# Capable VMs

23 Apr 24 - CHERITech '24

Jacob Bramley, Andrei Lascu, Duncan Lowther, Dejice Jacob,
**Jeremy Singer**, Laurie Tratt and Tom Wallis

**Virtual machines** (VMs, also known as managed language runtimes) are ubiquitous components in the modern software stack. They power the web, running in client-side browsers, server-side applications, and smartphone apps. In any ranking of popular programming languages, at least half of the top ten languages run on VMs (e.g. Python, Java, C#, JavaScript, PHP).

A key problem is that **VM security** has traditionally been a secondary concern relative to performance. Industrial strength VMs have large, complex code-bases, and large numbers of hand-crafted optimizations. Not only are they beyond any one person's ability to understand, but security has tended to be treated reactively: mature, widely used VMs such as HotSpot (the standard Java VM) regularly have 50-100 CVEs per year.

The **CapableVMs** project hypothesises that **CHERI** hardware-enforced **capabilities** are the first realistic technique to **make VM security proactive**.

# Why are virtual machines special?

lots of low-level platform-specific systems code

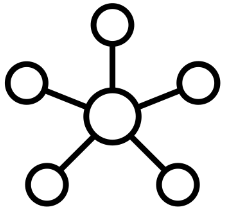
multiple interacting dynamic components

intensive allocation and garbage collection

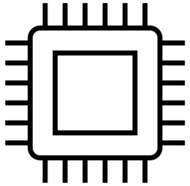
runtime code generation

# Why are virtual machines special?

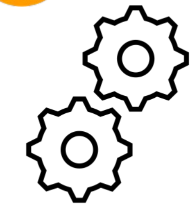
lots of low-level platform-specific systems code



multiple interacting dynamic components



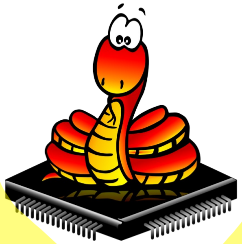
intensive allocation and garbage collection

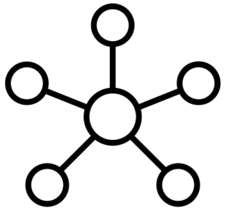


runtime code generation

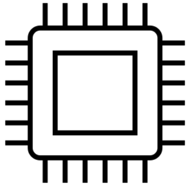
# Why are virtual machines special?

