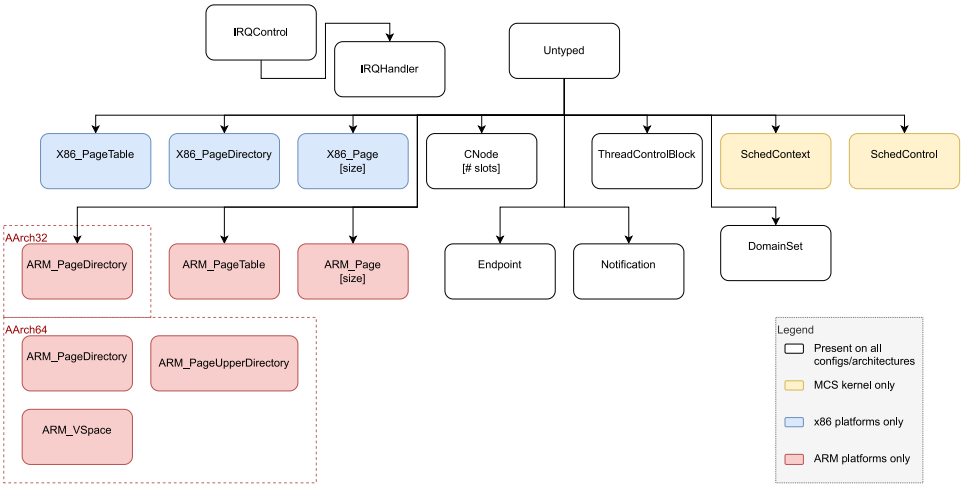

Figure 1.3: (Non-exhaustive) Overview of different capability types on seL4

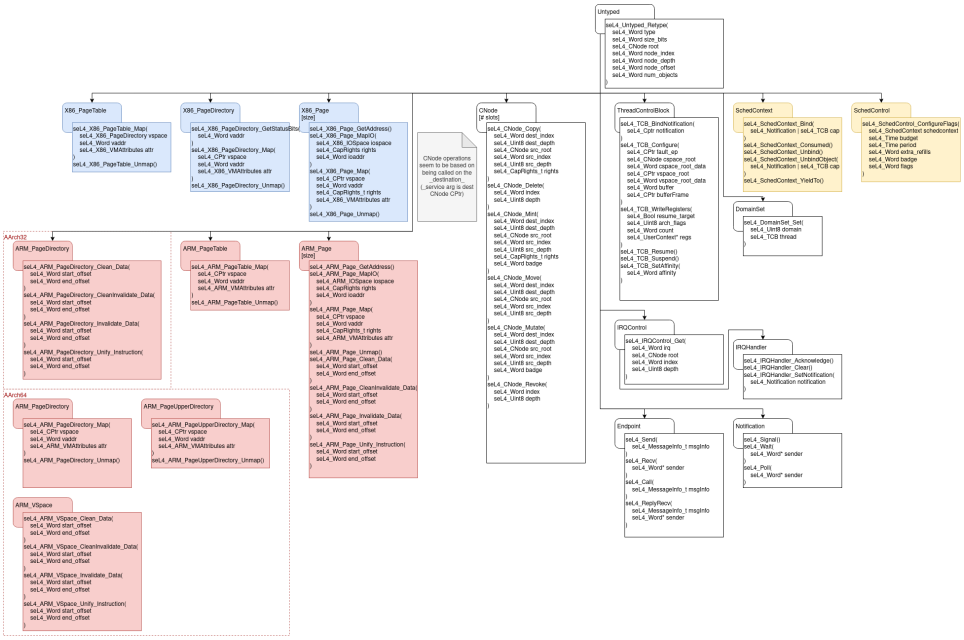

Figure 1.4: (Non-exhaustive) Overview of various API calls that can be made against the capability types of Figure 1.3

Many of these methods also involve passing other capabilities as arguments - for example, the virtual page mapping methods involve calling `Page_Map` methods on page capabilities, and passing the VSpace that the page is to be mapped into as an argument.

One of the key methods, probably used all in seL4 systems, is `seL4_Untyped_Retype`. This turns untyped memory into a derived, concrete type of capability, and is the main way useful capabilities in any system are set up. It too requires an additional capability as an argument - namely a `CNode` capability to be used for storing the resultant retyped capabilities within.

Besides retyping to produce specific types of new capabilities, capability management in general is mostly managed through use of methods invoked on the CNodes that store capabilities, including move, delete, copy, mutate (move with alteration). Most of these methods are shown in Figure 1.5.

Some capabilities can also be "badged" via the "mint" operation, where a new capability is created from an existing one, but with some additional word of data stored inside it as a "badge" value. Badged capabilities cannot be unbadged, rebadged, or have their badge value replaced or changed, and the badging itself can only be done by the owner of the original "unbadged" endpoint capability. The

```
seL4_CNode_Copy(
    Word  dest_index,
    Uint8 dest_depth,
    CNode src_root,
    Word  src_index,
    Uint8 src_depth,
    CapRights_t rights
)
seL4_CNode_Delete(
    Word  index,
    Uint8 depth
)
seL4_CNode_Mint(
    Word  dest_index,
    Uint8 dest_depth,
    CNode src_root,
    Word  src_index,
    Uint8 src_depth,
    CapRights_t rights,
    Word  badge
)
```

```
seL4_CNode_Move(
    Word  dest_index,
    Uint8 dest_depth,
    CNode src_root,
    Word  src_index,
    Uint8 src_depth
)
seL4_CNode_Mutate(
    Word  dest_index,
    Uint8 dest_depth,
    CNode src_root,
    Word  src_index,
    Uint8 src_depth,
    Word  badge
)
seL4_CNode_Revoke(
    Word  index,
    Uint8 depth
)
```

Figure 1.5: seL4 CNode methods with abbreviated argument pseudo types. Adapted from §3.1.2 and §10.3.1 of the seL4 manual [66]

most common situation this is used for is creating badged Endpoint capabilities - when an IPC message is sent through an endpoint using a badged capability, the badge value is exposed as an extra argument to the thread receiving it. This can be used by a 'server' thread to give out an endpoint to multiple clients or client contexts, providing them access to the one server endpoint in a way that is multiplexed on the server side. The badges can be used as a second layer of sorts for securely and uniquely identifying these clients or client contexts - for example, they could be used as a secure means of providing "file handles" for a file server, without requiring the file server to have explicit identity knowledge of what threads or components are on the other side of the endpoint the fileserver provides its API through.

A key distinction between seL4 capabilities is that of whether the invocations ultimately represent calling the kernel (and just the kernel) to do work, or whether the invocation represents calling into code elsewhere in userspace to do work. Endpoints, which power seL4 IPC, represent the latter - seL4 IPC will be covered more in section 1.5.

1.4 seL4 Protection Domains

Core to the security model of seL4 systems are the setup and design of 'protection domains'. These are essentially the walls within which code components and resources can be 'locked down' with, and primarily consist of both a 'CSpace' (capability space) and 'VSpace' (virtual address space). Every thread set up on an seL4 system has a particular CSpace and VSpace set on it, via properties on the thread object, and any userspace code on seL4 always executes within the context of one of these threads.

A **CSpace** is the set of all capabilities that the executing thread can make use of, and thus defines the 'world' of system access that a thread has. It consists of a root 'CNode' object, which is actually the physical memory storage location for capability datastructures, but more abstractly, defines the 'number space' that capability addresses are interpreted within, in terms of which capabilities exist at what capability addresses.

A **VSpace** is a virtual address space that the system Memory Management Unit (MMU) is set up by the kernel to interpret and use. This, similarly to the CSpace for capabilities, ultimately defines the 'number space' that memory addresses are interpreted within, in terms of which physical memory addresses should be used when accessing a virtual memory address.

Restricted VSpaces on seL4 are typically used as a security mechanism for device driver code - the driver is ultimately only ever given access to the specific control register memory addresses that can control the device, plus whatever physical memory the driver might need for data storage. This contrasts with Linux drivers, where the driver code executes within the entire kernel address space, which poses a large security risk for kernel exploits (see analysis of Linux kernel CVEs in [10]).

Restricted CSpaces are also core to the security of seL4 systems, as ultimately, a thread started in a CSpace will only ever be able to control or impact the system on the basis of the capabilities available to it within that CSpace. It can never invent or forge new capabilities - new capabilities can *only* come from retyping existing ones it was already explicitly given, or from capability transfer through seL4 IPC from another thread (as will be covered in the next section), or unless it is somehow explicitly donated a new capability by some other thread that has a capability to the root of its CSpace, via move/copy. This latter donation approach would typically only be taken for CSpace initialisation, as owning the root CN-

ode capabilities for multiple threads infers a high degree of trust on the domain holding those capabilities, and any further modifications to the CSpace of a thread once it has started may involve requiring knowledge of how that thread may or may not have filled up more of its CSpace through capability retyping or IPC.

Locked down protection domains are however generally not much use without the ability to talk to other protection domains, which is where seL4 IPC comes into the picture.

1.5 seL4 IPC and blocking semantics

seL4 IPC (inter-process communication) draws its design from its heritage of the L4 microkernel family, with a strong focus on fast performance (in a 'fastpath' case specifically), and a direct thread-to-thread communication model. For two threads in different protection domains to communicate via this IPC, they need to have capabilities to a shared Endpoint. IPC is in general a blocking operation - calling `Send()` or `Call()` on an endpoint will block the calling thread until such time that the IPC is handled by the other side of the endpoint. Similarly, calling `Recv()` or `ReplyRecv()` will block the calling thread until such time that a send is invoked from the other side of the endpoint.

IPC sending is invoked with a number of *message register* arguments, which are passed over to the receiving thread as output from the receive operation. In the general case, the messages are stored in an 'IPC buffer' set up as part of the thread control block, but in the fastpath case, the messages may actually be delivered within CPU registers kept aside during the switch from the sending thread to the receiver thread - a hallmark trick of the L4 microkernel.

IPC is in particular well optimised around the case of the `Call()` and `ReplyRecv()` syscall mechanisms, which combine the send and receive behaviours into a single call to provide for more optimised control flow, and which must be used for the fastpath to work. Issuing a `Call()` on an endpoint will provide the receiver with a one-time 'reply cap' capability, that allows the receiver to hand control back to the calling thread without needing to know its identity.

seL4 IPC in general is moreso, as per definitions discussed by Gernot Heiser [31], 'a user-controlled context switch with benefits', or 'the seL4 mechanism for implementing cross-domain function calls'.

Communication between protection domains can also be achieved by the use of Notifications, which exist primarily for signalling and synchronisation purposes. Notifications combined with shared memory mapped between two domains is a common pattern for high-throughput communication (as used in the recently-developed seL4 Device Driver Framework (sDDF) [35, §IV]).

1.6 seL4 boot process + root task capability bootstrap

When the seL4 kernel boots, it must carefully bootstrap some initial capabilities to faithfully represent the state of the system that it has been booted on, and any alterations to the state of the system that it has made as part of setting itself up. These alterations include:

- the use of physical memory for storing the kernel's own code, data, and stack
- the use of physical memory for storing the code, data, and stack of the root task the kernel has been configured to use
- the setup of a page table and associated mappings for the address space of the root task, which is ultimately stored somewhere in physical memory
- a CNode for the root task's CSpace to house various initial capabilities, including ones for all resources set up for it above. which is all, again, stored somewhere in physical memory

Some of this can be defined statically, but as seL4 can be booted on systems with dynamic hardware configuration (e.g. depending on what RAM sticks are physically installed), some of it must be determined at runtime as well. Part of the boot process also involves a 'bootstrap' where `Untyped` capabilities are created to represent all the areas of memory that have *not* been used by the kernel and its setup, which are then handed over to the root task - i.e. the capabilities are stored into the CSpace that the root task's initial thread is started with.

The root task itself needs to be able to know what initial capabilities it has been set up with, both statically and dynamically. Figure 1.6 shows an enumeration of the set of static / well-known capabilities that the kernel will always set up in the first 16 or so slots of the CSpace of the root task. On the dynamic side, as the seL4 capability APIs do not typically give any mechanisms for identifying capabilities or reporting information about their current state, this information has to be explicitly provided in memory that the root task can read. This is done via the 'bootinfo' struct, which the kernel sets up, then passes by pointer as the first argument to the entry point of the root task. The format and structure of the bootinfo structure is shown below in Figure 1.7.

```
/* caps with fixed slot positions in the root CNode */
enum {
    seL4_CapNull                =  0, /* null cap */
    seL4_CapInitThreadTCB       =  1, /* initial thread's TCB cap */
    seL4_CapInitThreadCNode     =  2, /* initial thread's root CNode cap */
    seL4_CapInitThreadVSpace    =  3, /* initial thread's VSpace cap */
    seL4_CapIRQControl          =  4, /* global IRQ controller cap */
    seL4_CapASIDControl         =  5, /* global ASID controller cap */
    seL4_CapInitThreadASIDPool  =  6, /* initial thread's ASID pool cap */
    seL4_CapIOPortControl       =  7, /* global IO port control cap (null cap if not
↪   supported) */
    seL4_CapIOSpace             =  8, /* global IO space cap (null cap if no IOMMU
↪   support) */
    seL4_CapBootInfoFrame       =  9, /* bootinfo frame cap */
    seL4_CapInitThreadIPCBuffer = 10, /* initial thread's IPC buffer frame cap */
    seL4_CapDomain              = 11, /* global domain controller cap */
    seL4_CapSMMUSIDControl      = 12,  /*global SMMU SID controller cap, null cap if
↪   not supported*/
    seL4_CapSMMUCBControl       = 13,  /*global SMMU CB controller cap, null cap if
↪   not supported*/
#ifdef CONFIG_KERNEL_MCS
    seL4_CapInitThreadSC        = 14, /* initial thread's scheduling context cap */
    seL4_NumInitialCaps         = 15
#else
    seL4_NumInitialCaps         = 14
#endif /* !CONFIG_KERNEL_MCS */
};
```

Figure 1.6: Fixed/'well-known' capability addresses for capabilities set up for the root task as part of the seL4 boot process.
Sourced from `libsel4/include/sel4/bootinfo_types.h` in the seL4 kernel codebase

```c
typedef seL4_Word seL4_SlotPos;

typedef struct seL4_SlotRegion {
    seL4_SlotPos start; /* first CNode slot position OF region */
    seL4_SlotPos end;   /* first CNode slot position AFTER region */
} seL4_SlotRegion;

typedef struct seL4_UntypedDesc {
    seL4_Word  paddr;   /* physical address of untyped cap  */
    seL4_Uint8 sizeBits;/* size (2^n) bytes of each untyped */
    seL4_Uint8 isDevice;/* whether the untyped is a device  */
    seL4_Uint8 padding[sizeof(seL4_Word) - 2 * sizeof(seL4_Uint8)];
} seL4_UntypedDesc;

typedef struct seL4_BootInfo {
    seL4_Word         extraLen;        /* length of any additional bootinfo
↪   information */
    seL4_NodeId       nodeID;          /* ID [0..numNodes-1] of the seL4 node (0 if
↪   uniprocessor) */
    seL4_Word         numNodes;        /* number of seL4 nodes (1 if uniprocessor)
↪   */
    seL4_Word         numIOPTLevels;   /* number of IOMMU PT levels (0 if no IOMMU
↪   support) */
    seL4_IPCBuffer   *ipcBuffer;       /* pointer to initial thread's IPC buffer */
    seL4_SlotRegion   empty;           /* empty slots (null caps) */
    seL4_SlotRegion   sharedFrames;    /* shared-frame caps (shared between seL4
↪   nodes) */
    seL4_SlotRegion   userImageFrames; /* userland-image frame caps */
    seL4_SlotRegion   userImagePaging; /* userland-image paging structure caps */
    seL4_SlotRegion   ioSpaceCaps;     /* IOSpace caps for ARM SMMU */
    seL4_SlotRegion   extraBIPages;    /* caps for any pages used to back the
↪   additional bootinfo information */
    seL4_Word         initThreadCNodeSizeBits; /* initial thread's root CNode size
↪   (2^n slots) */
    seL4_Domain       initThreadDomain; /* Initial thread's domain ID */
#ifdef CONFIG_KERNEL_MCS
    seL4_SlotRegion   schedcontrol; /* Caps to sched_control for each node */
#endif
    seL4_SlotRegion   untyped;         /* untyped-object caps (untyped caps) */
    seL4_UntypedDesc  untypedList[CONFIG_MAX_NUM_BOOTINFO_UNTYPED_CAPS]; /*
↪   information about each untyped */
    /* the untypedList should be the last entry in this struct, in order
     * to make this struct easier to represent in other languages */
} seL4_BootInfo;
```

Figure 1.7: C types showing the layout of data within the seL4 'bootinfo' struct passed to the root task of the system.
Sourced from `libsel4/include/sel4/bootinfo_types.h` in the seL4 kernel codebase

1.7 'Memory Server' example

One of the first and most important problems involved in the design of systems on top of seL4 is that of memory allocation. When the seL4 kernel boots up, as described in section 1.6, it hands the root task a set of `Untyped` capabilities that represent all remaining memory available for use after boot. From here, all subsequent capabilities in the system will have to descend from these initial capabilities, as subsequent threads, CSpaces, and VSpaces must all be constructed by splitting down and retyping these `Untyped` capabilities. This is a core part of seL4's security model, as it both ensures that all physical memory is accounted for, but also that there is a strong chain of authority behind how that physical memory gets distributed around all the components of the system[2].

Memory allocation in seL4 systems is often managed by some kind of *memory server*, which the root task itself may be set up to act as, or that the root task can set up with all or most of the untypeds the root task was handed from the kernel boot. This memory server can then provide an Endpoint-based API for other tasks / processes / components to request additional memory with, and any components / tasks that might need more memory can be set up with an endpoint to talk to the memory server with. In typical microkernel fashion, any *policy* for which components should be allowed more memory is not part of the microkernel itself - it must be implemented somehow as part of code in userspace.

The recently-developed 'KataOS' by Google Research [58] provides an example of this, with its *`MemoryManger`* CAmkES component[3], the interface of which is shown in Figure 1.8 and Figure 1.9.

