# Chromium and V8 on Morello

**(Developing and Evaluating an Open-Source Desktop for Arm Morello -** Innovate Project 10027440)

CHERITech-24, Cambridge, 2024-04-24

Dr Graeme Jenkinson
Director of Applied Technology | Capabilities Limited

CAPABILITIES LIMITED

# Chromium web browser

Chromium is the open-source web-browser project primarily maintained by Google. As well as being the base for Google's Chrome browser, Chromium is also used as the base for Microsoft Edge, Opera and app frameworks such as Electron.

# Why port Chromium to Morello?

- Hardening web browsers against vulnerability and exploitation is critical, as they are important platforms for human-machine interaction. These environments contain complex multi-million lines code bases rich in vulnerabilities.
- A number of research projects have sought to develop memory-safe web browsers. However, they compromise compatibility with contemporary websites. Providing a memory-safe web-browser that is both widely used and preserves compatibility with deployed web sites remains a challenging research goal.
- Chromium is extremely large, estimated around between 20 and 40 million lines of code, primarily written in unsafe C and C++. Porting the majority of this code to a memory safe language is economical unviable; with large direct and opportunity costs. Porting Chromium helps to validate one of CHERI's key propositions, that it offers a route to memory safety for large, legacy C and C++ applications.
- CHERI-based fine-grained, performant, compartmentalisation allows the redesign of Chromium's security model. This is especially important on mobile platforms where device resources restrict use of process-based isolation.

# Challenges (mix intrinsic and extrinsic factors)

- **Scale:**
  - Google engineers build Chromium as a distributed workload, a single Morello box is not well suited to these demands.
  - One of the few projects building tens of millions of lines of C++ code, issues with the Morello toolchain may be expected.
- **Complexity:**
  - High-level of complexity, especially in dependencies such as JS language runtimes.
- **Unsupported platform**:
  - As FreeBSD is not a supported platform, the Chromium project doesn't accept FreeBSD specific patches. This makes maintaining changes/patches for CHERI C/C++ and CheriBSD even more cumbersome and time consuming.
  - The baseline Chromium port is significantly behind mainline Chromium, and the project's development velocity is high. As a result makes many changes are immediately out of date.
  - Building the FreeBSD Chromium port is fragile, for example it requires a number of additional steps to replace libraries with system supplied versions or to link and copy files.
- **Third-party dependencies:**
  - Forking and patching the large number of third-party dependencies impacts productivity.
  - Many dependencies such as the matrix multiplication library ruy include optimizations in assembly that require extensive modifications.

My approach to the project is that this is a proof-of-concept. Good enough is OK, and so pragmatic changes are permissible, for example disabling assembly optimization or minor features.

# V8 Overview

V8 is an open-source, high-performance, Javascript and Web Assembly engine developed by Google and used by web browsers based on the open-source Chromium web browser including Google Chrome and Microsoft Edge. V8 is also integrated in application frameworks such as Electron and in the server-side Javascript environment Node.js.

V8 contains approximately 2 million lines of contemporary C++, with an additional several hundreds of thousands of lines generated from code written in V8's Torque DSL (along with approximately 600K lines of Javascripts primarily implementing builtin functions).