lots of low-level
platform-specific
systems code

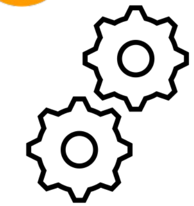




multiple interacting
dynamic
components



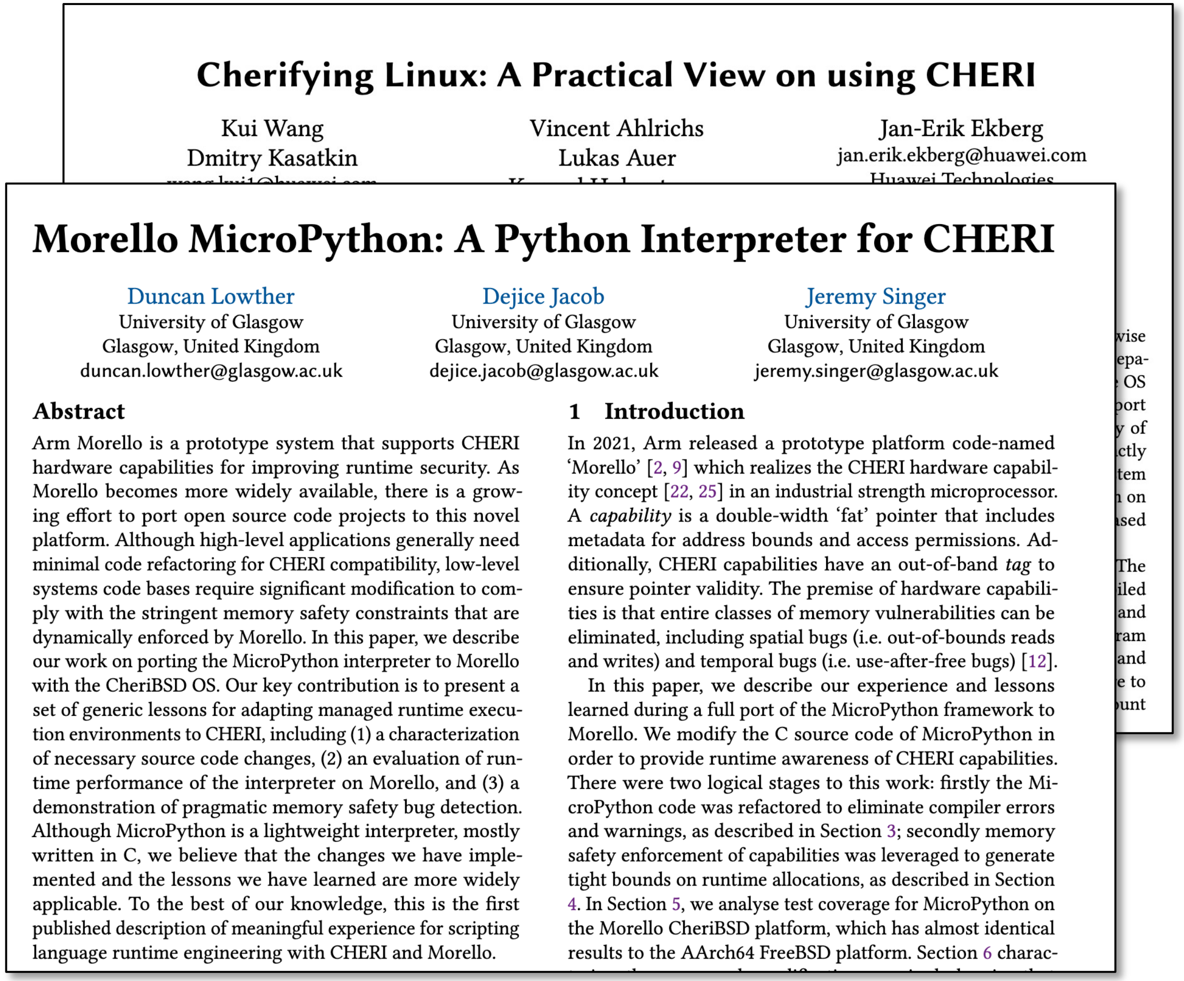
intensive
allocation and
garbage collection



runtime code
generation

# 1. Low-level system-specific code

- CHERIfication
- Specific porting process
- Measure proportion of LoC alterered
- KDE: 0.026%
- higher for systems code
  - 0.18% for MicroPython
  - 1% for snmalloc



**Cherifying Linux: A Practical View on using CHERI**

Kui Wang, Dmitry Kasatkin — Vincent Ahlrichs, Lukas Auer — Jan-Erik Ekberg, jan.erik.ekberg@huawei.com

**Morello MicroPython: A Python Interpreter for CHERI**

Duncan Lowther, University of Glasgow, Glasgow, United Kingdom, duncan.lowther@glasgow.ac.uk

Dejice Jacob, University of Glasgow, Glasgow, United Kingdom, dejice.jacob@glasgow.ac.uk

Jeremy Singer, University of Glasgow, Glasgow, United Kingdom, jeremy.singer@glasgow.ac.uk

**Abstract**

Arm Morello is a prototype system that supports CHERI hardware capabilities for improving runtime security. As Morello becomes more widely available, there is a growing effort to port open source code projects to this novel platform. Although high-level applications generally need minimal code refactoring for CHERI compatibility, low-level systems code bases require significant modification to comply with the stringent memory safety constraints that are dynamically enforced by Morello. In this paper, we describe our work on porting the MicroPython interpreter to Morello with the CheriBSD OS. Our key contribution is to present a set of generic lessons for adapting managed runtime execution environments to CHERI, including (1) a characterization of necessary source code changes, (2) an evaluation of runtime performance of the interpreter on Morello, and (3) a demonstration of pragmatic memory safety bug detection. Although MicroPython is a lightweight interpreter, mostly written in C, we believe that the changes we have implemented and the lessons we have learned are more widely applicable. To the best of our knowledge, this is the first published description of meaningful experience for scripting language runtime engineering with CHERI and Morello.