[2] A great example of this can be found in Figure 16 of [39, pg 2:46], which displays a full derivation graph of the capDL capabilities from an example seL4 system

[3] Note that whilst CAmkES itself does not usually provide for capability transfer through the seL4 endpoints set up for component communication, the KataOS developers have done this manually as part of their MemoryManager server and clients.

```
// Kata OS MemoryManager service.

import <LoggerInterface.camkes>;
import <MemoryInterface.camkes>;

component MemoryManager {
  provides MemoryInterface memory;

  maybe uses LoggerInterface logger;

  // Enable KataOS CAmkES support.
  attribute int kataos = true;

  // Mark the component that should receive the unallocated UntypedMemory
  // passed to the rootserver from the kernel. In addition to the
  // capabilities the component also gets a page with Bootinfo data that
  // includes updated UntypedMemory descriptors. In order to pass the
  // capabilitiies the component's cnode is up-sized to be large enough
  // to hold the extra capabilties.
  attribute int untyped_memory = true;
}
```

```
procedure MemoryInterface {
  include <MemoryManagerBindings.h>;

  MemoryManagerError alloc(in char request[]);
  MemoryManagerError free(in char request[]);
  MemoryManagerError stats(out RawMemoryStatsData data);

  void capscan();
  void debug();
};
```

Figure 1.8: `MemoryInterface` CAmkES component and interface from Google KataOS. The format of `char request[]` is detailled in Figure 1.9
Sourced from the KataOS repository `https://github.com/AmbiML/sparrow-kata-full/tree/main/apps/system/components/MemoryManager`

```rust
// Objects are potentially batched with caps to allocated objects returned
// in the container slots specified by the |bundle| objects.
pub trait MemoryManagerInterface {
    fn alloc(&mut self, bundle: &ObjDescBundle) -> Result<(), MemoryError>;
    fn free(&mut self, bundle: &ObjDescBundle) -> Result<(), MemoryError>;
    fn stats(&self) -> Result<MemoryManagerStats, MemoryError>;
    fn debug(&self) -> Result<(), MemoryError>;
}

pub struct ObjDesc {
    // Requested object type or type of object being released.
    pub type_: seL4_ObjectType,

    // Count of consecutive objects with the same type or, for CNode
    // objects the log2 number of slots to use in sizing the object,
    // or for untyped objects the log2 size in bytes, or for scheduler
    // context objects the size in bits. See seL4_ObjectType::size_bits().
    count: usize, // XXX oversized (except for untyped use)

    // CSpace address for realized objects requested. If |count| is >1
    // this descriptor describes objects with |cptr|'s [0..|count|).
    // Since each block of objects has it's own |cptr| one can describe
    // a collection with random layout in CSpace (useful for construction).
    //
    // Object capabilities returned by the MemoryManager have the maximal
    // rights. We depend on trusted agents (e.g. ProcessManager) to reduce
    // rights when assigning them to an application. This also applies to
    // the vm attributes of page frames (e.g. mark not executable as
    // appropriate).
    pub cptr: seL4_CPtr,
}
// ObjDescBundle holds a collection of ObjDesc's and their associated
// container (i.e. CNode). This enables full "path addressing" of the
// objects. Helper methods do move/copy operations between a component's
// top-level CNode and dynamically allocated CNodes.
#[derive(Clone, Debug, Serialize, Deserialize)]
pub struct ObjDescBundle {
    pub cnode: seL4_CPtr,
    pub depth: u8,
    pub objs: Vec<ObjDesc>,
}
```

Figure 1.9: Assorted snippets of the MemoryManager component from KataOS, showing its actual Rust-typing interface.
Sourced from the KataOS repository `https://github.com/AmbiML/sparrow-kata-full/tree/main/apps/system/components/MemoryManager`

Chapter 2

Object-Capability Language background

The Object-Capability family of languages stems back primarily to the development of E, a programming language devised by Mark Miller as part of his PhD thesis at John Hopkins University [45], which also formally proposed the Object-Capability model. This section provides some brief background to the language family and some common patterns associated with them that will be referenced later.

Taking inspiration from the same capability model from operating systems research, in the Object-Capability programming model, object *references* in the language are viewed as the 'capabilities' for accessing and controlling code (objects) or other resources. The model dictates that an object A can only control or interact with another object B if it has a 'reference' or 'capability' for the other object. For this to be secure, object references must be unforgeable, which is the case in memory-safe languages such as Java and JavaScript.

These references similarly must be given to objects explicitly for them to receive them in the first place, which allows for strong security reasoning over the control of use of code in the language. For example, untrusted third party code C can be run by a parent program P in a more trusted way, where C is only provided access to explicit subsets of the objects and code that P controls.

2.1 Confined Execution

A hallmark of many object-capability systems, especially Secure EcmaScript and its deriveratives (which will be covered more in section 4.2), is that of *confined execution*.

Normally in many programming language environments, the space of available primitives is essentially global - modules can be imported on-demand as they are needed. This makes programming easier and more useful, but it comes with big security risks if dynamic control is gained over the language execution environment by an attacker, or if untrusted code is allowed to execute.

In object-capability models, having mutable global objects immediately breaks the security model, as units of code could write references to these global objects to allow other units of code to obtain them without having been explicitly passed them. This leads to a tendency for *everything* being explicit. Even basic features such as clock access are put behind APIs that require access to specific root capabilities, such that code in any particular object-capability environment will only be able to use these features if they were *explicitly passed access to them*.

2.2 Vats and remote objects

Another object-capability language feature is that of 'vats', which are object spaces with a thread of execution control. The original distributed programming environment provided by E included a definition of vats [45, §14.1], which are 'hosted' on various different computers to facilitate distributed computing. Objects from a remote vat can be *introduced* into the local execution environment by means of a cryptographic object capability URL[1]. This is an example of a *sparse capability* model (as briefly mentioned in the different capability implementation models in section 1.1), where capabilities are treated as 'effectively unforgeable' by nature of being impracticably large random numbers.

The Secure EcmaScript ecosystem also includes a proof-of-concept Vat system called 'SwingSet'[2].

[1] See `http://erights.org/elang/concurrency/introducer.html` for an example

[2] See `https://github.com/Agoric/agoric-sdk/tree/master/packages/SwingSet`

Chapter 3

Project Motivation and current seL4 tooling review

As described in chapter 1, seL4 programming is mostly done in C, where seL4 capabilities must be managed manually through use of capability addresses and the `libsel4` kernel programming C API. Similar to the issues involved with working with pointers in C, working with capabilities in this environment is fiddly and it is difficult to use them and the associated APIs expressively when managing capabilities dynamically. Capability addresses are essentially 'magic numbers' that on their own give no indication of the type of capability they refer to, unless this is expressed through constant or variable naming, or an associated 'type' field. The type used by the all the `libsel4` API functions for capabilities is `seL4_CPtr`, which is just a C `typedef` of `seL4_Word`, an unsigned, word-sized integer type for the processor/platform the kernel is being built for.

CAMkES [26] and its associated tool CapDL [40, 12] have been developed to provide a more supportive component model for working with seL4 capabilities.

A sample capDL specification is shown in figure 3.1. Use of capDL at least provides for a more typed and specific way to refer to seL4 capabilities - however, only in a static manner, as capabilities are allocated / distributed up front at project build time. To use these capabilities at runtime, they are still ultimately accessed via magic `seL4_CPtr` numbers exported back out into C code.

CAMkES also only facilitates working with capabilities statically - capabilities are allocated for all the components up front at build time, and CAMkES's mechanisms do not provide for any runtime transfer of capabilities - doing so requires falling back onto the aforementioned `libsel4` C APIs.

```
arch ia32

objects {

  my_tcb = tcb
  my_cnode = cnode (3 bits)
  my_frame = frame (4k, paddr: 0x12345000) // paddr is optional
  my_page_table = pt
  my_page_directory = pd

}

caps {

  // Specify cap addresses (ie. CPtrs) in cnodes.
  my_cnode {
    1: my_tcb
    2: my_frame
    3: my_page_table
    4: my_page_directory
  }

  // Specify address space layout.
  // With 4gb page directories, 4mb page tables, and 4kb frames,
  // the frame at paddr 0x12345000 will be mapped at vaddr 0xABCDE000.
  my_pd {
    0x2AF: my_pt
  }
  my_pt {
    0xDE: my_frame
  }

  // Specify root cnode and root paging structure of thread.
  my_tcb {
    vspace: my_pd
    cspace: my_cnode
  }

}
```

Figure 3.1: Sample capDL spec
Sourced from the capDL documentation at `https://docs.sel4.systems/projects/capdl/`

When it comes to the challenge of building more dynamic systems on seL4, where the set of components on the system is not necessarily fixed or known at project-build time, there are not really any options or solutions for anything that maps more natively to the seL4 environment. Whilst CAmkES provides a good model for defining and connecting components, it does not really include any tools or solutions for spinning them up dynamically, as the components when built as part of a CAmkES 'assembly' have a strong tie back to the capabilities that are (statically) allocated for them via capDL. In any CAmkES project, the root task of the seL4 system is always the capDL loader[1], which is a C program that gets built with the inclusion of a generated `capdl_spec.c` file[2] that contains one large C struct defining all the capabilities that were described as part of the capDL spec generated from the assembly of CAmkES components. The capDL loader uses this as the basis for a one-time setup of real seL4 capabilities at runtime when it starts up, before starting all the threads of the components and then suspending itself.

For dynamic systems, projects will end up falling back onto plain POSIX layers, which re-introduce the issue of ambient authority as described in section 1.1. There is an existing alternate solution in the form of Genode [30, 28], which is a toolkit for building capability-focused operating systems. However, it has its own notion of capabilities that don't necessarily match seL4's, as it is designed to support building systems with many different kernels (including Linux).

Object-Capability languages pose an interesting and potential solution to this issue of constructing dynamic systems, as they are capable of giving degrees of assurance around what code can or cannot access in a more dynamic manner. This thus posed as the motivation for the project: to see if an object-capability language could be adapted onto seL4 in a way that would make working with the operating system's capabilities easier, more 'natural', or more useful, and in a way that more naturally matches the 'separated components' model that generally underlies the design and development of systems built atop seL4.

[1] See the `BuildCapDLApplication` and `DeclareRootserver` calls in `https://github.com/seL4/camkes-tool/blob/camkes-3.10.0/camkes/templates/camkes-gen.cmake#L596`

[2] See the definition of the `BuildCapDLApplication` CMake helper function in `https://github.com/seL4/capdl/blob/0.2.1/capdl-loader-app/helpers.cmake#L9`

Chapter 4

Survey of Object-Capability Languages and appropriateness for mapping

Many different object-capability (or 'ocap'[1]) languages exist, mostly from the results of various research papers and projects. The first stage of research work for this thesis / project was to explore what languages existed and to evaluate which might be the best pick for adapting to the seL4 microkernel environment. This section will cover the ocap languages that were considered as a form of literature review for the project.

[1]The term 'ocap' is used throughout this thesis to refer to object-capabilities in short.

4.1 E

One of the seminal authors on the topic of object-capability languages is Mark Miller, who wrote his PhD thesis on the topic in 2006 [45]. The thesis introduced the language E, which has become a seminal ocap language that has inspired the design of many other future languages.

E consists both of a language, and a distributed object programming environment, where 'vats' of objects can communicate between each other with secure message-passing. The CapTP and VatTP protocols provide means for objects to securely share references to other vats, with a 'swiss number' system for object identity that is essentially a sparse capability model, and the language contains various levels of object references, including 'NearRefs', 'FarRefs', and 'SturdyRefs', to represent different types of object references (and to thus ensure that the appropriate local or remote code is invoked depending on the type of reference).

E was initially developed as an extension to Java, and its primary available form is still called 'E-on-Java' [44], downloadable as a tarball with some bash scripts and a `.jar` file containing most of the language and runtime. The latest version, 0.9.3, is dated late 2016[2], but the version before appears to be from 2009[3], highlighting that E itself is quite an old language that does not seem to have been worked on or updated in a while.

E's basis of Java introduces ones of the key problems for porting a language onto seL4 - support for dependencies that the language in question is built on top of. Running JVM bytecode on seL4 would require a JVM runtime for the seL4 environment, which does not appear to currently exist, and porting one could be a very non-trivial exercise, as seL4 is in many ways quite a non-standard environment (e.g. the kernel itself is not a POSIX-compliant environment). JVM-based languages also typically rely on at-runtime JIT (just-in-time) compilation for performance, which could also be quite non-trivial to implement on top of seL4, and a lot of dynamic memory allocation and reliance on garbage collection. This could also be complicated to implement on seL4's capability-based memory management structure (and which would be implemented in a non-capability-based language, which somewhat defeats the purpose of the project).

Another version of E is also available, 'E-on-CL' (Common Lisp) [38], but

[2] See `http://erights.org/download/0-9-3/`

[3] See `http://wiki.erights.org/mediawiki/index.php?title=ReleaseNotes092&action=history`

suffers the same language dependency problem but for Common Lisp (and in fact still requires Java as well for the language parsing). For these reasons E was not investigated for mapping any further.

4.2 Jessie / Secure EcmaScript

A more recent work of Mark Miller's is the Jessie [3] language, which is built on top of the Secure EcmaScript system/runtime and designed to be "a small, safe ocap subset of Javascript". Secure EcmaScript itself was developed by Miller as part of a train of work including Caja [46] and "Dr. SES" (Distributed Resilient Secure EcmaScript) [47], and is now also referred to as 'Hardened Javascript' as a component of the 'Endo' Javascript platform [1].

Caja was a circa-2008 Google project (deprecated as of Jan 21 2021) aiming at providing a safe environment for running untrusted scripts within the context of a web page in web browser. In Caja, JavaScript scripts are translated into a 'safe' form, and passed objects as object capabilities from the executing page context to allow careful, contained execution of these untrusted scripts. The passed-in object capability objects meant that the scripts could still invoke code already on the page, albeit in a very explicitly controlled manner. Jessie is a later (circa-2019) development of the SES ecosystem that constitutes more of a dedicated language than just a Javascript runtime.

Jessie is appealing for many object-capability reasons, and was initially the frontrunner language choice for this project, but is quite complex in terms of implementation details. The language itself is defined as a parsing-expression grammar, and in the current implementation [2], the grammar is written in a 'quasi parsing-expression generator' that is itself written in Javascript, and which relies on string template tags, a more modern Javascript language feature.

Jessie and SES also rely on many not-yet-standardised extensions to Javascript such as Compartments [59] and Frozen Realms. This all combines to make a rather complicated dependency research project (in addition to the previously mentioned OS-level dependency problems), as any choice of Javascript runtime needs to consider whether the language features in question are supported or not, and if not, if they can be polyfilled with extra Javascript code (as is common practice in the web development community for supporting Javascript language features in older web browsers that don't support more modern features).

V8 is probably the most common/popular Javascript runtime, written primarily as the Javascript runtime for the Chromium browser project, and also used by NodeJS. As of recent years it has strong, mature support for many modern Javascript language features. However it is primarily written in C++, and relies strongly on JIT compilation for performance. No C++ development toolchain ap-

pears to currently exist for seL4, and JIT compilation (requiring the ability to write code into data sections of memory) would likely have been more complicated to implement, so it was not considered further for these reasons.

There are various other smaller Javascript runtimes that could have been investigated further, such as QuickJS [4], Duktape[5], and mJS [6]. However the general problems of poor performance (amplified by much of the Jessie language internals themselves being written in Javascript) made these still unideal choices for something to use for development within the seL4 microkernel environment.

Javascript is in general not a particularly performant programming language by its very nature, mainly due to the generality that some of its language features provide. In most modern contexts (such as web browsers), there is a strong reliance on JIT compilation, guided by profiling of the code at runtime, to achieve acceptable levels of performance. For example, objects in the language can have dynamically-assigned properties, which are often string-keyed and thus may require the allocation and use of a hashmap datastructure, even in cases where many objects that have the same structure are allocated. For performance improvement, many Javascript engines will perform 'inline caching' of the structure of objects based off their 'shape' - i.e. what properties they get defined with [51]. Even changing the order in which properties are defined can cause a large difference in performance (7x slowdown in the example shown in [51]).

[4] `https://bellard.org/quickjs/`
[5] `https://duktape.org/`
[6] `https://github.com/cesanta/mjs`

4.3 SHILL

SHILL [48] is a capability-based shell scripting language for FreeBSD, built to facilitate the execution of shell scripts in contained sandboxes that limit their resource access to only resources explicitly provided. SHILL scripts by definition come with *contracts*, which 'specify what capabilities a script requires and how it intends to use them' [48]. A similar project called Plash (Principle of Least Authority SHell) [53] exists for Linux, which uses a modified version of GNU libc to virtualise access to the filesystem.

SHILL is implemented as an extension to the Racket language [29] (a dialect of Lisp that descends from Scheme), using the language macro system. This poses another complicated language stack dependency that could prove quite difficult to port and debug. Additionally, the target domain of SHILL, a shell scripting environment, is somewhat smaller compared to that of a broader general-purpose programming language. For these reasons it was a lower priority option to consider, but one that nonetheless highlights another useful context that object-capabilities have been applied within.

4.4 Dala

The most recent ocap language found in the survey was Dala [27], published in 2021. Dala is similar to Pony (covered next in section 4.5) in that it is a language built to provide 'data-race freedom' - the ability to write parallel/concurrent code that comes with an assurance that no data races will occur. Dala introduces the idea of a 'safe' heap, distinct from a pre-existing 'unsafe' heap, where gradually-increasing guarantees can be made about about objects upon the heap relating to what thread or threads can access them and in what manner.

The current available implementation of Dala is called Daddala [7], which is written in Grace, another programming langauge, and realised atop a Grace interpreter called Moth (or, the Moth VM). Moth in turn is built on top of 'SOMns'[41] - an implementation of the Newspeak language [13], which is 'a dynamic, class-based, object-oriented language in the tradition of Smalltalk and Self' [41]. SOMns is based on TruffleSOM, an implementation of SOM (Simple Object Machine, that SOMns derives from) [56], realised using the Truffle framework, a Java framework that is part of the GraalVM compiler (which is also written in Java). This presents perhaps the most dizzying stack of language dependencies so far, which made it a very difficult option to proceed any further with due to the lack of a Java environment for seL4 as mentioned in section 4.1. The authors also note in a presentation on the language [64] that the present implementation currently includes many levels of dynamic checks, which could be removed in future work with a gradual type system, but currently could also pose a performance problem that is not ideal for the reasons mentioned in covering the previous ocap languages.

[7] `https://github.com/gracelang/moth-SOMns/tree/daddala`

4.5 Pony

Pony [19, 17] is a language that was developed as part of research at University College London, including Sylvan Clebsch's PhD thesis [17], which describes much of the language design and implementation in detail. It combines an actor-model of execution with an advanced type system, with the goal of providing a guarantee that data-races will be made impossible by nature of the type system checking correctly. The type system introduced a novel concept of 'reference capabilities' - a capability model for references to objects and variables in the language, with various 'deny' properties built into their type system rules for ensuring isolation of read and write access to data in memory.

Alongside the reference capability system, Pony's object and type system also provide object-capability guarantees, making it classed as an ocap language as well [61] - albeit one rooted within the actor model the language is built around.

One of Pony's core language goals was to achieve C-like performance, which makes it an attractive choice for developing on top of the seL4 environment. Unlike all the other languages explored above, Pony programs compile directly to machine code via LLVM, and the only other 'dependency' for the language is ultimately just a C library (`libponyrt`), which seemed much more feasible to port to the seL4 environment given that most seL4 projects are written in C.

Pony still comes with the dynamic memory allocation and garbage collection problems of the other languages, but at least has a garbage collection system that is designed in a more modern, novel way for performance in a multithreaded environment (mainly without requiring a '*stop-the-world* GC step'). For these reasons it became the frontrunner choice for investigating an implementation of.

More relevant details of Pony will be covered in chapter 5.

4.6 Other related art

As part of the search for an appropriate object-capability language, various other related developments were discovered that could be worth further investigation in future work.

4.6.1 Rust - `cap-std` and `ferros` crates

Rust [42] is a popular systems programming language that has emerged in recent years, as a project from Mozilla, which was put to use there to try and reduce concurrency bugs within the engine of the Firefox browser as part of another project called 'Servo' [4]. Rust's borrow checking is in fact very similar to Pony's reference capability exclusion guarantees, with Pony even comparing it explicitly in discussion of its reference capabilities [19, §6 and table 4].

There is an assorted degree of 3rd-party Rust support for seL4 userspace development, including the Robogalia project [23], and the Ferros [7] and `selfe-sys` [6] Rust libraries from Auxon Corporation. More recently, Google has also produced a research project for a secure operating system built on top of seL4, called KataOS [58], which is implemented almost entirely in Rust. (This announcement unfortunately came out in the last few months of this thesis / research project, and as such was not able to be investigated in-depth.)