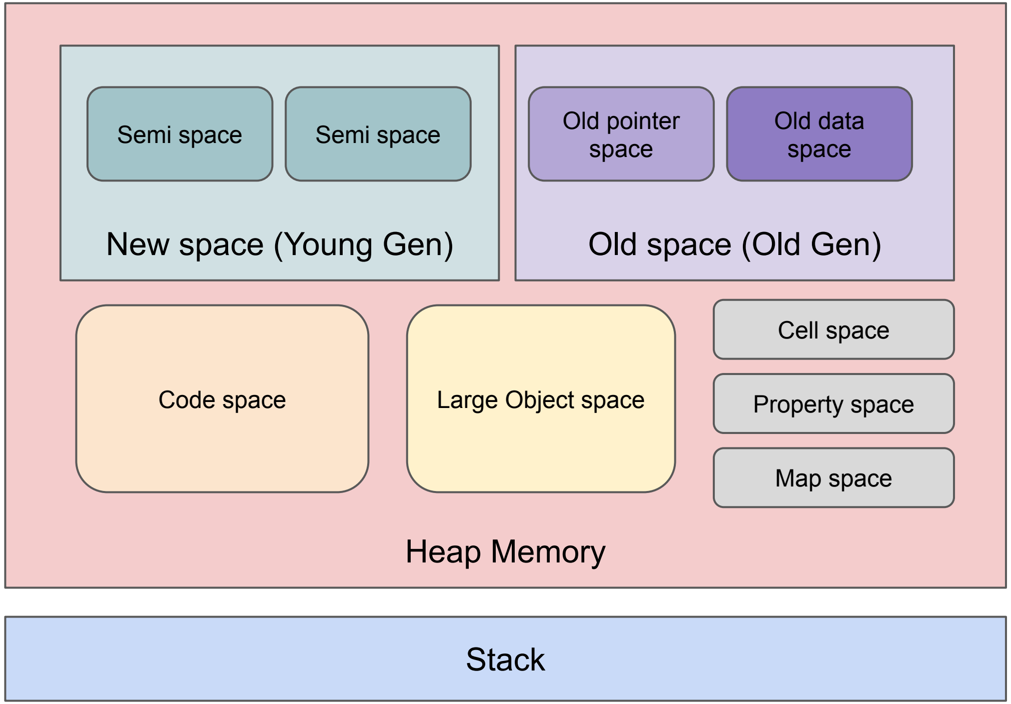
Language runtimes known to be significantly more disruptive to port to CHERI C/C++. Therefore, our initial plan was to postpone v8 until a working Chromium was available. However…

- Chromium cannot be built without support for Javascript.
- And v8 cannot be built without support for purecap code generation.

Therefore, our initial project goal is to port V8 to CHERI C/C++ to run on Morello.

Through this work we've learned a lot about V8, but still plenty to understand. My focus will be on design frictions between v8 and CHERI and how we've chosen to reconcile these issues.