**1 Introduction**

In 2021, Arm released a prototype platform code-named 'Morello' [2, 9] which realizes the CHERI hardware capability concept [22, 25] in an industrial strength microprocessor. A *capability* is a double-width 'fat' pointer that includes metadata for address bounds and access permissions. Additionally, CHERI capabilities have an out-of-band *tag* to ensure pointer validity. The premise of hardware capabilities is that entire classes of memory vulnerabilities can be eliminated, including spatial bugs (i.e. out-of-bounds reads and writes) and temporal bugs (i.e. use-after-free bugs) [12].

In this paper, we describe our experience and lessons learned during a full port of the MicroPython framework to Morello. We modify the C source code of MicroPython in order to provide runtime awareness of CHERI capabilities. There were two logical stages to this work: firstly the MicroPython code was refactored to eliminate compiler errors and warnings, as described in Section 3; secondly memory safety enforcement of capabilities was leveraged to generate tight bounds on runtime allocations, as described in Section 4. In Section 5, we analyse test coverage for MicroPython on the Morello CheriBSD platform, which has almost identical results to the AArch64 FreeBSD platform. Section 6 charac-

```python
import dodgylib

tiny1 = bytearray(3)
tiny2 = bytearray(12)

# setup 'Oh' raw string
tiny1[0] = 0x4f  # O
tiny1[1] = 0x68  # h
tiny1[2] = 0x00  # \0

# setup 'Hello' raw string
tiny2[0] = 0x48  # H
tiny2[1] = 0x65  # e
tiny2[2] = 0x6c  # l
tiny2[3] = 0x6c  # l
tiny2[4] = 0x6f  # o
tiny2[5] = 0x00  # \0
```

```python
print(tiny1.decode('utf-8'))
dodgylib.dodgy(tiny1)
print(tiny2.decode('utf-8'))
```

```
root@amarena:~ #
./micropython-hybrid
exploit.py

Oh
HACK!!
```

```
import uctypes as uct

def dodgy(x):
    ptr = uct.addressof(x)
    unsafe =
uct.bytearray_at(ptr, 2000)
    i = 0
    while unsafe[i] != 0x65 or
unsafe[i+1] != 0x6c:
        i += 1
        if i > 2000:
            break
    unsafe[i] = 0x41
    unsafe[i+1] = 0x43
    unsafe[i+2] = 0x4b
    unsafe[i+3] = 0x21
    unsafe[i+4] = 0x21
    unsafe[i+5] = 0x00
    return
```

```
print(tiny1.decode('utf-8'))
dodgylib.dodgy(tiny1)
print(tiny2.decode('utf-8'))
```

```
root@amarena:~ #
./micropython-purecap
exploit.py
Oh
In-address space security
exception (core dumped)
```

# CVE-2023-7158 Detail

## Description

A vulnerability was found in MicroPython up to 1.21.0. It has been classified as critical. Affected is the function slice_indices of the file objslice.c. The manipulation leads to heap-based buffer ove
the attack remotely. The exploit has been dis
be used. Upgrading to version 1.22.0 is able t
recommended to upgrade the affected comp
vulnerability is VDB-249180.