On the object-capability front though, it is not clear if any full / complete object-capability environment has been created within or on top of Rust yet. A project from Bytecode Alliance called `cap-std` [14] has been produced, which is a 'capability-based version of the Rust standard library', however its main usage appears to have been as the basis for the WebAssembly System Interface implementation of the Wasmtime WebAssembly runtime (which will be covered in the next section).

Rust is definitely an option worthy of further research, but given that other ocap languages had confirmed / stable ocap environments, it did not end up being investigated further as part of this thesis.

4.6.2 WebAssembly + WebAssembly System Interface (WASI)

A perhaps unexpected development to include capability concepts turned out to be the upcoming WebAssembly standard and execution system, with the WebAssembly System Interface proposal in particular being based strongly around a capability-based security model.

WebAssembly [33] is a developing standard for a stack-based virtual machine and corresponding binary instruction format, primarily designed for providing an environment within web browsing contexts for high-performance code execution, and a target that compiled languages can compile to to execute within those contexts, which can provide a means for C/C++ applications to be re-compiled to run in a web browser context. There are many examples of this in action powered by Emscriptem, an LLVM-to-WebAssembly compiler, including a port [5] of the (C-based) Quake III engine (ioquake3) to the browser via WebAssembly and WebGL, and support for targeting the C++-based Qt UI and software development framework into browser WebAssembly environments [63].

One of WebAssembly's primary design goals is to provide a *safe* execution environment, and the execution environment is thus carefully sandboxed by design from the environment it runs on, to prevent the broader system that the web browser (or other execution environment) is running on from being exploited. Aspects of memory safety are a key consideration in the design, as explained in this section from the original published paper presenting its motivation and initial design:

> ***Security*** Linear memory is disjoint from code space, the execution stack, and the engine's data structures; therefore compiled programs cannot corrupt their execution environment, jump to arbitrary locations, or perform other undefined behavior. At worst, a buggy or exploited WebAssembly program can make a mess of the data in its own memory. This means that even untrusted modules can be safely executed in the same address space as other code. [33]

Due to compiled WebAssembly code not being stored in the linear memory that can be addressed from the code, a key addition to the specification was that of function *tables*, which provide a protected indexing environment for function lookup, to provide compatibility with the commonly used C paradigm of passing functions by pointer whilst preventing malicious code from corrupting the code areas of memory. The 1.0 WebAssembly spec [70] only specifies tables for

storing function references, but notes that for table elements, 'In future versions of WebAssembly, additional element types may be introduced' - and the 2.0 spec draft (as of 2022-04-19) [69] includes an `externref` type alongside the function ref type, for 'references to objects owned by the embedder and that can be passed into WebAssembly' [69].

The 2.0 spec draft also now describes tables more broadly as: 'a vector of opaque values of a particular reference type' [69, §2.5.4]. This very similar to that of a CSpace/CNode in capability-based operating systems (as descibed for seL4 in section 1.4), where objects (capabilities) are stored in protected-indexed tables (CSpaces/CNodes) separate to that of normal, addressable memory.

The WebAssembly sandboxing also means that much OS functionality is completely unavailable to code in the WebAssembly runtime environment - at least solely in the core specification, and not in the way compiled code usually makes use of OS functionality (i.e. system calls). The WebAssembly System Interface (WASI) [74, 73] is a proposal from a subgroup under the W3C WebAssembly Community Group for providing a mechanism for exposing API interfaces to these kinds of features, and aims to be developed from the ground up to provide capability-focused APIs. In fact the first 'high-level goal' of WASI [73, §WASI High Level Goals] mentions defining APIs that 'preserve the essential sandboxed nature of WebAssembly through a Capability-based API design', and 'capability-based security' is the first of its listed 'design principles' [73, §WASI Design Principles]. This section goes on to mention how WebAssembly's current reference types can be used as a basis for implementing such capability-based security design.

As of October 2022, WASI consists solely of a set of early-stage proposals, with the furthest-progressed proposals at stage 2 only having 'proposed spec texts available' [72]. There is an implementation of some WASI interfaces and functions [15] in the Wasmtime WebAssembly runtime (via the `wasmtime-wasi` Rust crate, which uses the `cap-std` crate mentioned in subsection 4.6.1), developed by the ByteCode Alliance. Another WebAssembly runtime, WasmEdge, has been ported to run on seL4 [55, 54], but has its own separate set of WASI proposal support [71] which as of October 2022 only includes proposals for (Linux) sockets, cryptography functions, neural network access (inference), and network proxy control, all targeting the Linux platform.

As the citations in this section show, much of WebAssembly is in draft status and still under proposal (with WASI even more so than WebAssembly), so there has not really been as much to work with as of right now. Additionally,

WebAssembly is not quite a programming language in the same sense as C / C++ / Java / JavaScript, given that it is a ultimately a bytecode format much closer to machine assembly. However it is worth highlighting as a potential future avenue for revisiting, given how many parallels there are with the capability-based resource protection and usage model.

4.6.3 Microsoft Singularity Project

The Microsoft 'Singularity' project [37, 36] contains many similar parallels to this research. The Sing♯ language [25] developed for it has many similarities to Pony - it was focused primarily on message-passing communication through strongly-typed/contract-based 'channels', including a first-class `switch receive` statement for blocking on message receiving, and had a model for eliminating data races via tracking pointer ownership at compile time [see 25, §3]. Buffers and other memory data structures in Sing♯/Singularity were also capable of being transferred through messages thanks to an 'exchange heap' provided by the operating system.

Singularity was a large integrated language-based system project, with Microsoft's Common Intermediate Language (CIL, also previously known as MSIL) as a common bytecode compile target that was then compiled to machine code by a compiler and runtime system called Bartok. A binary version of Bartok (for Windows) was released as part of the Singularity Research Development Kit, which was a very Windows-centric project (e.g. MSBuild as the build system), and released under the Microsoft Research License Agreement. This made adopting anything practically difficult, and the standard issues of adapting a complex bytecode-based runtime to seL4 would likely have applied as well.

Chapter 5

Pony Background

After the review of ocap languages in chapter 4, Pony was ultimately settled on as the language of choice to focus on trying to adopt to the seL4 environment, mainly due to it being the language with seemingly the lowest amount of dependencies to port into the minimal C environment usually provided when developing on top of seL4.

Whilst the language has already been briefly outlined in section 4.5, some further background detail of Pony relevant for later sections of this thesis are covered in this section.

Sylvan Clebsch's 2017 PhD thesis on Pony [17] will be referenced much as part of this, and has thus been aliased as 'ClebschThesis' wherever it is cited, for clearer reference.

5.1 The Pony language + execution model

As outlined in section 4.5, Pony is an actor-model programming language. Actor types are defined with 'behaviours', which are asynchronous message-handling methods, and calling them is termed 'invoking' a behaviour of an actor, which actually just consists of sending a message to it for it to pick up and execute the behaviour with later. Actor behaviours thus become units of sequential execution, where the programmer can know that whenever any one message is being handled for a particular actor instance, no other behaviour on that actor could be being handled at the same time, either concurrently or in parallel.

Actor models are usually designed to achieve highly performant parallelism without requiring the programmer to manually wrangle threads and locks. In Pony, parallelism is achieved under the assumption that there will be many actors allocated during the course of the program execution, each with their own queue of incoming messages that get sent into to invoke the defined behaviours on the actor, and that any one actor can be picked up and run by a 'runtime thread' when it has messages in its queue.

An example of the runtime model is shown in Figure 5.1. Actor A on the left has many messages in its queue, and is currently being handled by runtime thread 1, which has popped a message off the queue to find it being one representing an invocation of the `behvFoo` behaviour for that actor type. The runtime thread thus calls in to the compiled code for that behaviour, executing it with argument values from the incoming message. The execution of this behaviour includes a line that invokes the `behvBar` behaviour of Actor B. The runtime code for this simply pushes a message representing the invocation of `behvBar` onto Actor B's message queue, which is a lock-free operation.

As part of its development, Pony also introduced two novel schemes of garbage collection and dead-actor collection: ORCA - '**O**wnership and **R**eference **C**ounting-based Garbage Collection in the **A**ctor World' [20]- and **M**essage-based **A**ctor **C**ollection (MAC) [18]. These schemes both make use of the same message-passing mechanism used for invoking actor behaviours to interleave reference-counting and state-related information into the same model used for the distributed communication, to allow for garbage and dead actor collection to occur without having to pause the entire execution of the program.

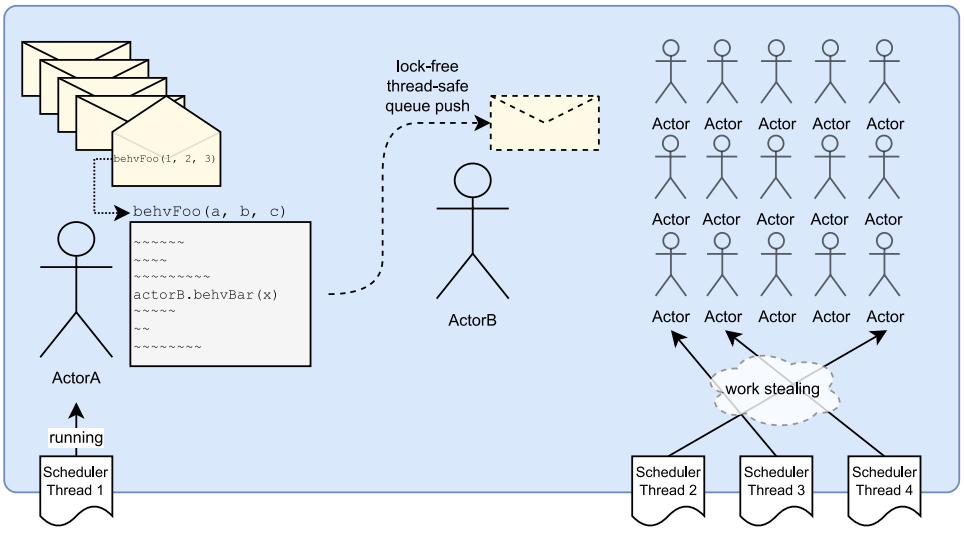

Figure 5.1: Overview of Pony actor messaging at runtime

5.2 Compilation model and `libponyrt` runtime

As initially mentioned in section 4.5, Pony programs are compiled via LLVM, resulting in a final static binary of machine code that can be optimised ahead-of-time as part of the compilation, rather than most of the other ocap language options that involve a runtime interpreter or JIT (just-in-time) compiler. An overview of the compilation process is shown in Figure 5.2.

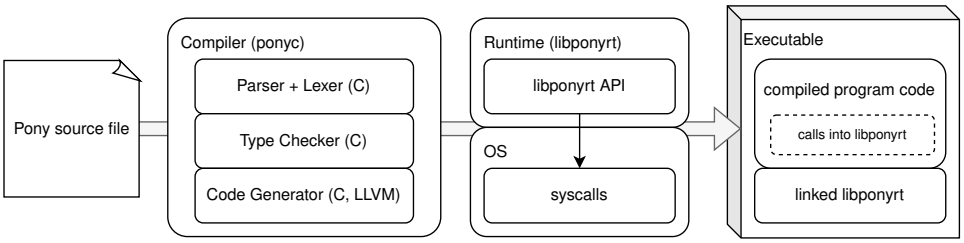

Figure 5.2: Overview of the Pony compiler, runtime, and output programs

Final output Pony programs ultimately consist both of some binary code that LLVM has produced as dictated by what Pony code the user wrote in the input source file, as well as a '`libponyrt`' library, which is written in C and distributed alongside the compiler. The program code part that is generated by the compiler will include many calls to `libponyrt` functions, which are ultimately connected to a compiled copy of `libponyrt` as part of a final linking process. The compiler and runtime are distributed as part of the `ponyc` project, which is available on GitHub [62].

Pony source files are first processed by a lexer and parser implemented in C, into an AST form in memory, which is then type-checked (again in the same C code), before finally being processed by a series of calls to LLVM via its C API to generate the program's code. This includes both a 'types' section where code and metadata for all the actors and classes is compiled and stored, as well as a `main()` function which bootstraps the runtime (see section 8.1 further ahead for more details about this).

At this stage, the generated program code can be inspected in the form of LLVM IR (intermediate representation), which the compiler can optionally output to a file. However in normal operation, the code is outputted as a compiled machine-code object file (via a call to `LLVMTargetMachineEmitToFile`), before being linked against a pre-compiled copy of `libponyrt` and threading libraries to produce a final executable binary file. (An optional `--runtimebc` compiler

flag can also be used to inline all the runtime calls from a pre-compiled LLVM IR bytecode file, which will allow for more aggressive optimisation by LLVM before producing the program's object file).

This compilation process was ultimately quite a natural environment to work with as part developing something for seL4, as C code and binary-level assembly are the common working environments when developing on top seL4 and its primitives.

5.3 Runtime components and API

The public functions provided by the `libponyrt` runtime are summarised below in Figure 5.3.

```
// runtime control
pony_init()
pony_start()
pony_stop()
// get global context
pony_ctx()
// threads
pony_register_thread()
pony_unregister_thread()
// allocate actor
pony_create(ctx, type)
// switch current actor
pony_become(ctx, actor)
// message-passing
pony_alloc_msg(size_index, id)
pony_send(ctx, to_actor, msg)
// allocate on current actor's heap
pony_alloc(ctx, size)
```

```
// scheduling
pony_schedule(ctx, actor)
pony_unschedule(ctx, actor)

// tracing for garbage collection
pony_trace(ctx, addr)
pony_traceknown(ctx, addr, type,
↪  mutability)
pony_traceunknown(ctx, addr, mutability)
// message-based garbage collection
pony_gc_send(ctx)
pony_send_done(ctx)
pony_gc_recv(ctx)
pony_recv_done(ctx)
pony_gc_acquire(ctx)
pony_acquire_done(ctx)
pony_gc_release(ctx)
pony_release_done(ctx)
```

Figure 5.3: `libponyrt` runtime functions

An overview of all the components of the runtime is also shown in Figure 5.4, adapted from the Runtime Implementation appendix of Clebsch's thesis on Pony [ClebschThesis, Appendix A].

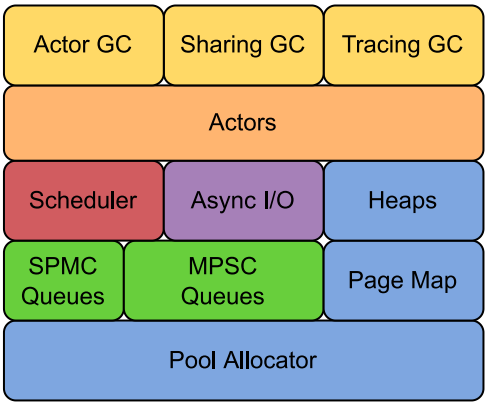

Figure 5.4: Components of the Pony runtime
Adapted from Figure A.1 of ClebschThesis

The lowest building blocks of the runtime can be broken down into three categories - memory allocation components, lock-free queues for communication and work-stealing, and thread management for scheduling. (There is also an ad-

ditional category in the form of the asynchronous I/O component, but this was not considered in depth within this project due to it being built on top of I/O kernel primitives on Linux, BSD/MacOS, and Windows, that are much higher-level constructed abstractions than those provided by the minimal interface of seL4.)

The pool allocator sits at the bottom, as it is used by basically all other components, providing a system for size-classed allocation[1] and freeing of memory in a per-thread and thread-safe manner. Allocations it is used for include actors, queues, messages, and the actor heaps used for object allocation.

Whilst the backing space for housing the per-actor heaps is allocated from the pool allocator, the heaps themselves are powered by separate heap code, shown by the separate "Heaps" component in Figure 5.4. The Page Map is another component, used as part of the garbage collection, to re-identify the actor that a heap object was allocated on.

The scheduler maintains runtime threads that each execute a main loop for finding and processing actors with work available, and processing their message queues, dispatching the appropriate compiled program code for each message's type. Queues of actors for work stealing are implemented using the SPMC queues - each runtime scheduler thread keeps its own queue that actors are pushed onto when they have messages added to them, and that the other threads can 'steal' from if they have no other work to do themselves.

Actors themselves, as a runtime datastructure, consist of:

- A pointer to their compiled type
- A (MPSC) message queue
- Local flags for runtime options / state
- A heap for object allocation
- A garbage collection info datastucture

The garbage collection components sit on top of all of this, as they send special runtime messages through the same actor message queues, alongside actual actor behaviour invocation messages generated by the running program.

Full details of the components and garbage collection scheme can be found in the extensive Appendix A of Clebsch's Pony thesis [ClebschThesis, Appendix A].

[1] Size classes range from 2^5 / 32 bytes, to 2^20 / 1 megabyte

5.4 Pony capabilities

Pony's OCap model is explained in a page from its tutorial:

> "A capability is an unforgeable token that (a) designates an object and (b) gives the program the authority to perform a specific set of actions on that object."
>
> So what's that token? It's an address. A pointer. A reference. It's just… an object.
>
> Since Pony has no pointer arithmetic and is both type-safe and memory-safe, object references can't be "invented" (i.e. forged) by the program. You can only get one by constructing an object or being passed an object. [61]

(There is also a noted exception that the C FFI (Foreign Function Interface) can break this guarantee - however it does not have to be used in all code written in Pony.)

When combined with the nature of the underlying Pony memory allocation system, and how ultimately *all* objects are allocated on per-actor heaps (even if they are later fully surrendered by the actor they were allocated on and sent off to other actors), Pony object capabilities can essentially be understood to be those heap allocations. No other, separate number is used to identify the object - throughout the whole language and runtime, objects are always identified by their memory address, and that address, or pointers to it, are the unique, unforgeable tokens of the capability model. This is covered in section 2.2.3 of Clebsch's thesis on Pony [ClebschThesis], but also highlighted specifically in chapter 6 when comparing the language message passing to Erlang:

> Erlang achieves fully concurrent passive object garbage collection by copying passive objects sent in messages to the *process-local heap* of the destination. This comes at a cost: copying the passive objects can be expensive when large data structures are passed between actors, both when the message is sent (due to the time taken to copy the message) and over time (due to the resulting increased memory usage). Copying message contents also means that *object identity* must be encoded in the data structure by the programmer, rather than being implicitly derived from the object's memory address. While this

> is less important for a functional language such as Erlang, it is important for an *object capability* language such as Pony. [ClebschThesis, Chapter 6, page 102]

There is an important note to draw from all this, which is that the current version of Pony assumes a single address space for execution, and that this is a key underpinning of how the pointers / memory addresses can be (and are) used as a reliable means of identity for the object capabilities.

Key Observation 1 *Pony in its current form assumes a single address-space, and this is a strong root of its object capability model - memory addresses can be used as unique identifiers, and are unforgeable due to its memory allocation model.*

5.5 Standard Library authorities

Pony's use of the object capability model is perhaps best exemplified through its standard library, the entirety of which has been designed with capability-based security in mind. Much of this stems from the use of 'static singleton' `primitive` types for various forms of authority. Some examples of this are shown in the following figures.

Figure 5.5 shows the various capability primitives used within the `net` package of the standard library. This package has a more extensive hierarchy of authority, with the base `NetAuth` stemming further down into specific capabilities for using DNS, UDP, and TCP, with TCP authority being even split further between both listening and working with an established TCP connection. The `TCPListener` and `TCPConnection` actors are shown with their constructors to illustrate that corresponding authority capabilities are required to use them, along with the `TCPListenNotify` and `TCPConnectNotify` interfaces that are used to call into when connections are established.

Figure 5.6 shows the various capability primitives used in the `files` package. To interact with the filesystem, you must construct a `FilePath` class to represent a path on the file system, which requires the `FileAuth` capability as an argument. The `FilePath` is also constructed with a bit-flag `caps` argument of possible file operation capabilities that the path can be endowed with, to allow differing levels of filesystem access via the constructed `FilePath` object.

Figure 5.7 shows the capability primitives used as part of the `serialise` serialisation package. There are explicit authority capabilities descending from the root `AmbientAuth` for serialisation and deserialisation, as well as two more specific capabilities for granting the rights to inspect data that has been serialised (`OutputSerialisedAuth`), and the rights to treat arbitrary bytes data as serialised data (`InputSerialisedAuth`).

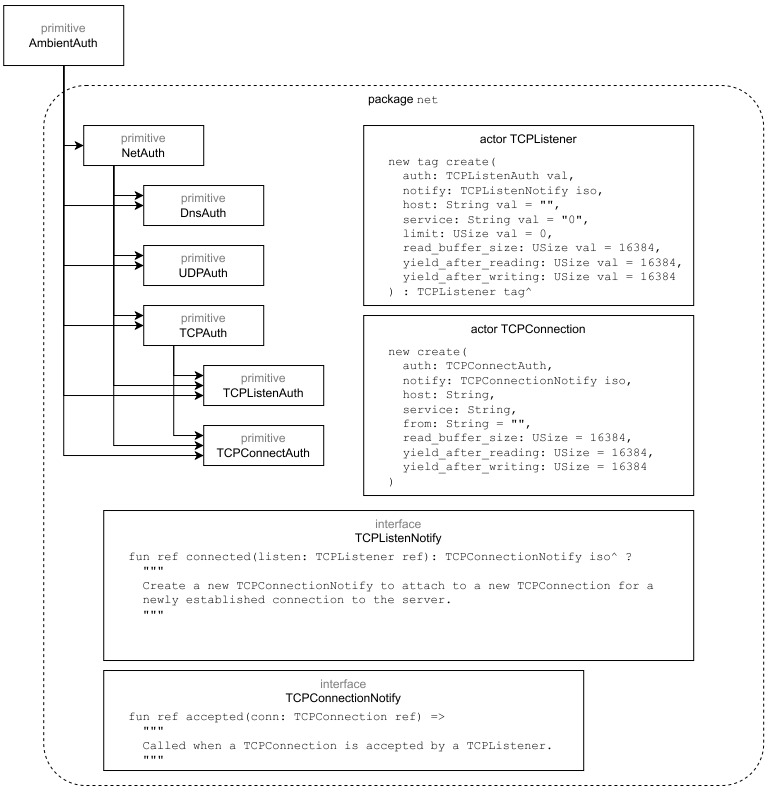

Figure 5.5: Authority primitives and assorted base types for the `net` package of the Pony standard library.
Adapted from the Pony standard library sources (`packages/net` in [62])

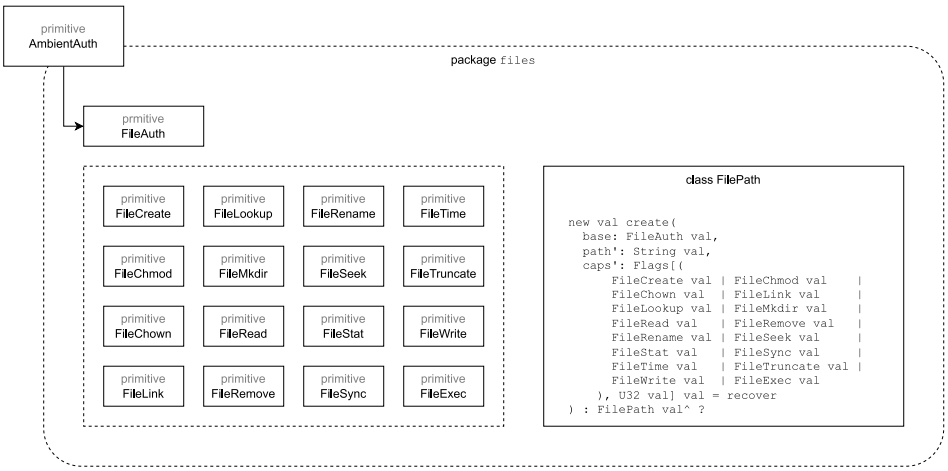

Figure 5.6: Authority primitives for the `files` package of the Pony standard library. The `FilePath` class is shown with its constructor to illustrate that the `FileAuth` capability is required as an argument for interaction with the filesystem.
Adapted from the Pony standard library sources (`packages/files` in [62])

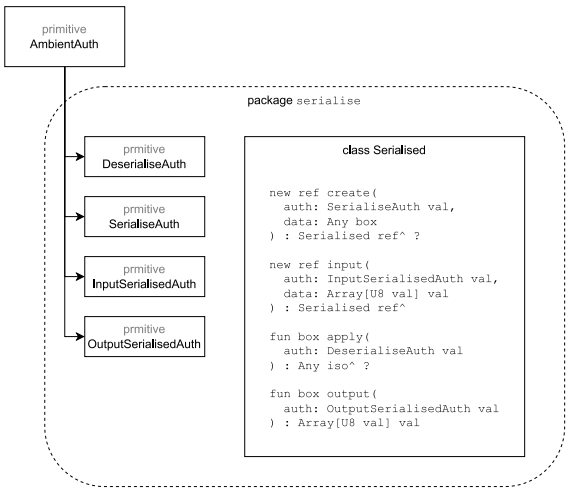

Figure 5.7: Authority primitives for the `serialise` package of the Pony standard library. The constructor and various methods of the `Serialised` class are shown to illustrate that capabilities are required for the various serialisation operations.
Adapted from the Pony standard library sources (`packages/serialise` in [62])

5.6 Sample Pony program

A sample 'ping-pong' Pony program is given below in Listing 1, for the purposes of giving a simple showcase of the message-passing functionality of Pony, and how its reference capability typing system is used to guarantee isolated, data-race-free access to objects.

In this program, the `Main` actor constructs a number of `Mailer` actors, and 'pings' each of them by invoking the `ping` actor behaviour on them. The mailers then later 'pong' back to the main actor via a separate invocation of the `pong` behaviour on the `Main` actor.

A `PingPongBall` class is also introduced, which the `Main` actor constructs but then 'kicks' over to the `Mailer`, which also then 'kicks' it back to `Main` actor. This helps show how objects can get 'moved' between actors, and how write access is safely handed over, highlighting the way isolated access to objects works, and the specific reference capability typing requirements required to do this.

A key restriction of Pony's actor model is illustrated by this program - behaviours cannot be awaited, they are send-only. To know when all Mailers have finished 'ponging' back, the Main actor would have to keep track of some sort of state that will track when the total number of 'pong' calls have come back, as part of either the `pong` or `pong_finished_callback` behaviours. An `if` check on this state in those behaviours could then be used as the trigger point to continue some work that needs to happen after all ponging is complete.

```
use "collections"

class PingPongBall
  var _id: U32
  var counter: U32 = 0
  new create(id: U32) =>
    _id = id

  fun box tostr(): String =>
    "id: " + this._id.string() + ", counter: " + this.counter.string()

actor Mailer
  var _id: U32
  new create(id: U32) =>
    _id = id

  be ping(receiver: Main, ball: PingPongBall iso, pass: U32) =>
    for i in Range[U32](0, pass) do
      receiver.pong()
      ball.counter = ball.counter + 1
    end
    receiver.pong_finished_callback(
      ("Mailer" + _id.string() + ": " + pass.string() + " pongs done"),
      consume ball
```

```
    )

actor Main
  var _env: Env
  var _size: U32 = 3
  var _pass: U32 = 0
  var _pongs: U64 = 0

  new create(env: Env) =>
    _env = env

    try
      parse_args()?
      start_messaging()
    else
      usage()
    end

  be pong() =>
    _pongs = _pongs + 1
    _env.out.print("    pong!!!")

  be pong_finished_callback(msg: String, ball: PingPongBall iso) =>
    _env.out.print("got msg: " + msg)
    _env.out.print("got ball: " + ball.tostr())
    _env.out.print("pongs is @ " + _pongs.string())

  fun ref start_messaging() =>
    for i in Range[U32](0, _size) do
      // construct a new ball to 'kick' over to the mailer.
      // constructed classes are `ref` rcap type by default, so need to use
      // `recover` to pull back to `iso` type to permit sending off to the
      // other actor.
      let newball : PingPongBall iso = recover PingPongBall(i) end
      _env.out.print("kicking ball: " + newball.tostr())
      // construct a mailer and call ping on it, kicking the ball over to it at
      // the same time. the newball variable must be consumed so that it can be
      // received as `iso` refcap type
      Mailer(i).ping(this, consume newball, _pass)
    end

  fun ref parse_args() ? =>
    _size = _env.args(1)?.u32()?
    _pass = _env.args(2)?.u32()?

  fun ref usage() =>
    _env.out.print(
      """
      mailbox OPTIONS
        N   number of sending actors
        M   number of messages to pass from each sender to the receiver
      """
      )
```

Listing 1: Sample ping-pong ‘mailbox’ Pony program.
Adapted from ‘mailbox’ in examples folder of ponyc repo [62], with addition of a `PingPongBall` class to illustrate reference capability typing required for sending objects to other actors.

5.7 'Causal' messaging

A property that is key to much of the design of Pony and its novel garbage collector is that of *causal* messaging. This is described in Clebsch's thesis as 'a message order guarantee wherein an *effect* (a message) does not get delivered until after all of its *causes*, where the causes of a message are every message that the sending actor has previously sent or received.' [ClebschThesis, §2.2.4]. This essentially means that the message queues of actors are always processed in order, messages are never selectively skipped or handled earlier than others, and that no jitter or delay might cause some message A_1 to arrive *after* any effects caused by some later subsequently-sent message A_2.

5.8 'Distributed Pony'

Pony in its current release-available form is ultimately still a single-process and single-node programming language, in that all of its actors and objects are only allocated within the one address space of a single process, and no actor-to-actor communication is available inherently at the language level without setting up explicit communication channels like sockets between separate Pony runtime instance.

However, work has been undertaken to develop a 'distributed' version of Pony, and Clebsch's thesis on the language even has many accomodations for a distributed design (see sections 3.2.3, 5.6, and 6.6 of ClebschThesis). A separate masters thesis by another early Pony developer, Sebastian Blessing, [11] investigated an attempt to create a version of Pony to support 'transparent distributed programming', which is defined as follows:

> Any application written in Pony should scale in a distributed network of runtime process without *any* changes to the code being necessary. This task is challenging, because we want to achieve that *any* conceptual property given by Pony in the concurrent setting also holds in a distributed context. [11]

One of the key issues in the distributed context is maintaining the causal message delivery that Pony is built upon. In particular, the (inherently-)distributed garbage collection is powered by a cycle detector that is reliant on message delivery between actors being causal, and thus handled in an expected sequence. Blessing's thesis focuses on the use of a tree-network topology, combined with properties of TCP (Transmission Control Protocol) to maintain causal message delivery guarantees [11, Chapter 3]. Clebsch's thesis, published around 4 years after Blessing's, proposes some extensions to the garbage collection protocols [ClebschThesis, §5.6 and §6.6] that would relax this requirement on causal order to pairwise FIFO order, but this does not yet appear to have been implemented.

Chapter 6

Comparison of related concepts in Pony and seL4

Whilst not completely equivalent in type, with one being a programming language and the other an operating system kernel, seL4 and Pony (and its runtime) are both 'systems' that can be compared on many similar points, or along common lines. This is a useful first starting point for designing any form of mapping of Pony onto seL4, to see what alignment can or cannot be found along these lines - several such comparison points are covered in the following sections.

6.1 Synchronous v.s. asynchronous models

One quite important foundational difference between Pony and seL4 is that their message passing models are somewhat mismatched. Pony is highly focused on asynchronicity, with the core message passing between actors being inherently asynchronous and send-only. seL4 IPC on the other hand is strongly focused on synchronous communication. The two models are explored and compared in detail in this section.

Message passing in Pony is inherently asynchronous. When an actor invokes another actor's behaviour (the core message passing action in Pony), a message is simply pushed to the message queue of the target actor (which is actually just a linked list of messages) - see Figure 6.1 - and the source actor continues its next line of execution. Critically, behaviours cannot be awaited - to have the source actor do something as the result of whatever the sent message is supposed to trigger, the actor itself must be passed as an argument for the target actor to (asynchronously) invoke behaviour of later once it is scheduled to handle the message it got sent.

This contrasts with seL4's model, which is for the most part inherently synchronous. When seL4 IPC `Send()`s or `Call()`s are conducted, they block the calling thread, and are typically expected to be immediately handled by a server thread that is blocked on an IPC receive - otherwise the send/call is queued for being handled by the first available receiver.

A Pony implementation on seL4 would need to be quite careful about this distinction, as the control flow of Pony's runtime scheduler threads (and when they move from one actor to another) needs to be carefully managed and reasoned about.

1: **function** MESSAGEQ_PUSH(messageq_t* *q*, pony_msg_t* *first*, pony_msg_t* *last*)
2: **atomic**(*last.next* ← `null`)
3: thread_fence()
4: **atomic**(*prev* ← *q.head*, *q.head* ← *last*)
5: *prev.next* ← *first*
6: **return** is_empty?(*prev*)

Figure 6.1: Pony runtime's message-queue push function, used for all message passing
Adapted into psuedo-algorithmic form from `messageq_push` in `libponyrt/actor/messageq.c` in the `ponyc` repo [62]

The kernel does also provide non-blocking send and receive functions, which can be used on IPC endpoints. However, there is a critical detail that the non-blocking send cannot be reliably used for depositing messages, as it only works as expected when a thread is already blocked and waiting on the endpoint, and does not give any indication of whether the sent message was actually delivered or not.[1,2] This is in part due to Endpoints themselves not being the storage location for IPC messages - on send, messages are either left waiting in the IPC buffer of the sending thread (when a blocking send occurs without a receive ready), or stored in the IPC buffer of the receiving thread (when a non-blocking send succeeds and the receive thread is already blocked and waiting). Endpoints themselves internally only contain pointers to thread control blocks.

Key Observation 2 *seL4 IPC does not support reliable non-blocking sending - an `NBSend` call to an Endpoint silently drops messages if a thread is not ready and blocked waiting on the other side of the Endpoint.*

The NBRecv method is designed primarily for use with seL4 Notification objects, not seL4 endpoint IPC. Notifications exist primarily to provide a means of synchronisation between threads, and do provide more of an asynchronous model - however, critically, they do not provide for capability transfer as part of their invocation, which Endpoint IPC does.

It is also worth noting that Send-only and Receive-only IPC is also even disadvised in general when developing for seL4, other than for "protocol initialisation or exception handling" - see the "IPC no-no's" section of [31].

Pony has been designed with an assumption that the message queues it uses are unbounded, to 'prevent both deadlock and incorrectly reporting resource exhaustion' [ClebschThesis, §A.6.1]. More broadly, it assumes that message sending always succeeds, and that it is always possible to successfully send messages. A strict mapping of Pony's message passing onto seL4's IPC / message passing

[1] From the seL4 manual: 'If the message cannot be delivered immediately, i.e., there is no receiver waiting on the destination Endpoint, the message is silently dropped.' [`NBSend` syscall in seL4Manual, §2.2]

[2] This lack of reliable non-blocking send is not a feature of all capability-based operating systems - for example, Barrelfish has a similar feature of endpoint-based IPC for same-core communication, with message sending that is *always* non-blocking, and returns a reliable error code if delivery cannot take place. Endpoints themselves are also buffered, allowing for multiple sequential sends, and ultimately just represent particular memory locations on a heap area of the 'domain control block' (rough equivalent to seL4 thread) [8, §5.1], making it more straightforward to multiplex sending.

mechanisms would not be able to provide these guarantees in an asynchronous context, without extra implementation on top of the mechanisms.

6.2 Message-passing message size

Message size is another aspect of these communication models to consider and compare. seL4 IPC messages are limited to a maximum size of 120 CPU words (defined from the `seL4_MsgMaxLegnth` constant in the lib_sel4 library) - on 32-bit architectures this totals 480 bytes, on 64-bit, 960 bytes.

Pony message sizes do not appear to have any upper bound - arguments are packed on to the end of message struct after a fixed-size standard header, so the total message size will be based on the size of the types of the arguments. Messages are always dynamically allocated, so the message size is passed to the allocator as an argument at runtime.

The conclusion to draw from this is that any mapping of Pony's language-level message passing onto seL4 IPC message passing would either have to involve variable-length transmission (which can be common for microkernel IPC protocols), or have to introduce strictly bounded message sizes at compile-time by only allowing enough arguments to fit in one IPC invocation.

6.3 Pony allocation sizes v.s. seL4 object sizes

Pony and seL4 both use capabilities to represent access to memory, albeit in slightly different ways, and, for the focus of this section, with different sizing considerations when it comes to areas of memory.

Pony's ocap model, covered in 5.4, ultimately means that all Pony objects are allocated by the runtime allocator, and all object references/capabilities are passed by the runtime as pointers to those allocations - albeit in a manner that is guaranteed to be safe by the type system and the manner in which the Pony compiler generates machine code (e.g. the compiler knows the sizes of all types at runtime and uses this to ensure machine code would never be generated that would access unmapped memory).

The appendix sections in Clebsch's thesis [ClebschThesis, mainly §A.3.1] detail this allocator, outlining its use of power-of-two size-classes for allocation, and that allocations are always rounded up to the first size class that fits them. Importantly, it also describes that the allocator's minimum allocation size is 32 (2^5) bytes - in part due to the design of the allocator and how it uses space in freed allocations to store bookkeeping data (specifics described in [ClebschThesis, §A.3.3]).

This contrasts somewhat with seL4's units for working with memory. System page size (usually 4KB, 2^{12}) is in many cases the smallest level of granularity that can be worked with, especially in the scenario of sharing or sending physical memory between protection domains. As mentioned in 6.2, IPC messages are also limited to a max size of 480 or 960 bytes.

A takeaway from this comparison is that, for Pony objects between the max IPC size and the memory page size, either multiple IPC calls are required for transferring the object's data between protection domains, or full single pages are required per-object, which would potentially waste an amount of space.

6.4 Memory Address spaces

Memory addressing is fundamental to both any programming language and any operating system, and as such is an important element in the design of both Pony and seL4.

A large part of how seL4 is able to provide its strong isolation guarantees comes from how it does virtual memory management. Components are typically strongly isolated into separate address spaces, with explicity-supplied communication methods (IPC or explicitly shared memory) being the only way to communicate between these address spaces.

Pony's current implementation is built around quite a strong single-address-space assumption, as covered in section 5.4. Objects in Pony are identified via the memory address of their heap allocation, and communication between actors is implemented via linked lists within the same address space. This proves an issue when it comes to designing any mapping for implementation, as will be discussed in section 7.

6.5 Capability enforcement / Trust boundaries

Pony and seL4 both enforce trusted separation of resources using a capability model, albeit via different means.

In seL4, this separation is enforced via its system of CSpaces / protected tables of capabilities, and more specifically **runtime checks** performed by the kernel when handling system calls (which are the means of 'invoking' capabilities).