# CVE-2023-7152 Detail

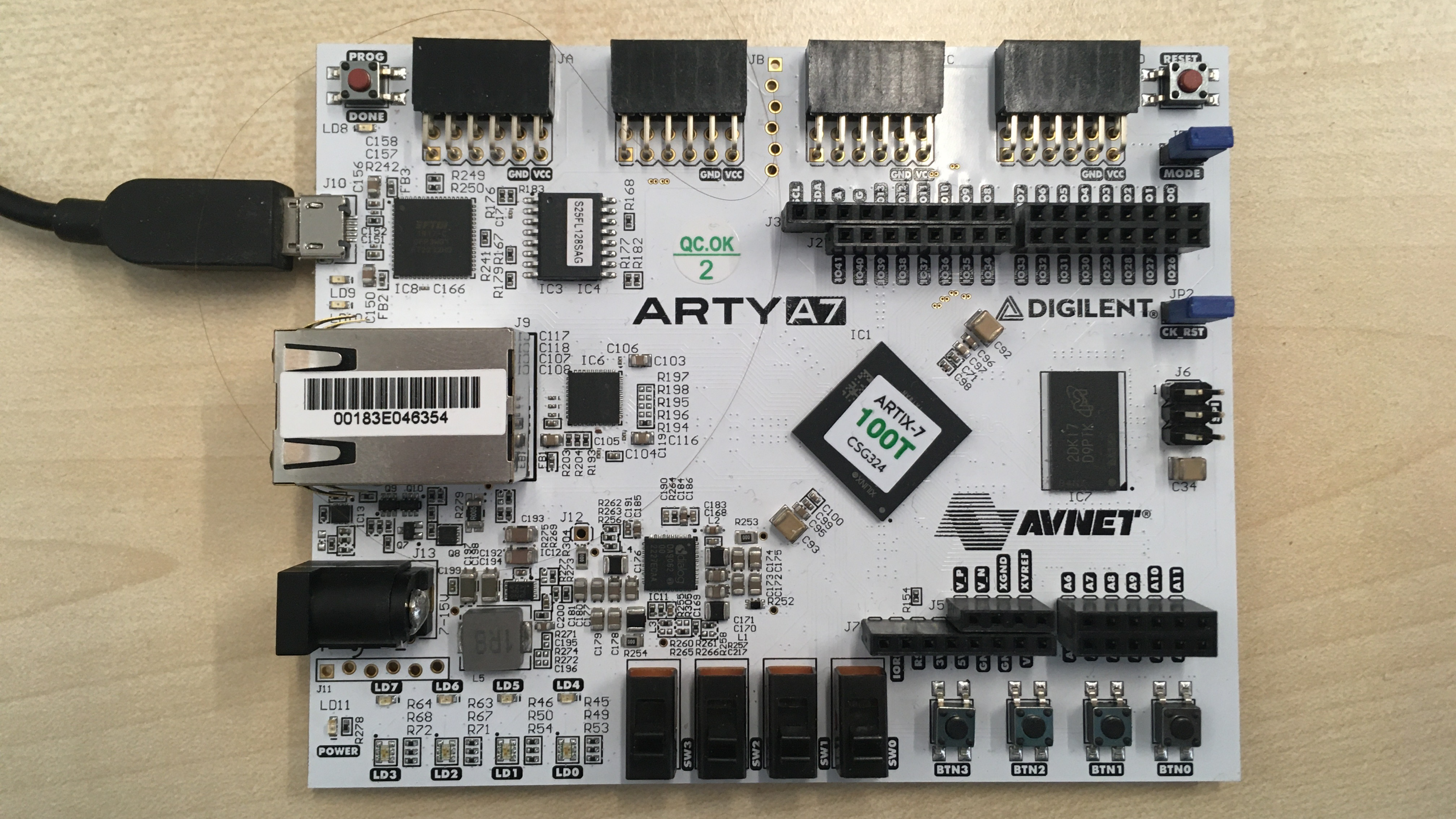
## Description

A vulnerability, which was classified as critical, has been found in MicroPython 1.21.0/1.22.0-preview. Affected by this issue is the function poll_set_add_fd of the file extmod/modselect.c. The manipulation leads to use after free. The exploit has been disclosed to the public and may be used. The patch is identified as 8b24aa36ba978eafc6114b6798b47b7bfecdca26. It is recommended to apply a patch to fix this issue. VDB-249158 is the identifier assigned to this vulnerability.