Pony on the other hand can enforce much trust separation without runtime checks, due to the nature of its type system - in fact it was built with a goal to prove that many such runtime checks could be elimintated by a strong type system [ClebschThesis, §1]. This means that the enforcement of its object-capability system is mostly performed via the compiler at project **compile-time**, with assumptions that the generated code, its use of the runtime, and the runtime itself (e.g. the memory allocator) is safe and correct. These assumptions underlie guarantees that objects will never be allocated on the same memory area, and that the ownership, isolation, and sharing rules of the language along the lines of the type system are enforced, which ultimately provide the object capability safety model of the language.

Even runtime checks such as file operation permissions are still guided by the type system, as the checks performed are still against static types. An example of this is shown in the `rename` method of the `FilePath` class from the Pony standard library below in Listing 2.

```
// abridged snippet from file_caps.pony
use "collections"

type FileCaps is Flags[
  ( FileCreate
  | FileChmod
  | FileChown
  | FileLink
  | FileLookup
  | FileMkdir
  | FileRead
  | FileRemove
  | FileRename
  | FileSeek
  | FileStat
  | FileSync
  | FileTime
  | FileTruncate
  | FileWrite
  | FileExec
  ),
  U32 ]

// abridged snippet from file_path.pony
```

```
use @rename[I32](old_path: Pointer[U8] tag, new_path: Pointer[U8] tag)

class val FilePath
  """
  A FilePath represents a capability to access a path. The path will be
  represented as an absolute path and a set of capabilities for operations on
  that path.
  """
  let path: String
    """
    Absolute filesystem path.
    """
  let caps: FileCaps = FileCaps
    """
    Set of capabilities for operations on `path`.
    """

  // ---

  fun rename(new_path: FilePath): Bool =>
    """
    Rename a file or directory.
    """
    if not caps(FileRename) or not new_path.caps(FileCreate) then
      return false
    end

    0 == @rename(path.cstring(), new_path.path.cstring())
```

Listing 2: Cut-down view of the `FilePath` class in the Pony standard library, illustrating a runtime capability check.
Taken from the 'files' package of the Pony standard library, available in the `ponyc` repo [62]

The usefulness of this resource separation enforcement is best understood or examined by looking at the scenario of 'untrusted' code execution. On seL4, if you have code that you do not trust, you would normally either execute it within a virtual machine, or some other locked-down protection domain, where the CSpace of that domain only has capabilities that allow it to do explicitly what you want to allow the code to do with the rest of your system (typically, in the form of capabilities for endpoints to other components). The checks performed by the kernel give this separtaion / isolation guarantee.

In Pony and other object-capability languages, the trust boundary is similarly the *object* capabilities handed to the untrusted code - in a well-designed ocap system with no global variable authority, all that malicious code is capable of doing is via the objects specifically handed to it when first called. In Pony specifically, system operations such as filesystem and network access can only be performed via explicitly-passed object capabilities that derive from the `AmbientAuth` capability

handed to the `Main` actor of any Pony program. (see section 5.5 for an overview of some of these authority capabilities).

A limitation of Pony is that it does not currently have the ability to dynamically evaluate code, as in other ocap languages like Caja [46] (with its motivating use case of running unknown 3rd-party scripts atop a pre-existing javascript environment on a web page). Pony also does not quite support the ocap paradigm of creating limited execution contexts, such as the `confine(exprSrc, endowments)` function[3], or more recently, 'realms' and 'compartments'[4], from the Secure EcmaScript (SES) system. However, there is still a motivating use case for Pony's ocap system of including 3rd-party-authored library code into a Pony project. Because of the object-capability guarantees, you can be sure that code you use from a 3rd-party library will only ever be able to perform certain classes of actions (e.g. use the filesystem, or use the network), if the actors / classes / functions / interfaces of that code were *explicitly passed* the authority to do so.

[3] See [47, §2.3 SES: Securing JavaScript] for definition / more info

[4] See `README.md` for SES in the `packages/ses/` folder of [1], and the associated TC39 proposal for standardising Compartments within the ECMAScript / JavaScript standard [59]

6.6 API contracts over message-passing channels

Message-passing as a mechanism is only useful to the degree of any agreement between either sides of the communication channel on how to communicate.

In seL4 userspace, having a capability to an endpoint is only useful so long as you know how to talk to whatever is on the other side of it. (The same can also be said of non-IPC capabilities for objects such as `Untypeds` and memory objects - when it comes to using them, it is a matter of knowing the right `message` format to deliver to the `Call()` syscall interface, as covered in section 1.2). There can also be performance concerns when it comes to communicating over an endpoint - the IPC 'fastpath' only works when no capabilities are being transferred as part of the IPC call and when the message is small enough to fit in a dedicated subset of message-passing registers[5].

Additionally, the capability transfer mechanism provided by seL4 IPC does not by itself give any information about the type of the capability that comes through when receiving a transfer (see [66, §4.2.2 Capability Transfer]) - for this to be understood, there must be something within the contract between sender and receiver to dictate the information (or assumption) of what capability type has come through as part of the transfer.

This leads to the API contract across an Endpoint channel being quite important. In CAmkES [26], components are specified with 'procedure' APIs that give C-like arguments and return types - see Figure 6.2 - and which ultimately are consumed in other C files by generated C functions with equivalent type arguments. As another example, the seL4 Foundation's `libsel4rpc` library [6] provides a means for 'clients' to request resource capabilities from some other 'parent' / 'server' over an Endpoint channel, with a simple typing contract established through the use of Google's 'Protocol Buffer' data serialisation framework [32][7]. However, as `libsel4rpc` is still ultimately implemented in C, no language-level type information can be leveraged to make the message protocol come 'for free' or be automatically type-checked - a 'type' field in the message struct still has to be

[5]The number of registers for fastpath IPC is specified by the `seL4_FastMessageRegisters` constant, which is anywhere between 1 and 4 depending on the architecture the kernel is targeting and what feature flags were set during compilation.

[6]See the `libsel4rpc` folder of the `seL4_projects_libs` project / repository: `https://github.com/seL4/seL4_projects_libs/tree/master/libsel4rpc`.

[7]Note that `libsel4rpc` uses `nanopb` for its protocol buffers specifically, an ANSI C implementation of the framework - see `https://github.com/nanopb/nanopb` or `https://jpa.kapsi.fi/nanopb/docs/`

```
procedure DHCP {
    uint32_t discover(in uint64_t hwaddr, out uint32_t siaddr);
    uint32_t request(in uint32_t ip, in uint32_t siaddr);
}

component Client {
    control;
    uses DHCP dhcp;
}

component Server {
    has mutex lock;
    provides DHCP client1;
    provides DHCP client2;
    provides DHCP client3;
    provides DHCP client4;
}

assembly {
    composition {
        component Client a;
        component Client b;
        component Client c;
        component Client d;
        component Server s;

        connection seL4RPCCall c1(from a.dhcp, to s.client1);
        connection seL4RPCCall c2(from b.dhcp, to s.client2);
        connection seL4RPCCall c3(from c.dhcp, to s.client3);
        connection seL4RPCCall c4(from d.dhcp, to s.client4);
    }
}
```

Figure 6.2: Example CAmkES spec for a DHCP server, showing the C-like API defined in the `DHCP` procedure block.
Adapted from `camkes/apps/dhcp` in the main / example apps CAmkES repository (`https://github.com/seL4/camkes`)

manually inspected, and the correct C structs still have to be manually selected via a generated union to get the correct access to the correct message / struct format depending on the 'type' field.

In Pony, this kind of API contract across the message-passing primitive of actor-to-actor communication is defined by the behaviours on actor types defined in program source code and their type signatures, and thus set / fixed at the point of compilation. Behaviours are ultimately compiled out with various identity numbers, unique within the scope of a particular actor type, that are used by any code generated to invoke those behaviours on actors of that type - the relevant number of the destination behaviour is what is inserted into the `id` field of the message sent to the destination actor, and the arguments of the behaviour are simply appended to the end of the message after a standardised message header.

In fact, Pony does not provide any language-level way to use the lower-level raw message-passing functionality (i.e. send a message with arbitrary integer ID) at all - in part because the message-passing is also used by the runtime for garbage collection and various other purposes, which use some special reserved message IDs, but also in part because the static type system of behaviours provides a set of guard rails to prevent incorrect messaging behaviour. This latter point is important to note, because the fact that these guard rails are a static assumption of the language eliminates the requirement for any support for error handling if incorrect messaging were to occur, especially when combined with the design of Pony's message passing being send-only, as covered in section 6.1. (This approach of eliminating classes of error-handling through static typing is also taken by the C++ Actor Framework [16, §5.2].)

Finding some way to leverage this typed message-passing system on top of seL4 IPC thus became an attractive goal of the research, as will be covered in the next section - especially if the seL4 IPC capability transfer could somehow also be modelled as part of this API model.

It is worth noting that the lack of access to lower-level message-passing is not necessarily the case in other ocap languages. In E [45, 43], remote calls are made using an 'eventual-send' syntax that in the end is just sugar-syntax for `E.send(dest, "method", arg1)`, which returns a Promise representing the eventual result of the call (the `E` static is part of of a helper library called ELib, a Java library distributed as part of the language). Dr. SES [47] has a similar `!` 'bang' operator that is similarly sugar syntax for `Q.send` - see Figure 6.3. In these situations, the static guarantees present in Pony's system no longer apply, so an error-handling mechanism is required: if the message-passing is made with an incorrect method name, the promise is 'rejected' into an error state.

Immediate syntax	Eventual syntax	Expansion
`p.m(x,y)`	`p ! m(x,y)`	`Q(p).send("m",x,y)`
`p(x,y)`	`p ! (x,y)`	`Q(p).fcall(x,y)`
`p.m`	`p ! m`	`Q(p).get("m")`

Figure 6.3: Examples of the infix `!` / "eventually" operator from Dr. SES and its '`Q`' library.
Taken directly from [47, §2.4, page 7]

The idea of having 'contracts' over 'channels' is also a key concept of the Microsoft Singularity project, as briefly mentioned in subsection 4.6.3 of the ocap

language review.

6.7 Authority for memory allocation

As discussed in section 1.7, a core problem involved in the design of systems on top of seL4 is that of memory allocation. Systems are often built with some form of 'memory server' that is given all the `Untyped` capabilities that are provided from the boot process. This server can be set up with whatever allocation policy is appropriate for the system at hand being built - i.e. some trusted tasks may be given higher memory request quotas than others. It is not uncommon for some tasks to be constructed with *no* access to the memory server - i.e. they will only ever be able to use the amount of memory they were initially set up with.

Pony on the other hand makes quite a strong assumption that memory is freely available and that it is always possible to ask for more. Because message queues are unbounded, they can in theory The memory allocation infrastructure of the language does not allow for handling the case of running out of memory (as can be seen in Figure 8.3, which will be covered later in chapter 8).

More generally, it can be argued that most programming languages (e.g. those built on POSIX foundations and assumptions) come with an *ambient authority for memory allocation.* Capability-based systems like seL4 where memory is accounted for through the capabilities introduces the possibility for additional memory to, in theory, come from multiple possible authorities, or to not even be accessible at all. Even an object-capability language like Pony does not model this - although it could be argued to be out of scope for the language given its strong implicit reliance on regular memory allocation for actor-to-actor communication. This could be an interesting avenue for further object-capability language research.

Key Observation 3 *Pony, and even object-capability languages more broadly, do not include any explicit modelling of authority for the right to request more working memory, which is a somewhat unique feature of confined-component systems built on a microkernel like seL4.*

Chapter 7

Possible useful Pony ocap models for seL4 programming

Now that a background of seL4 and Pony has been established, and that a comparison has been done of their related concepts, we can come to the task of attempting to overlay the two capability models.

The stated goal of this thesis / project (as per chapter 3) was to see if there was any way to use an object-capability language to make programming a capability-based operating system *easier*. Hence it was not simply just a question of getting the language to 'work', but to see if there were ways the language could be adapted to the operating system's capability environment in a way that could make the operating system capabilities more 'natural' to work with.

Four possible such 'models' for implementing Pony on seL4 were thus explored - each with a different particular focus in terms of similarities between the language and the operating system that could perhaps be lined up in a useful way:

7.1 Handing off/around seL4 IPC endpoint(s) for talking to objects

An early possible language-to-OS caps model considered was the idea of using seL4 endpoints to represent remote Pony object capabilities, as 'the rights to talk to an object', given Endpoints are already capability objects on seL4, and a primary means of seL4 cross-domain communication. However, this became less obviously feasible as the work of comparing seL4 and Pony (chapter 6) progressed.

The main issue with this model is with the blocking nature of the interaction with Endpoints, as covered in section 6.1, and critically, the fact that non-blocking send cannot be used as a means for reliably delivering messages (Key Observation 2). In a situation where individual endpoints were used for representing every remote object, there would thus be a reliance on a thread calling `Recv()` on the other side to prevent any invocation of the object from becoming a blocking operation. Threads cannot call `Recv()` on multiple objects, and even with the facility of `NBRecv()` allowing for a thread to check if work is available, the endpoints would need to be checked in some sort of loop that would increase with size with the number of objects at play, and induce an ordering decision over how to check all the available endpoints for messages. With a large number of actors/objects, this would likely soon blow out to become a situation involving a very large number of threads required.

Badging multiple capabilities to the same endpoint with the memory addresses of the object capabilities could be one way around this, as then the side giving out the endpoint would only need to blocking receive on a endpoint per communication channel. However, this was left outside the scope of the project and not considered further.

Given these issues (and mainly the issue of multiplexing over many endpoints), an alternate approach was developed for investigating further, where message-transfer between protection domains would be multiplexed through some kind of "message pump" endpoint. This, and other broader issues involved in working with remote actors / objects is covered in the next section.

7.2 Remote actor communication through message pump endpoint

A conceptually simple starting place for an integration of Pony that works both with seL4 confinement and Pony's inherent actor model is one where actors are somehow spread across seL4 protection domains (CSpaces / address spaces). This would be roughly analagous to the ocap paradigm of 'vats': spaces within which sets of objects exist and can be addressed via.

There are several problems introduced at this point, which make any implementation complicated, as it must address all of them in some way:

- Naming problem - how to identify and/or refer to specific remote actors? Across address spaces, Pony's current ocap model of "the actor's identity is its address" won't hold.
- Bootstrap problem - how to obtain a remote reference to a remote actor in the first place?
- Explicit v.s. implicit remoteness / placement problem - should remote actors be represented any differently to local ones? Should their APIs be any different or should there be a design goal to keep them the same? Should actors be explicitly placed / set up in remote domains, or will their setup be more automated?
- Communication mechanism - by what means can actors talk to each other across protection domains?
- Communication channels - by what topology do actors communicate to other actors? Do they have direct channels between them or do they communicate via a mediator?
- seL4 capability transfer - can the actor messaging be set up to support moving capabilities from one domain to another, as well as simply passing values and actor IDs as per the current Pony model?

The naming and placement problems are addressed to some degree by Sebastian Blessing's Distributed Pony thesis [11], as introduced briefly in section 5.8. For naming, actors are still identified using their allocation memory address on the node they are allocated on, but also combined with a 'node ID' that is unique

for each node in the 'cluster of Ponies'. A trick similar to the process of 'pointer tagging' is used for achieving unique identification across the cluster within only 64-bits, by leveraging an observation or assumption that 'The memory subsystem actually only uses 48-bits for main-memory addressing.' [11, §3.5.1 / page 46] (This assumption is difficult to cite or confirm, but appears to be an assumption that can be made for current x86-64 systems - 48 bits allows for addressing 256 terabytes of memory, an upper limit that is quite assumable for current hardware). This is combined with an observation that 'Pony's pool allocator aligns any allocation on a 64B boundary. Hence, the lower 5 bits of any memory address are always zero.' [11, §3.5.1 / page 46-47][1] This leaves 21 bits spare for storing a node ID, giving a roughly 2 million upper limit to the number of nodes. A similar approach could probably be used for seL4 protection domains, so long as they are identifyable within the running seL4 system by a reliable ID.

The communication problems are addressed next. On seL4, the two main means of cross-domain communication are either IPC through endpoints, or shared memory mapped into both address spaces.

seL4 IPC comes with the general requirement of being synchronous, which presents as a mismatch of sorts with Pony's asynchronous message-passing model, as discussed in section 6.1. Shared memory is usually the go-to means for asynchronous communication in seL4 systems - However seL4 capability transfer cannot be directly facilitated this way, without some other control mechanism over a communication protocol on the shared that involves making CSpace modification operations. (An approach like this would be similar to how the Barrelfish operating system handles cross-core capability operations, using a userspace 'monitor' - see §4.1 and Chapter 5 of [57], which covers various possible approaches to handling capabilties distributed across multiple cores). This approach was not explored as part of the project due to the increasing stack of implementation requirements on top of the initial hurdle of just getting the existing Pony runtime elements working in the seL4 environment, but could potentially be explored in other work. A shared-memory approach might also make sense for a Pony implementation on seL4 that does *not* include language-level support for capability referencing and transfer between protection domains, but since leveraging a language to make operating system capability management *easier* was the goal of

[1] This 64-byte alignment may have been a historical assumption a different version of Pony from the time of the thesis (~2013) as the current release Pony allocator has a 32-byte minimum allocation size.

this project, this out of scope and not explored either.

Blocking behaviour in Pony's message passing is undesirable as the language has been built around quite a strong non-blocking assumption to prevent deadlock (see [ClebschThesis, §2.1]). However, decoupling the blocking behaviour via means of an intermediate 'message pump' thread could alleviate this. Whilst seL4 IPC is blocking, it is generally designed to be used under an assumption that the thread on the other side of the IPC will already be blocked on a prior `Recv()` or `ReplyRecv()` call, ready and waiting to receive data. It is also in general designed primarily for providing a fast context switch between protection domains on the same core [31], as part of its heritage from the original L4 microkernel. Cross-core IPC *is* supported, through means of using the thread IPC buffers, but is not recommended [31, "IPC no-no's"], cannot provide the usual direct control flow switch associated with L4/seL4 IPC, and comes with a performance hit as the receive thread will only resume when re-scheduled on the other core.

Under this model, to provide for fast cross-core message delivery, a possible solution could be to have "message receiver" threads set up within each protection domain for any remote CPU core that needs to be able to deliver messages into the current protection domain, with each receive thread having its affinity set to that source core, and having an associated message pump endpoint set up that the source protection domain would have a capability to. These receive threads would be assigned to a simple classic seL4 server `ReplyRecv` loop that only receives messages, then finds the appropriate destination actor for a received message, pushes the message onto its queue (which is a lock-free, thread-safe operation), then returns back to the `ReplyRecv` call, handing control flow back to the actor that sent the message. This would be likely to be a fairly quick operation with direct control flow transfer, maintaining the design property of Pony that message delivery is quick and non-blocking.

In a simplified environment of one protection domain (and thus address space of allocated actors), each protection domain would have $N_{cores} - 1$ receive threads, alongside the Pony runtime thread for that domain's core, totalling $N_{cores} \times (N_{cores} - 1)$ extra threads and endpoints. This arrangement is shown below in Figure 7.1.

Ultimately, this cross-core message delivery system was not able to be implemented or tested in this project due to the preliminary work required in getting the Pony runtime components working on seL4, as covered in chapter 8, but could be an approach to experiment with in the future.

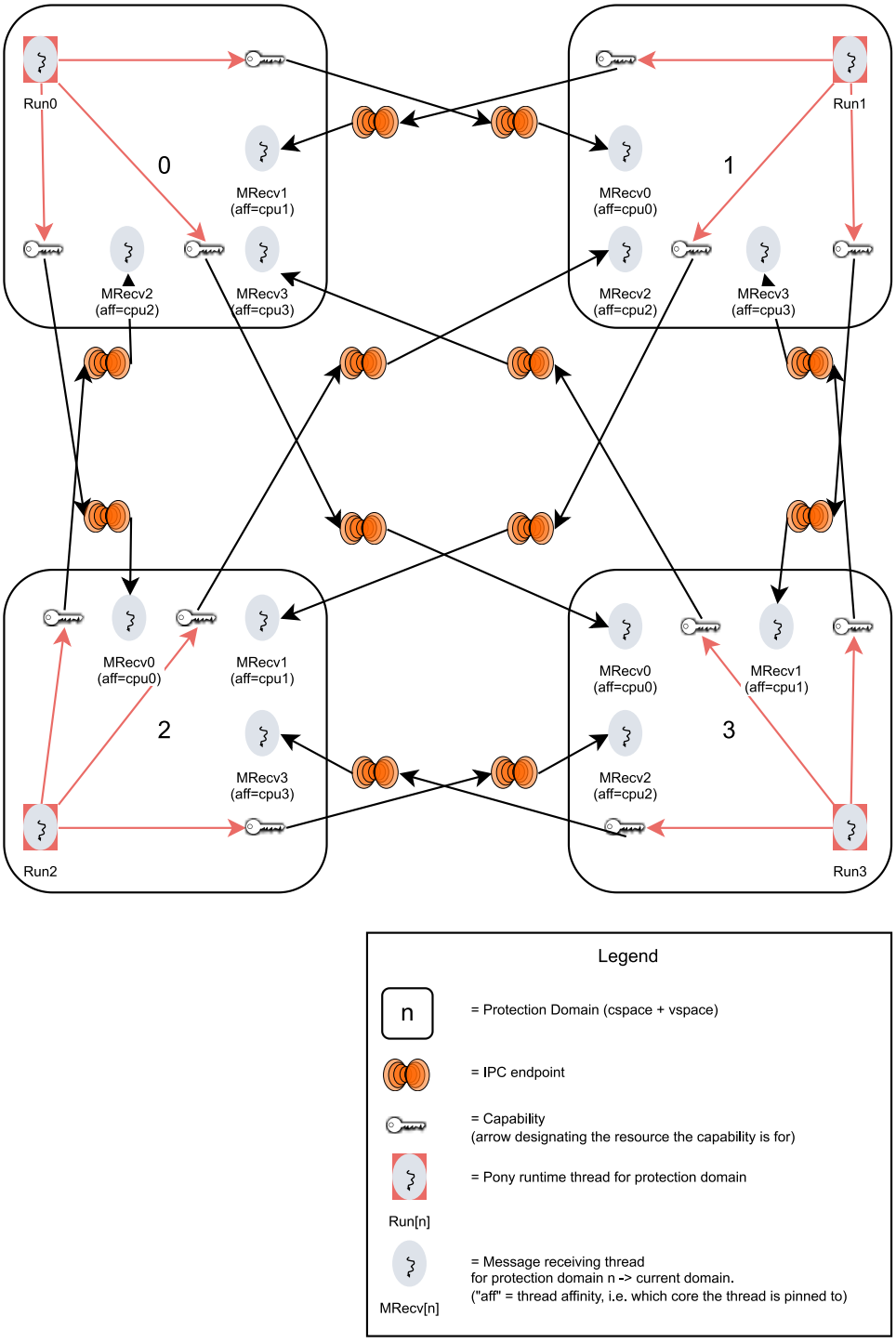

Figure 7.1: Example of seL4 threads and endpoints required for cross-domain message queue message-passing support for a quad-core, quad-domain Pony runtime

7.3 Handing off physical memory that contains data

Another way of aligning concepts in Pony and seL4 that was considered was to see if the "handoff" nature of Pony's reference capability system when message-passing could be leveraged to represent handing away mapped memory from one seL4 protection domain across to another address space.

When making a Pony message-passing call to another actor, if a variable is being passed along with the message, there are various typing guarantees that must be met depending on how the variable will be used on the other side. A simple example of this is shown below in a snippet from the code sample of Listing 1, where for the `Main` actor to give a `Mailer` actors `iso` write access to the `PingPongBall`, the `Main` actor must surrender its own access to the `PingPongBall` by 'consuming' its own reference to the ball (and thus removing its ability to continue to access the ball further along in the code / current scope).

```
// within a function body in 'actor Main':
  let newball : PingPongBall iso = recover PingPongBall(i) end
  _env.out.print("kicking ball: " + newball.tostr())
  // construct a mailer and call ping on it, kicking the ball over to it at
  // the same time. the newball variable must be consumed so that it can be
  // received as `iso` refcap type
  Mailer(i).ping(this, consume newball, _pass)
```

Under the right conditions in concert with managing seL4 capabilities for physical memory, this kind of handoff could be represented on seL4 by the process of physical backing memory at some virtual address being unmapped from the current virtual address space, and then transferred via seL4 IPC. This is in some ways similar to the `splice()` system call on Linux [60], which allows moving data between file-descriptor objects such as pipes, but on seL4 this would be at a much more granular level of unmanaged physical memory capabilities.

As per the discussion in section 6.3, this would have a limiting smallest-allocation size of a system page (i.e. 4kb). Additionally, large objects would have to be either allocated on larger pages, or carefully kept mapped via multiple sequential pages. This model also, as per discussion in section 7.2, hinges on some ability to either establish a Pony runtime with actors in separate address spaces, or to establish a communication mechanism between actors running in different address spaces.

With a functioning Pony runtime for seL4, this model could potentially be investigated and evaluated (both in terms of performance and usability). However as a fully-functional runtime could not be achieved in the scope of this project, this was not possible, and was not investigated any further.

7.4 Embed seL4 capability types into Pony types

The first place to start for an implementation of Pony on seL4 should arguably be to get seL4 capability types and their invocations represented within the language somehow. As code written in Pony on its own (without custom C extensions) cannot invoke system calls directly, some types representing seL4 primitives are essentially required to make working with them in any Pony program possible.

There is an interesting challenge involved in trying to maintain object-capability safety for userspace representations of capabilities, which are ultimately just stack or heap allocations containing pointer-like numbers for objects in some foreign address space. seL4 object types could be defined in Pony that accept capability addresses as their constructor argument, but this would essentially break the capability model, or provide an unsound one, as they could be constructed with arbitrary number arguments. On the seL4 system side of things, security would still be maintained, as the capability addresses are still validated by the kernel - capability operations with invalid addresses would simply fail with error responses.

Ideally, similar to the object-capability model, capabilities should only be accessible if you start with them (e.g. from a root program argument), or if you are se

This would likely require the functionality of private constructors, which is something that Pony does not seem to support in its current version.

Regardless, even Pony types for seL4 kernel objects with capability address constructor arguments would be useful, as it would provide for a more logical object-oriented programming model for things that are ultimately kernel objects with various ‘methods’ available for them, depending on their type.

Chapter 8

Porting the Pony runtime environment to seL4

As part of this project, work was undertaken to port Pony's runtime environment into a native seL4 project / environment, with the goal of being able to support running compiled Pony programs linked against an seL4-friendly `libponyrt`, or at least some form of skeleton Pony environment to test various integration models with more manually. Due to time constraints, and the complex tradeoffs involved in possible integration models (as covered in chapter 7) making it unclear which integration path forward was worth committing to trying to implement, a full runtime environment was not quite achieved, but several components of the runtime were nonetheless successfully ported. This section will discuss the steps the port was broken down into, and the work involved in performing the port [1].

To briefly recap relevant parts of chapter 5: Pony programs are ultimately compiled code blobs from LLVM containing calls to various pony 'runtime' functions, which then get linked against a separately-compiled `libponyrt` library (written in C) which provides the implementations of all these runtime functions. Thus porting the code of `libponyrt` into an seL4 environment became the main first goal of the implementation / porting effort.

[1] The port was using commit `ca6d725043f3637b6e4246450945b9d6f3778a80` of the `ponyc` codebase, from March 19th, which was the HEAD commit around that time and only a few commits ahead of the v0.49.1 release of the language

8.1 Which runtime components to port first? Analysis of the `main()` procedure of a Pony program

To avoid the infeasable approach of simply putting all the `libponyrt` code into an seL4 environment and trying to fix any/all compiler errors presented, a piecemeal approach was taken of porting small parts of the runtime function-by-function / file-by-file.

This raised the question of which runtime elements to port first. To guide choosing elements, the `main()` procedure of a small Pony program was analysed to see what runtime calls it used specifically, and thus which calls would be the minimum necessary to port to support a small / trivial Pony program.

Compiled Pony programs are produced via the Pony compiler's use of the LLVM C API. Programs can be outputted/viewed as LLVM IR, but disassembly of the final binary offers the best view of the program runtime functions that are used to kickstart the Pony program. The `main()` procedure of a sample Pony program is shown in Figure 8.1.

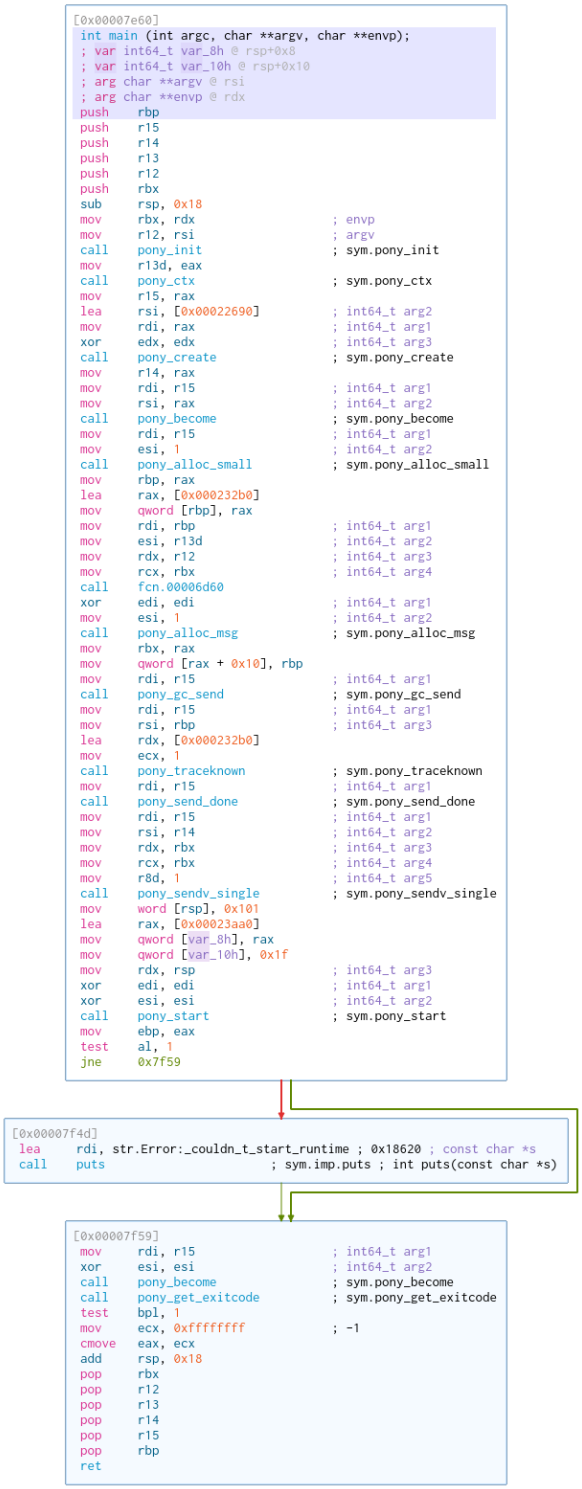

Figure 8.1: Disassembly of the C `main()` procedure of a compiled Pony program

From this disassembly, we see the following runtime functions used:

1. `pony_init`
2. `pony_ctx`
3. `pony_create`
4. `pony_become`
5. `pony_alloc_small`
6. `pony_alloc_msg`
7. `pony_gc_send`
8. `pony_traceknown`
9. `pony_send_done`
10. `pony_sendv_single`
11. `pony_start`
12. `pony_become`
13. `pony_get_exitcode`

These can be loosely grouped together and the main program startup explained with more detail / context as follows:

1. Initialise the Pony runtime environment (runtime call 1: `pony_init()`). This mainly sets up the schedulding-related information (e.g. determines number of threads to use).

2. Create (i.e. allocate) the 'main' actor and switch the runtime context to it (runtime calls 2-4. `pony_ctx()` simply fetches the global Pony context struct and is used to get it into a variable for calling other runtime functions with).

3. Create (i.e. allocate) the `env` / 'environment' struct, which is the constructor argument for the Main actor. (This gets allocated on the main actor's heap and is what runtime call 5 (`pony_alloc_small()`) represents.)

4. Create (i.e allocate) an initial bootstrap message for the Main actor (runtime call 6). The message will have an implicit 'id' ('type') of 0, which corresponds to calling the compiled 'dispatch' function of the actor, and the first value inside the message is set to be a pointer to the env struct. This is how the env struct is 'delivered' to the Main actor, and, critically for the object-capability side of the language, acts as the one sole point that the root `AmbientAuth` capability is delivered in to the program.

5. Step through all the message-based garbage collection-related steps for this initial message and its contents (runtime calls 7, 8, and 9)

6. Put the message on the Main actor's queue by sending the message to it (10). This will implicitly cause the Main actor to be put on the global work queue.

7. 'Start' the runtime, which will create and start all the runtime scheduler threads, then 'join' on all of them in series (i.e. block until they are all complete). Each runtime thread continually tries to pop an actor from first the global work queue, or failing that, its own local queue, and will then process the popped actor's message queue, until queiescence is detected (i.e. all scheduler threads have empty queues - see [ClebschThesis, §A.12.3] for more info on this).

8. At this point after all the scheduler threads have completed, clear the actor pointer (`pony_become(NULL)`, then get and return the global-variable-stored exit code from the runtime environment (runtime calls 12 and 13).

From here, it became clear that the key required components to port were going to be:

- Actor allocation (which uses the underlying pool allocator)
- Pool allocator
- Actor heaps
- Message allocation (which again uses the underlying pool allocator)
- Message queues, sending (queue push), and consumption (queue pop)
- Scheduler, work queue(s)

The garbage collection components are arguably 'required' too, but were left out of scope to focus on a *minimum* amount of work to get something running, especially due to the complexities of the garbage collection process, and its strong reliance on actor-to-actor communication (which could potentially be quite different if a model involving seL4-confined actors was approached).

Of all these components, the lowest-level one depended on *most* was obviously the pool allocator, which became the first component to try and port.

8.2 Step 1: base seL4 environment

Before even porting anything though, a base seL4 environment was required to port into, and some basic seL4 components that were likely going to be required for using as part of the port.

Starting development of an seL4 system from scratch can be a daunting task due to the number of userspace components required to get even a basic system running. It is more common to start with CAmkES and set up an appropriate set of existing components to build a static system out of. However CAmkES itself comes with a lot of prescriptive initialization code and root task that sets up the static system defined in the component assembly, and as this project was potentially for investigating more dynamic interactions between seL4 capabilities, it was decided to instead start from scratch with a custom root task.

seL4 development almost always involves the construction of userspace code at the same time. As such, development is usually conducted through a 'project', which is a collection of Git repositories including the kernel itself, which are assembled together.

The seL4 release process includes the maintenance of several such 'projects' which are basically the available starting places for seL4 system development [67]:

1. sel4test-manifest
2. sel4bench-manifest
3. camkes-manifest
4. camkes-vm-examples-manifest
5. sel4webserver-manifest
6. sel4-tutorials-manifest
7. rumprun-sel4-demoapps
8. verification-manifest

The CAmkES manifests were excluded given the reasons mentioned above. The seL4webserver projects runs a webserver inside a Linux VM inside a CAmkES system, and thus were similarly excluded. The seL4 tutorials manifest mainly includes project-generator code for setting up specific prescribed seL4 mechanism tutorials, which are all highly tailored to specific simple examples for the purpose of the tutorial at hand, so they were also excluded.

The rumprun app focuses specifically on supporting components of the NetBSD kernel via the 'rump kernel' approach, to allow for running unmodified NetBSD

drivers atop seL4 [65]. This project does not appear to have much active support or use, and is overly complex for the task at hand, so was similarly excluded.

The verification manifest was also excluded as it exists primarily to support working on the seL4 proofs with Isabelle/HOL.

This left the seL4test and seL4bench manifests as the main two projects to consider[2]. They ultimately served as good reference points for starting a new blank project, as whilst they each have a specific focus on bootstrapping environments for testing and benchmarking specifically, they both make use of similar support code for establishing their base root tasks and running environments. This includes for example:

- the `sel4runtime` library, which makes a basic C environment available and manages access to the `seL4_BootInfo` struct that the kernel passes to the root task on startup (as shown back in Figure 1.7), and
- the `libsel4platsupport` and `libplatsupport` libraries, which can be used to provide basic I/O facilities such as setting up access to the system serial output device, and connecting C's `printf()` to it to allow program output to be seen when running the built OS project in the QEMU simulator.

There are a few other key support libraries that both projects make use of, which were essential to reducing the amount of work required to perform straightforward memory and resource management on top seL4, including:

- the `allocman` untyped memory allocation manager
- the `vka` (Virtual Kernel Allocator) kernel object allocator library / interface
- the `vspace` library, to support virtual address space management

`allocman` provides careful automatic management of the recursive set of dependencies involved in the problem of allocating and de-allocating different seL4 kernel object resources from the base 'Untyped' memory capabilities / object type. Untypeds must be split down into subsequent smaller Untyped capabilities (which can be done by issuing the retype operation on an Untyped with target type of Untyped), before finally being retyped into the desired final object. All the retyping operation results must be stored in unused CSpace slots, of which more

[2]The 'sos' 'simple operating system' base project used for the UNSW Advanced Operating System subject was also consulted for reference: `https://github.com/SEL4PROJ/AOS`

may need to be allocated beforehand. This is further complicated by the matter that more CSpace slots have to be obtained by allocating a CNode object, which itself must come from an Untyped object. All of these operations also require data bookkeeping, which must be stored somewhere in physical memory - space for which the library must also manage.

These kinds of allocation problems were ones that did not need to be solved in any special way for the Pony implementation, so `allocman` was taken as an off-the-shelf solution. A starter `allocman` instance can be bootstrapped from the untypeds described in the bootinfo, by using the `bootstrap_use_current_simple` function, and passing it a static buffer to start using for bookkeeping data, which can simply be declared as a static array in the rootserver program. (The kernel will ensure this will be backed by mapped virtual memory as part of its own initialization process that sets up the root task.)

Once an `allocman` instance is set up, it can be used through the `vka` interface which allows for making arbitrary kernel object allocation requests, such as asking for a new thread, endpoint, or pagetable object.

This all finally leads up to the `vspace` library, which provides higher-level facilities for managing a virtual address space, that is ultimately controlled by VSpace capabilities under the hood. `libsel4utils` provides the `sel4utils_bootstrap_vspace_with_bootinfo` function, which takes a `vka` instance and the bootinfo struct, and sets up a `vspace` instance pre-configured with the current state of the virtual address space, as per the details of the bootinfo struct.