# Other findings from MicroPython

- Pointer size assumptions
  - don't affect correctness only
  - they also have an impact on performance

- Porting to a variety of platforms
  - Morello:  github.com/glasgowPLI/micropython
  - working on CHERIoT RISC-V Ibex core


PROG
DONE
RESET
MODE
CK_RST
QC.OK
2
ARTY A7
DIGILENT
ARTIX-7
100T
CSG324
AVNET
S25FL128SAG
00183E046354
POWER
BTN3
BTN2
BTN1
BTN0
SW3
SW2
SW1
SW0
LD0
LD1
LD2
LD3
LD4
LD5
LD6
LD7

# Why are virtual machines special?

lots of low-level platform-specific systems code

multiple interacting dynamic components

intensive allocation and garbage collection

runtime code generation

# 2. Multiple interacting components

- compartmentalization (c18n) is **lightweight isolation**
- hybrid code enables DDC-based isolation
- need a **compartment switcher**
- need a **libc** per compartment
- need clever tricks to handle dynamic loading
- overhead - how small should each compartment be?
  - compartment per *function*
  - compartment per *shared object*
  - alternative compartment boundaries?

# Alternative c18n strategy

- For purecap MicroPython code
- We isolate at FFI boundaries
  - e.g. calls to external C libraries
  - (work in progress)

# Why are virtual machines special?

lots of low-level platform-specific systems code

multiple interacting dynamic components

intensive allocation and garbage collection

runtime code generation

# 3. Malloc and GC

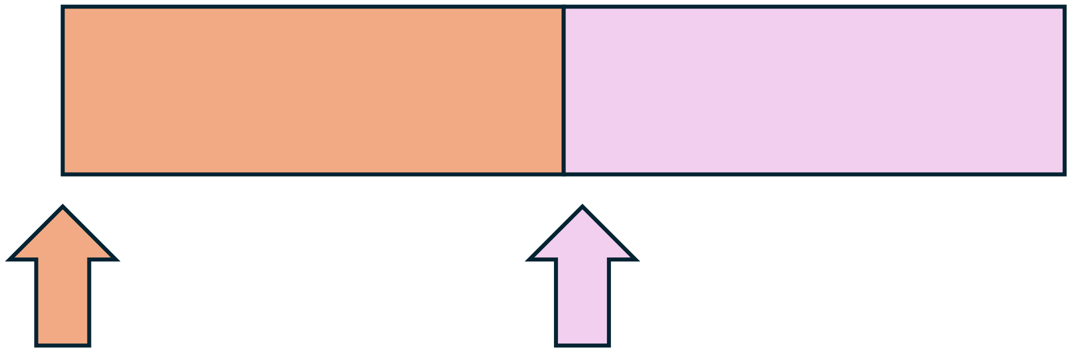
Complications include:

- finding and tracing the root set
- scanning the full heap
- moving objects

- We have studied **BDWGC**
- Morello & RISC-V: github.com/capablevms/bdwgc

# Observations about purecap GC

- Can't afford to lose capability tags
- conservative -> precise
- overhead reduction!
- issue with sealed caps in userspace code
- issue with coalescing



```
int main() {
  int i, size, *buffer;

  srand(SEED);

  for (i=0; i<NUM_ALLOCS; i++) {
    size = rand() % 1024;
    // printf("alloc buffer size %d\n", size);
    buffer = (int *)malloc(size * sizeof(int));

    // confuse conservative collector?
    savebuffer[i%(NUM_ALLOCS>>5)] = (long int)buffer;
  }

  return 0;

}
```

# Why are virtual machines special?

lots of low-level platform-specific systems code

multiple interacting dynamic components

intensive allocation and garbage collection

runtime code generation

# 4. Runtime code generation

- Several baseline interpreters ported to Morello purecap: WARDuino, MicroPython, JSC
- Some investigations on runtime code generation: JSC (& v8)
- This is work-in-progress

# Summary

- Our Capable VMs project has demonstrated that

**CHERI *does* provide defence-in-depth against VM-based exploits**

Challenges include:

1. how to quantify additional **defence**?
2. how to measure **performance**?
3. how to encourage **adoption**?