With a vspace instance set up, all the pieces are in place to support filling more of the root task's virtual address space out with mapped physical memory. This includes the ability to map physical memory addresses that are actually control registers for devices. Consequently, at this point the the `platsupport_serial_setup_simple` function can be invoked, which will find the physical memory addresses associated with the system serial device for the platform the kernel has been built for, map them in to the current address space, and then update some static function pointers such that subsequent calls to `printf` will use the serial device for character output.

Once a base project and root task with all these libraries was set up and configured, all the tools and functionalities required for the port were in place and the porting work itself could begin.

8.3 Step 2: porting the allocator

With a base seL4 enviornment established, the task of porting the Pony pool memory allocator over came next, as it had been identified as the component that made sense to try and port first as discussed in section 8.1.

Analysis of the allocator code and the descriptive writeup of it in [ClebschThesis, §A.3] revealed many layers of places where allocated memory can ultimately come from when a runtime call for allocation is made. A breakdown of these layers is shown in Figure 8.2.

A key dependency noted at this point was that several core datastructures within the allocator are expected to be local per scheduler thread, and that this is currently implemented by the variables in question being declared as C thread-local-storage variables. Thread-local-storage is quite a complicated feature [24] that relies on co-operation between both the C compiler, and the operating system's process loading procedures and dynamic linker, via the ELF format that compiled code is ultimately stored within. If a new thread is started in the same address space, a spot in that address space must be reserved / allocated for that new thread's thread-local storage, the "image" for the TLS copied in from the program ELF, and the thread must be configred to use this new storage area. The `sel4runtime` library provides support for some of these operations, and ensures that the initial thread in the root task will have correctly-functioning thread-local variables, but for additional threads, library functions must be invoked manually as part of whatever threading model is established as part of the particular seL4 system being built. [3]

Key Observation 4 *`libponyrt` relies on C thread-local storage for many of its key datastructures, which does not necessarily come for free on seL4, and must be carefully considered and managed depending on the process-loading, threading, and address space models designed for the target seL4 system being put together.*

[3] An example of how thread-local storage can be managed using `sel4runtime`'s functions can be found within the `benchmark_spawn_process` function in `https://github.com/seL4/sel4bench/blob/12.1.x-compatible/libsel4benchsupport/src/support.c#L254` (line #254 at time of writing)

1. A *thread-local* list of **previously-freed size-classed allocations** is first consulted.

   This will be populated throughout a program's execution whenever memory is freed by the runtime.

   If nothing can be found here...

2. A *global* list of **previously-freed size-classed allocations** is consulted.

   This list gets populated whenever a certain threshold of thread-local allocation-freeing accumulates in one thread. Operations on the list are made using atomic lock-free operations (e.g. compare-and-swap loop for list push).

   If this list is empty or has nothing large enough on it...

3. (a) For sub-1kB allocations, a *thread-local* **(1kB) block of contiguous memory** is consulted

      If the allocation fits within the space available on this block, that space is deducted from the block and used for the allocation. Otherwise (or if there was no block present), a new block is fetched by making a recursive call into the pool allocator for 1kB, and space on this block is used instead.

   (b) For >1kB allocations, a thread-local *list* of **previously-freed or unused larger blocks of memory** is consulted.

      If there is room on any of these, that space is taken and used for the allocation.

   Finally, if none of the above options were able to supply space for the requested allocation...

4. A large new 'block' is created by **asking the underlying operating system for more pages of virtual address space**.

   (a) In cases of very large allocations (>128MB, a tuneable parameter), the whole set of returned pages is used for the allocation.

   (b) Otherwise, 128MB (or equivalent tuneable size parameter) of space is asked for, the allocation is deducted from this, and the remaining space is added into the **thread-local *list*** of free blocks mentioned in item 3b.

Figure 8.2: Description of the various levels of memory space sources used in the Pony pool allocator
Adapted from code in `mem/pool.c` in the `libponyrt` folder of the `ponyc` repo [62]

Another dependency revealed at this point is the Pony codebase's use of C11 atomics, such as the `atomic_store_explicit`, `atomic_load_explicit`, and `atomic_exchange_explicit` functions, which underly all of the thread-safe lock-free datastructure operations used by the runtime.

The `libponyrt` code ultimately relies on the `<stdatomic.h>` file for these atomics. This file comes shipped as part of a compiler itself[4]. However all seL4 project invoke GCC with the `-nostdinc` flag[5], which excludes all system-level header files on the compilation host system from the set of paths available for including header files from. Explicitly re-including the relevant directory did not seem to work[6], so symlinking the file into the project was done as a workaround.

[4] `stdatomic.h` was found in directory `/usr/lib/gcc/x86_64-linux-gnu/10/include/stdatomic.h` within the seL4 docker build environment

[5] The root source of this flag is the `add_default_compilation_options` CMake macro in `helpers/environment_flags.cmake` from the `cmake-tool` seL4 repo. This helper gets called by `musllibc_set_environment_flags`, which is called by `musllibc_setup_build_environment_with_sel4runtime`, the top-level command used to bring the seL4-friendly musl libc library into build scope for compiling C code for seL4 environments.

[6] This appears to be an open issue for CMake projects: `https://gitlab.kitware.com/cmake/cmake/-/issues/19227`

The final ‘dependency’ to take care of was the allocator’s requesting of more virual memory.

Ultimately, when the allocator code is first invoked at runtime, all of the levels of memory space sources described in Figure 8.2 are empty, as no existing memory has been freed from prior allocation. This means that the all the levels of checks will cascade down into an initial ‘call to the underlying operating system for more pages of memory’. Inside `libponyrt`, this is implemented within the function `ponyint_virt_alloc`. The existing version of this function is shown below in Figure 8.3.

```c
#ifdef PLATFORM_IS_POSIX_BASED
#include <sys/mman.h>
#endif

#if defined(PLATFORM_IS_MACOSX)
#include <mach/vm_statistics.h>
#endif

void* ponyint_virt_alloc(size_t bytes)
{
  void* p;
  bool ok = true;

#if defined(PLATFORM_IS_WINDOWS)
  p = VirtualAlloc(NULL, bytes, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
  if(p == NULL)
    ok = false;
#elif defined(PLATFORM_IS_POSIX_BASED)
#if defined(PLATFORM_IS_LINUX)
  p = mmap(0, bytes, PROT_READ | PROT_WRITE,
    MAP_PRIVATE | MAP_ANONYMOUS | MAP_NORESERVE, -1, 0);
#elif defined(PLATFORM_IS_MACOSX)
  p = mmap(0, bytes, PROT_READ | PROT_WRITE,
    MAP_PRIVATE | MAP_ANON, -1, 0);
#elif defined(PLATFORM_IS_DRAGONFLY)
  p = mmap(0, bytes, PROT_READ | PROT_WRITE,
    MAP_PRIVATE | MAP_ANON, -1, 0);
#elif defined(PLATFORM_IS_OPENBSD)
  p = mmap(0, bytes, PROT_READ | PROT_WRITE,
    MAP_PRIVATE | MAP_ANON, -1, 0);
#elif defined(PLATFORM_IS_BSD)
#ifndef MAP_ALIGNED_SUPER
#define MAP_ALIGNED_SUPER 0
#endif
  p = mmap(0, bytes, PROT_READ | PROT_WRITE,
    MAP_PRIVATE | MAP_ANON | MAP_ALIGNED_SUPER, -1, 0);
#endif
  if(p == MAP_FAILED)
    ok = false;
#endif

  if(!ok)
  {
    perror("out of memory: ");
    abort();
  }

  return p;
}
```

Figure 8.3: Existing `ponyint_virt_alloc` implementation from the release Pony runtime
Taken from `libponyrt` in the `ponyc` repo [62]

However, obviously this will not work on seL4, as it does not provide `mmap` as a syscall (see section 1.2). With the `vspace` interface available from the root task environment established in section 8.2, the `ponyint_virt_alloc` function was thus altered to the one shown in Figure 8.4.

```c
#include <vspace/vspace.h>
#include <utils/page.h>

/*
  seL4 allocation code based on dynamic morecore in
  libsel4muslcsys/src/sys_morecore.c

  Globals need to be defined somewhere (probably in the
  main function of your app. And setup something like

  sel4pony_this_vspace = &vspace;
*/
vspace_t *sel4pony_this_vspace = NULL;

void* ponyint_virt_alloc(size_t bytes)
{
  void* p;
  bool ok = true;

  #ifdef PONYSEL4ALLOC_USE_LARGE_PAGES
  uint32_t pages = BYTES_TO_SIZE_BITS_PAGES(bytes, seL4_LargePageBits);
  size_t paged_bytes = ((size_t)pages) * BIT(seL4_LargePageBits);
  fprintf(stderr, "Pulling %d new large pages into vspace (0x%xB from 0x%xB
↪  request)\n", pages, paged_bytes, bytes);
  if (paged_bytes != bytes) {
    fprintf(stderr, "alloc spillover of %d bytes\n", (paged_bytes - bytes));
  }
  p = vspace_new_pages(sel4pony_this_vspace, seL4_AllRights, pages,
↪  seL4_LargePageBits);
  #else
  uint32_t pages = BYTES_TO_4K_PAGES(bytes);
  fprintf(stderr, "Pulling %d new pages (%d) into vspace\n", pages, bytes);
  p = vspace_new_pages(sel4pony_this_vspace, seL4_AllRights, pages, seL4_PageBits);
  #endif
  if(p == NULL)
    ok = false;

  if(!ok)
  {
    fprintf(stderr, "OS alloc failed (vspace_new_pages returned NULL)\n");
    abort();
  }

  return p;
}
```

Figure 8.4: `ponyint_virt_alloc` implementation developed for the seL4 environment

At this point, the allocator code successfully compiled and could be tested within the `main()` function of the root task (after all the environment setup described in section 8.2). Initially, when testing with `ponyint_virt_alloc` configured to use default 4kb pages on a 512mb virtual machine, and to ask the `vspace` library for 2MB at a time (512 4kb pages), the allocator would fail when hitting the third call to `vspace_new_pages`. This is probably due to the large amount of capability and bookkeeping overhead for allocating so many pages at once. However, switching to use x86 large pages for the `vspace` requests fixed this and allowed for around 500mb of memory to be allocated before receiving the same out-of-space error.

8.4 Step 3: porting the SPMC messages queues

With a functioning pool allocator, Pony message allocation now worked too, so the next step was to port over the SPMC (single-producer multiple-consumer) message queues used for the core Pony actor-to-actor message passing functionality.

With C11 atomics already out of the way from porting the pool allocator, this next step was relatively painless, and the `ponyint_actor_messageq_push` and `ponyint_actor_messageq_pop` funtions were successfully ported and tested within the seL4 environment.

At this stage, all of the minimum components required for some sort of functional actor runtime except the work-stealing scheduling were in place. A simple test C program, similar in style to the ping-pong 'mailbox' Pony program of Listing 1, was constructed to test all the moving parts, which is shown below in Listing 3. Note that due to the lack of work-stealing, the actor message queues must be explicitly handled multiple times, to ensure that all messages produced by prior runs are handled.

```
typedef struct pony_ctx_t pony_ctx_t;
typedef struct pony_simpleactor_t pony_simpleactor_t;

/** Dispatch function.
 *
 * Each actor has a dispatch function that is invoked when the actor handles
 * a message.
 */
typedef void (*pony_dispatch_fn)(pony_ctx_t* ctx, pony_simpleactor_t* actor,
  pony_msg_t* m);

typedef struct pony_faketype_t {
  uint32_t id;
  char* name;
  pony_dispatch_fn dispatch;
} pony_faketype_t;

struct pony_simpleactor_t {
  pony_faketype_t* type;
  messageq_t q;
  // PONY_ATOMIC(uint8_t) flags;

  // keep things accessed by other actors on a separate cache line
  // alignas(64) heap_t heap; // 52/104 bytes
  // gc_t gc; // 48/88 bytes
};
struct pony_ctx_t {};

enum MainMethods {
    M_helloworlddebug,
    M_main,
    M_pong
};

typedef struct main__main__msg_t {
    pony_msg_t msghdr;
    // TODO-ACTORALLOC: this actor arg should be removed and the Ponger actor
    // should be allocated by the main actor itself
    pony_simpleactor_t* pongactor;
    uint32_t pingc;
} main__main__msg_t;
typedef struct main__pong__msg_t {
    pony_msg_t msghdr;
} main__pong__msg_t;

enum PongerMethods {
    P_helloworlddebug,
    P_ping
};

typedef struct ponger__ping__msg_t {
    pony_msg_t msghdr;
    pony_simpleactor_t* pongdest;
    uint32_t pongc;
} ponger__ping__msg_t;
```

```
void Main_dispatch(pony_ctx_t* ctx, pony_simpleactor_t* a, pony_msg_t* m) {
    switch(m->id) {
        case M_helloworlddebug:
            fprintf(stderr, "Actor<Main> @ %x says Hello World!!\n", a);
            break;
        case M_main:
            fprintf(stderr, "Actor<Main>.main() @ %x \n", a);
            // allocate / obtain ref to ponger
            pony_simpleactor_t* ponger_ref = ((main__main__msg_t*)m)->pongactor;
            // send ping messages
            for(int i = 0; i < 5; i++) {
                fprintf(stderr, "Actor<Main>.main() @ %x - pushing ping msg #%d\n",
↪  a, i);
                ponger__ping__msg_t* m =
↪  (ponger__ping__msg_t*)pony_alloc_msg(POOL_INDEX(sizeof(ponger__ping__msg_t)),
↪  P_ping);
                m->pongc = 2;
                m->pongdest = a;
                ponyint_actor_messageq_push_single(
                    &ponger_ref->q,
                    (pony_msg_t*)m, (pony_msg_t*)m
                );
            }
            break;
        case M_pong:
            fprintf(stderr, "Actor<Main>.pong() @ %x - pong!\n", a);
            break;
        default:
            fprintf(stderr, "Main_dispatch @ %x: unknown message type %d\n", a,
↪  m->id);
            break;
    }
}
void Ponger_dispatch(pony_ctx_t* ctx, pony_simpleactor_t* a, pony_msg_t* m) {
    switch(m->id) {
        case P_helloworlddebug:
            fprintf(stderr, "Actor<Ponger> @ %x says Hello World!!\n", a);
            break;
        case P_ping:
            fprintf(stderr, "Actor<Ponger>.ping() @ %x\n", a);
            // Grab ponger ref from msg
            pony_simpleactor_t* pongdest_ref = ((ponger__ping__msg_t*)m)->pongdest;
            for (int i = 0; i < ((ponger__ping__msg_t*)m)->pongc; i++) {
                fprintf(stderr, "Actor<Ponger>.ping() @ %x - pushing pong msg
↪  #%d\n", a, i);
                main__pong__msg_t* m =
↪  (main__pong__msg_t*)pony_alloc_msg(POOL_INDEX(sizeof(main__pong__msg_t)),
↪  M_pong);
                ponyint_actor_messageq_push_single(
                    &pongdest_ref->q,
                    (pony_msg_t*)m, (pony_msg_t*)m
                );
            }
            break;
        default:
            fprintf(stderr, "Ponger_dispatch @ %x: unknown message type %d\n", a,
↪  m->id);
            break;
    }
}
```

```
static pony_faketype_t mainactor_t = {
    .id=1,
    .name="Main",
    .dispatch=Main_dispatch
};
static pony_faketype_t pongeractor_t = {
    .id=1,
    .name="Ponger",
    .dispatch=Ponger_dispatch
};
void handle_msgq(pony_simpleactor_t* actor) {
    size_t batch = 100/*PONY_SCHED_BATCH*/;

    pony_msg_t* msg;
    size_t app = 0;

    // If we have been scheduled, the head will not be marked as empty.
    pony_msg_t* head = atomic_load_explicit(&actor->q.head, memory_order_relaxed);

    while((msg = ponyint_actor_messageq_pop(&actor->q)) != NULL) {
        printf("got msg - id: %d\n", msg->id);
        actor->type->dispatch(NULL, actor, msg);
        app++;
        // Stop handling a batch if we reach the head we found when we were
        // scheduled.
        if(msg == head)
            break;
    }
}

void test_sel4pony_simpleactor(uint32_t pingc) {
    pony_simpleactor_t actor_main = {
        .type = &mainactor_t
    };
    pony_simpleactor_t actor_ponger = {
        .type = &pongeractor_t
    };
    ponyint_messageq_init(&actor_main.q);
    ponyint_messageq_init(&actor_ponger.q);

    main__main__msg_t* main__main__msg = (main__main__msg_t*)pony_alloc_msg(0,
↪   M_main);
    // TODO-ACTORALLOC: this actor should be allocated by the main actor itself
    // For now, the pong actor is a static alloc passed as an argument to main()
    main__main__msg->pongactor = &actor_ponger;
    main__main__msg->pingc = pingc;
    ponyint_actor_messageq_push(
        &actor_main.q,
        (pony_msg_t*)main__main__msg,
        (pony_msg_t*)main__main__msg
    );
    // and now.... dispatch! dispatch?
    handle_msgq(&actor_main);
    // TODO: work-stealing
    handle_msgq(&actor_ponger);
    handle_msgq(&actor_main);
}
```

Listing 3: 'simpleactor' test C program for demonstrating the functional Pony allocator and message queues.

8.5 Future steps: scheduling, actor heaps, garbage collection

For future work, the next steps to investigate porting would be the work-stealing scheduler, with its MPSC (multiple-producer single-consumer) work queues, and some form of runtime threads, each of which would require appropriate thread-local storage. As per the analysis of the `main()` procedure in section 8.1, the threads will need to be able to be 'joined', which is a POSIX thread functinoality that seL4 threads do not provide on their own, so some other sort of synchronisation method (likely via a Notification object) would be required.

Whilst real Pony actor allocation should be mostly achievable with the current port, due to the main dependency for their allocation being simply the pool allocator, actors themselves have their own heaps, which there is assorted further code that needs to be ported for the support for allocating objects from them.

Additionally, there are many components of the garbage-collection subsystem that would need to be ported as well to support real Pony programs, as even the program initialisation shown in Figure 8.1 makes use of it.

Chapter 9

Conclusions

Pony is a complex language, built very strongly around its actor execution model, and atop many layers of assumptions - causal message passing, single address spaces, and POSIX-compatible runtime threads. This has ultimately made it somewhat restrictive to design mappings for seL4 with.

The seL4 primitives that its capabilities are used for are also very minimalist and low-level, and the majority of them either control or represent concepts well below the level of what a memory-safe programming language usually reasons about, or come with particular invocation concerns (i.e. blocking implications) surrounding the seL4 threading and IPC model, which is quite specific to seL4 / L4 microkernels. Examples of the low-level concepts include physical memory (not virtual memory), and the management / fine alteration of virtual address spaces, something usually taken care of by the operating system, below the level that the programming language is targeting. On the threading and IPC front, seL4 threads are somewhat particular, as opposed to what threads are more generally understood to represent on most operating systems (i.e. POSIX threads). This makes design options surrounding their language-level representation come with many strings attached - such as the inability for non-blocking send to be used reliably on seL4 Endpoints (as per Key Observation 2).

This ultimately made finding any natural mapping of Pony onto seL4's capabilities difficult, due to Pony's general design reliance on assuming non-blocking behaviour, and a 'balanced' thread environment where work in the language is handled and performed in the same way regardless of which core code is being executed on. This is somewhat at odds with seL4, where the core you are on can

matter more[1].

seL4 is also more generally not yet really built for large multi-core systems[2] where the concurrency and parallelism in Pony can really be exploited. In regards to multicore systems, the seL4 website currently states[3] that:

> The multicore kernel uses a big-lock approach, which makes sense for tightly-coupled cores that share an L2 cache [50]. It is not meant to scale to many cores, where instead multikernel is the right approach (running separate kernel image on each cluster of cores sharing a cache). This "clustered multikernel" configuration is presently not supported, though..

This "clustered multikernel" approach has been covered in [68], but there are complicated problems posed for maintaining the verification proofs that underly seL4's world-leading security assurances. The Barrelfish operating system, which pioneered the multikernel model [9], and which, as mentioned, is uses a capability system that is in fact based off of seL4, could be worth investigating a mapping of Pony for too. There are complicated distributed capability problems involved in the multikernel model, which the Barrelfish project has investigated [57, Chapter 5], but have been outside the scope of any analysis of a language mapping in this thesis.

Despite being an object-capability language that can achieve strong logical separation between units of code built within a single project, Pony also lacks support for features found in other ocap languages, such as dynamic code execution / evaluation, and any of the more explicitly confined execution contexts provided by the 'vat' models that other distributed ocap programming environments espouse. More generally, whilst it follows object-capability discipline, and is a language that can provide high levels of performant concurrency, it does not come with any remote or explicit distributed object model such as the NearRefs and FarRefs of E. Even the proposed distributed model of Pony [11] addressed in section 5.8 again assumes an automatically-balanced exection model, where the

[1] It is worth noting that this thesis was conducted on the 'mainline' seL4, and not the newer MCS 'mixed-criticality systems' version, which introduces the concept of 'passive threads' that are not pinned to any one particular CPU core. This could be worth further investigation - however the MCS model also comes with time scheduling budgets that could conflict with Pony's balanced thread execution model (and in fact was not considered initially in this project for these reasons)

[2] The thesis that adopted seL4's capability model into Barrelfish [49] even refers to it as 'a single-core capability system'

[3] See `https://docs.sel4.systems/projects/sel4/frequently-asked-questions.html#does-sel4-support-multicore`

work-stealing algorithm can cause actors to be migrated at any point in time, and thus the model through which local and remote communication is presented is aimed to be kept uniform. Something with more explicit management of remote objects in confined compartments might map better onto seL4's model of confined protection domains.

Regardless, if a full port of the Pony runtime to a singular seL4 protection domain can be achieved, there could potentially still be interesting future work to investigate for working with higher-level capabilities set up by other existing seL4 project infrastructure. Currently, it is not possible to start a whole Pony program with a restricted subset of capabilities - the Main actor will always get passed the `AmbientAuth` capability, which can be downtyped into any of the various authorities the language has been designed around (as highlighted in section 5.5). However, it would not be too difficult to alter this such that more granular, specific subsets of capability types could be provided as part of some API via the root `env` argument, given that the `env` struct is already set up by generated machine code from LLVM that Pony source code cannot see or alter. On an seL4 CAmkES system, these could be set up to represent access to various CAmkES component connectors, backed by capability addresses in the `env` struct that the program could be explicitly set up with.

In regards to other language options, there is a distinct lack of 'systems' ocap languages in the options found from the survey that have the high performance most likely to be saught for the design of systems built with seL4. C++ is a common choice for high performance programming, but is notoriously memory-unsafe - however an ocap 'discpline' could perhaps be teased out of a subset of it, leveraging its private constructors and model for delegated 'friendship' with the `friend` declaration. Further research on something within Rust could be useful future work worth pursuing, as it is likely to be able to give the high performance guarantees required, whilst also still being memory-safe when used without `unsafe` blocks.

Additionally, the issue of *ambient authority for memory allocation* as raised in Key Observation 3 from section 6.7 is an interesting result of the research. This could be worth re-examining within the context of object-capability languages, especially in embedded development contexts where smaller amounts of memory lead to memory management being a more important issue.

Bibliography

[1] Agoric Inc. endojs/endo: Endo is a distributed secure JavaScript sandbox, based on SES. https://github.com/endojs/endo, . Github Repository.

[2] Agoric Inc. Jessica - Jessie (secure distributed Javascript) Compiler Architecture. https://github.com/agoric-labs/jessica/, . Github Repository.

[3] Agoric Inc. Jessie, simple universal safe mobile code. https://github.com/endojs/Jessie, . Github Repository.

[4] B. Anderson, L. Bergstrom, M. Goregaokar, J. Matthews, K. McAllister, J. Moffitt, and S. Sapin. Engineering the Servo Web Browser Engine Using Rust. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, page 81–89, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342056. doi: 10.1145/2889160.2889229. URL https://doi.org/10.1145/2889160.2889229.

[5] Anthony Pesch. inolen/quakejs. https://github.com/inolen/quakejs, 2014. GitHub Repository, live demo available at http://www.quakejs.com/.

[6] Auxon Corporation. selfe-sys - A generated thin wrapper around libsel4.a, with supporting subcrates. https://github.com/auxoncorp/selfe-sys, 2016. GitHub repository.

[7] Auxon Corporation. ferros - A Rust library to add extra assurances to seL4 development. https://github.com/auxoncorp/ferros, 2019. GitHub repository.

[8] A. Baumann. Inter-dispatcher communication in Barrelfish. Technical Note 011, ETH Zurich, December 2011. URL https://barrelfish.org/publications/TN-011-IDC.pdf.

[9] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 29–44, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605587523. doi: 10.1145/1629575.1629579. URL https://doi.org/10.1145/1629575.1629579.

[10] S. Biggs, D. Lee, and G. Heiser. The jury is in: Monolithic os design is flawed: Microkernel-based designs improve security. In *Proceedings of the 9th Asia-Pacific Workshop on Systems*, APSys '18, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450360067. doi: 10.1145/3265723.3265733. URL https://doi.org/10.1145/3265723.3265733.

[11] S. Blessing. A String of Ponies: Transparent Distributed Programming with Actors. Available online at https://www.ponylang.io/media/papers/a_string_of_ponies.pdf or https://www.doc.ic.ac.uk/~scb12/publications/s.blessing.pdf, 2013. Masters Thesis.

[12] A. Boyton, J. Andronick, C. Bannister, M. Fernandez, X. Gao, D. Greenaway, G. Klein, C. Lewis, and T. Sewell. Formally Verified System Initialisation. In Lindsay Groves, Jing Sun, editor, *Proceedings of the 15th International Conference on Formal Engineering Methods*, pages 70–85, Queenstown, New Zealand, Oct. 2013. Springer. doi: 10.1007/978-3-642-41202-8_6.

[13] G. Bracha, P. Ahe, V. Bykov, R. Macnak, E. Miranda, and B. Maddox. The Newspeak Programming Language. https://newspeaklanguage.org/. Website.

[14] Bytecode Alliance. cap-std - Capability-based version of the Rust standard library . https://github.com/bytecodealliance/cap-std, 2020. GitHub repository.

[15] Bytecode Alliance. WASI (Embedding in Rust example) - Wasmtime. https://docs.wasmtime.dev/examples-rust-wasi.html, 2020. Example page from Wasmtime documentation, Accessed: 2022-11-08, Last update: 2021-06-04 (see documentation source at https://github.com/bytecodealliance/wasmtime/blob/main/docs/examples-rust-wasi.md).

[16] D. Charousset, R. Hiesgen, and T. C. Schmidt. Revisiting Actor Programming in C++. *Computer Languages, Systems & Structures*, 45(C):105–131, apr 2016. ISSN 1477-8424. doi: 10.1016/j.cl.2016.01.002. URL https://doi.org/10.1016/j.cl.2016.01.002.

[17] S. Clebsch. *'Pony': co-designing a type system and a runtime.* PhD thesis, Imperial College London, 2017. Available online at https://spiral.imperial.ac.uk/handle/10044/1/65656.

[18] S. Clebsch and S. Drossopoulou. Fully Concurrent Garbage Collection of Actors on Many-Core Machines. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, page 553–570, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450323741. doi: 10.1145/2509136.2509557. URL https://doi.org/10.1145/2509136.2509557.

[19] S. Clebsch, S. Drossopoulou, S. Blessing, and A. McNeil. Deny Capabilities for Safe, Fast Actors. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE! 2015, page 1–12, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450339018. doi: 10.1145/2824815.2824816. URL https://doi.org/10.1145/2824815.2824816.

[20] S. Clebsch, J. Franco, S. Drossopoulou, A. M. Yang, T. Wrigstad, and J. Vitek. Orca: GC and Type System Co-Design for Actor Languages. *Proc. ACM Program. Lang.*, 1(OOPSLA), oct 2017. doi: 10.1145/3133896. URL https://doi.org/10.1145/3133896.

[21] P.-E. Dagand, A. Baumann, and T. Roscoe. Filet-o-Fish: Practical and Dependable Domain-Specific Languages for OS Development. In *Proceedings of the Fifth Workshop on Programming Languages and Operating Systems*, PLOS '09, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605588445. doi: 10.1145/1745438.1745446. URL https://doi.org/10.1145/1745438.1745446.

[22] J. B. Dennis and E. C. Van Horn. Programming semantics for multiprogrammed computations. *Commun. ACM*, 9(3):143–155, mar 1966. ISSN 0001-0782. doi: 10.1145/365230.365252. URL https://doi.org/10.1145/365230.365252.

[23] R. Developers. Robigalia. https://rbg.systems/. Website, Last updated: December 17 2021, accessed: November 26 2022.

[24] U. Drepper. ELF Handling For Thread-Local Storage. Technical report, August 2013. URL http://people.redhat.com/drepper/tls.pdf. Version 0.21. Also available at https://www.akkadia.org/drepper/tls.pdf.

[25] M. Fahndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. Larus, and S. Levi. Language Support for Fast and Reliable Message-based Communication in Singularity OS. In *Proceedings of the EuroSys 2006 Conference*, pages 177–190. Association for Computing Machinery, Inc., April 2006. URL https://www.microsoft.com/en-us/research/publication/language-support-for-fast-and-reliable-message-based-communication-in-singularit

[26] M. Fernandez, G. Klein, I. Kuz, and T. Murray. CAmkES Formalisation of a Component Platform. Technical report, NICTA and UNSW, Australia, Nov. 2013.

[27] K. Fernandez-Reyes, I. O. Gariano, J. Noble, E. Greenwood-Thessman, M. Homer, and T. Wrigstad. Dala: a simple capability-based dynamic language design for data race-freedom. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 1–17, 2021.

[28] N. Feske. Introducing Genode. https://genode-labs.com/publications/nfeske-genode-fosdem-2012-02.pdf. Slides, FOSDEM 2012, Brussels, February 2012.

[29] M. Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Design Inc., 2010. https://racket-lang.org/tr1/.

[30] Genode Labs. Genode Operating System Framework - About Genode. https://genode.org/about/index. Website Page, accessed: November 27 2022.

[31] Gernot Heiser. How to (and how not to) use seL4 IPC. https://microkerneldude.org/2019/03/07/how-to-and-how-not-to-use-sel4-ipc/, March 2019. Blog Post.

[32] Google Inc. Protocol Buffers - Google Developer Documentation. https://developers.google.com/protocol-buffers/. Documentation Website, accessed: November 20 2022.

[33] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien. Bringing the Web up to Speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 185–200, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349888. doi: 10.1145/3062341.3062363. URL https://doi.org/10.1145/3062341.3062363.

[34] N. Hardy. The confused deputy: (or why capabilities might have been invented). *SIGOPS Oper. Syst. Rev.*, 22(4):36–38, oct 1988. ISSN 0163-5980. doi: 10.1145/54289.871709. URL https://doi.org/10.1145/54289.871709.

[35] G. Heiser, L. Parker, P. Chubb, I. Velickovic, and B. Leslie. Can we put the "S" into IoT? In *IEEE World Forum on Internet of Things*, Yokohama, JP, Nov. 2022.

[36] G. Hunt and J. Larus. Singularity: Rethinking the Software Stack. *ACM SIGOPS Operating Systems Review*, 41(2):37–49, April 2007. URL https://www.microsoft.com/en-us/research/publication/singularity-rethinking-the-software-stack/.

[37] G. Hunt, J. Larus, M. Abadi, M. Aiken, P. Barham, M. Fahndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. Zill. An Overview of the Singularity Project. Technical Report MSR-TR-2005-135, Microsoft Research, October 2005. URL https://www.microsoft.com/en-us/research/publication/an-overview-of-the-singularity-project/.

[38] Kevin Reid. kpreid/e-on-cl: E language implementation targeting Common Lisp. https://github.com/kpreid/e-on-cl. Github Repository.

[39] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser. Comprehensive formal verification of an OS microkernel. *ACM Trans. Comput. Syst.*, 32(1), feb 2014. ISSN 0734-2071. doi: 10.1145/2560537. URL https://doi.org/10.1145/2560537.

[40] I. Kuz, G. Klein, C. Lewis, and A. C. Walker. capDL: A Language for Describing Capability-Based Systems. In *Asia-Pacific Workshop on Systems (APSys)*, pages 31–35, New Delhi, India, Aug. 2010.

[41] S. Marr. smarr/SOMns: SOMns: A Newspeak for Concurrency Research. `https://github.com/smarr/SOMns`. GitHub repository.

[42] N. D. Matsakis and F. S. Klock. The Rust Language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, page 103–104, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450332170. doi: 10.1145/2663171.2663188. URL `https://doi.org/10.1145/2663171.2663188`.

[43] M. S. Miller. The E language. `http://erights.org/elang/`, . Website.

[44] M. S. Miller. Welcome to ERights.Org. `http://erights.org/index.html`, . Website.

[45] M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD Thesis, Johns Hopkins University, May 2006.

[46] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized JavaScript. `https://www.researchgate.net/publication/248520657_Caja_Safe_active_content_in_sanitized_JavaScript`, June 2008. Technical Report, Google, Inc.

[47] M. S. Miller, T. V. Cutsem, and B. Tulloh. Distributed Electronic Rights in JavaScript. In *ESOP'13 22nd European Symposium on Programming*, 2013. URL `https://research.google/pubs/pub40673/`.

[48] S. Moore, C. Dimoulas, D. King, and S. Chong. SHILL: A secure shell scripting language. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 183–199, Broomfield, CO, Oct. 2014. USENIX Association. ISBN 978-1-931971-16-4. URL `https://www.usenix.org/conference/osdi14/technical-sessions/presentation/moore`.

[49] M. Nevill. An evaluation of capabilities for a multikernel. Master's thesis, ETH Zurich, 2012. URL `https://barrelfish.org/publications/nevill-master-capabilities.pdf`. Systems Group Masters Thesis Nr. 46.

[50] S. Peters, A. Danis, K. Elphinstone, and G. Heiser. For a microkernel, a big lock is fine. In *Asia-Pacific Workshop on Systems (APSys)*, Tokyo, JP, July 2015. ACM.

[51] Rainer Hahnekamp. JavaScript essentials: why you should know how the engine works. https://www.freecodecamp.org/news/javascript-essentials-why-you-should-know-how-the-engine-works-c2cc0d321553/. Website Article.

[52] Y. Rezgui. Permissionless is the future of Storage on Android. https://medium.com/androiddevelopers/permissionless-is-the-future-of-storage-on-android-3fbceeb3d70a, October 2022. Medium Article from Android Developers, "The official Android Developers publication on Medium".

[53] M. Seaborn. Plash: the Principle of Least Authority shell. https://www.cs.jhu.edu/~seaborn/plash/plash-orig.html. Website.

[54] Second State. Containerization on the edge. https://www.secondstate.io/articles/wasmedge-sel4/, 2021. Website Article, Accessed: 2022-11-08.

[55] Second State. second-state/wasmedge-seL4: Integrate WasmEdge with seL4. https://github.com/second-state/wasmedge-seL4, 2021. GitHub Repository, Accessed: 2022-11-08.

[56] Simple Object Machine. SOM: Simple Object Machine. https://som-st.github.io/. Website.

[57] A. Singhania, I. Kuz, M. Nevill, and S. Gerber. Capability Management in Barrelfish. Technical Note 013, ETH Zurich, June 2017. URL https://barrelfish.org/publications/TN-013-CapabilityManagement.pdf.

[58] J. Tate-Gans and S. Leffler. Announcing KataOS and Sparrow. https://opensource.googleblog.com/2022/10/announcing-kataos-and-sparrow.html, October 2022. Blog Post, Google Open Source Blog.

[59] TC39 Proposal. tc39/proposal-compartments: Compartmentalization of host behavior hooks for JS. https://github.com/tc39/

proposal-compartments. Github Repository, various authors including Mark Miller.

[60] corbet. Two new system calls: splice() and sync_file_range(). https://lwn.net/Articles/178199/, April 2006. Article.

[61] The Pony Developers. Object Capabilities - Pony Tutorial. https://tutorial.ponylang.io/object-capabilities/object-capabilities.html, . Official Pony Language Tutorial website.

[62] The Pony Developers. ponylang/ponyc: Pony is an open-source, actor-model, capabilities-secure, high performance programming language. https://github.com/ponylang/ponyc, . GitHub Repository.

[63] The Qt Company. Qt for WebAssembly | Qt 5.15. https://doc.qt.io/qt-5/wasm.html. Documentation Website.

[64] Tobias Wrigstad. Dala: A simple capability-based dynamic language design for data race-freedom. https://www.youtube.com/watch?v=2Su47a8cxuw, 2021. YouTube Video of Presentation from Onward! Papers 2021, part of SPLASH 2021.

[65] Trustworthy Systems. Using Rump kernels to run unmodified NetBSD drivers on seL4. https://research.csiro.au/tsblog/using-rump-kernels-to-run-unmodified-netbsd-drivers-on-sel4/, March 2017. Trustworthy Systems Blog Post.

[66] Trustworthy Systems Team, Data61. seL4 Manual - v12.1.0. http://sel4.systems/Info/Docs/seL4-manual-12.1.0.pdf, .

[67] Trustworthy Systems Team, Data61. Release Process | seL4 docs. https://docs.sel4.systems/processes/release-process, accessed: November 10 2022, .

[68] M. von Tessin. *The Clustered Multikernel: An Approach to Formal Verification of Multiprocessor Operating-System Kernels*. PhD thesis, School of Computer Science and Engineering, UNSW, Sydney, Australia, Sydney, Australia, Dec. 2013.

[69] W3C First Public Working Draft Document. WebAssembly Core Specification. URL https://www.w3.org/TR/2022/WD-wasm-core-2-20220419/.

[70] W3C Recommendation Document. WebAssembly Core Specification. URL https://www.w3.org/TR/wasm-core-1/.

[71] WasmEdge (Cloud Native Computing Foundation Sandbox project). Supported WASM And WASI Proposals - WasmEdge Runtime. https://wasmedge.org/book/en/features/proposals.html. Page from WasmEdge documentation, Accessed: 2022-11-08, Last update: 2021-10-27 (see documentation source at https://github.com/WasmEdge/WasmEdge/blob/master/docs/book/en/src/features/proposals.md).

[72] WebAssembly System Interface Subgroup. WASI/Proposals.md at main · WebAssembly/WASI. https://github.com/WebAssembly/WASI/commits/main/Proposals.md, 2019. Document in GitHub Repository, Accessed: 2022-11-08, Last update: 2022-10-04.

[73] WebAssembly System Interface Subgroup. WASI/README.md at main · WebAssembly/WASI. https://github.com/WebAssembly/WASI/blob/main/README.md, 2019. README of GitHub Repository, Accessed: 2022-07-11, Last update: 2022-03-02.

[74] WebAssembly System Interface Subgroup. WASI |. https://wasi.dev/, 2019. Website, Accessed: 2022-11-08.