# Overview of the v8 Memory Model



```
const myVar = 42;

const myArray = [];
```

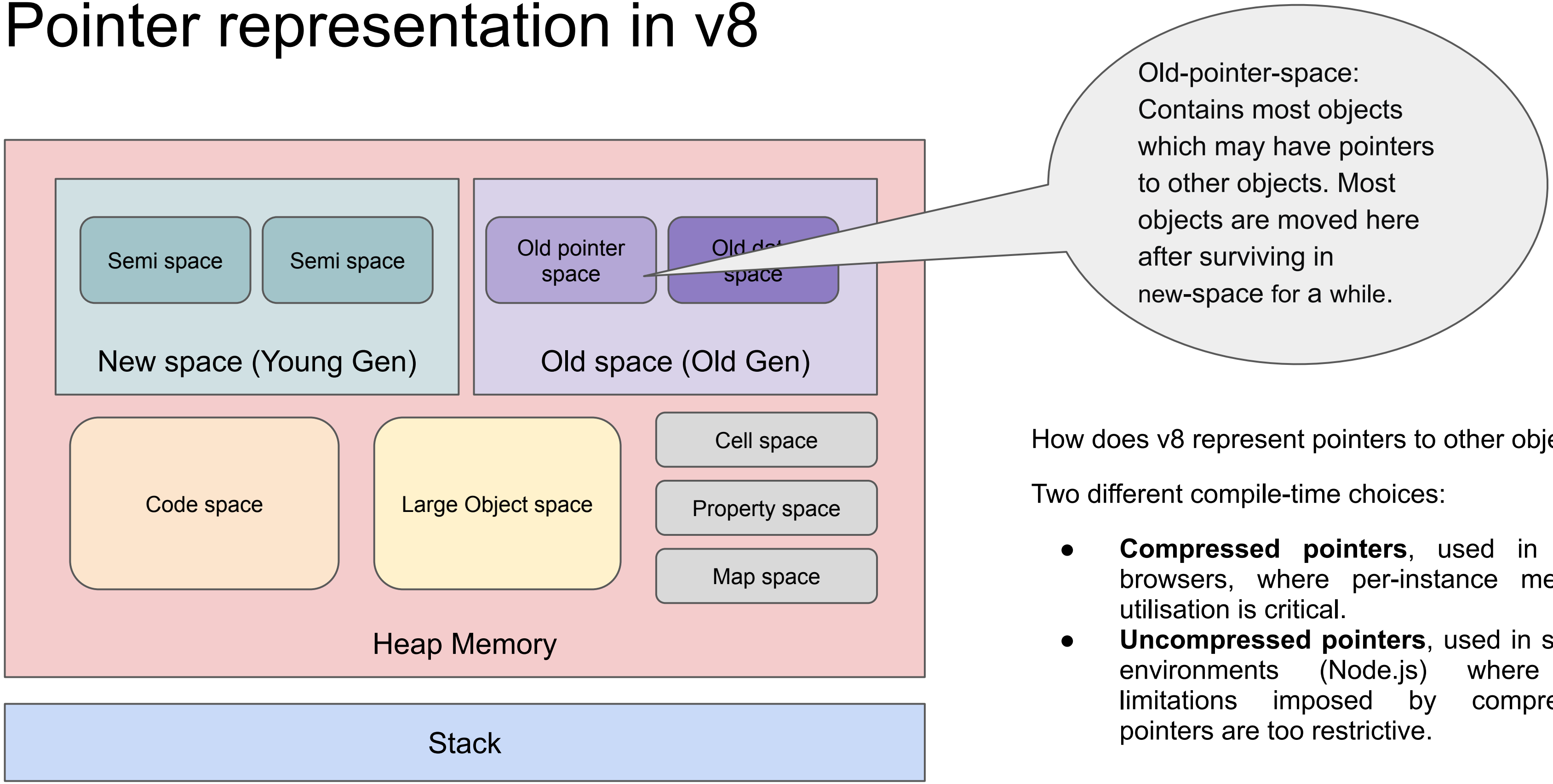
Call stack

| Address | Value |
|---|---|
| addr1 | 42 |
| addr2 | addr3 |

Heap

| Address | Value |
|---|---|
| addr3 | [] |

# Pointer representation in v8



Old-pointer-space: Contains most objects which may have pointers to other objects. Most objects are moved here after surviving in new-space for a while.

How does v8 represent pointers to other objects?

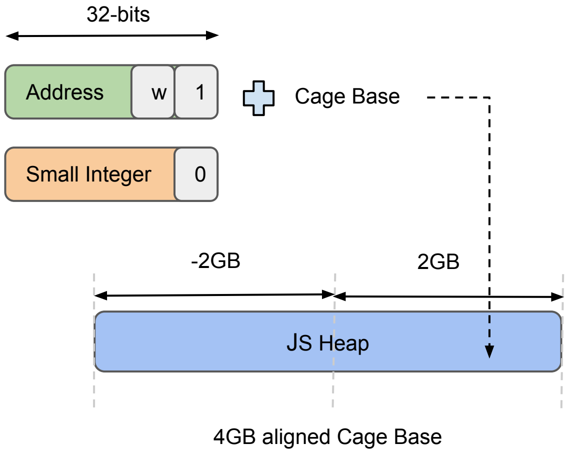
Two different compile-time choices:

- **Compressed pointers**, used in web browsers, where per-instance memory utilisation is critical.
- **Uncompressed pointers**, used in server environments (Node.js) where the limitations imposed by compressed pointers are too restrictive.

# Pointer representation in v8: 64-bit architecture

Compressed pointer representation

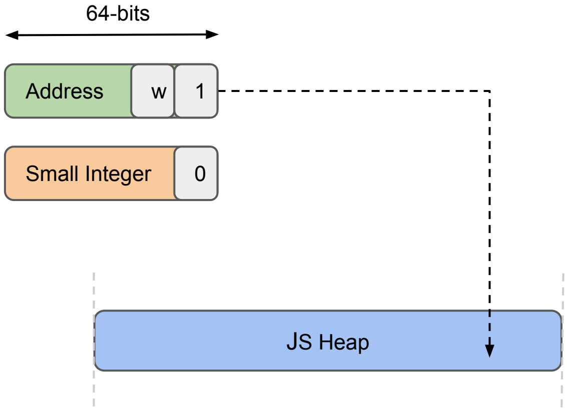
Pointers are offsets into a 4GB size, 4GB aligned memory region referred to as a compression cage.



*Main focus of porting work to date*

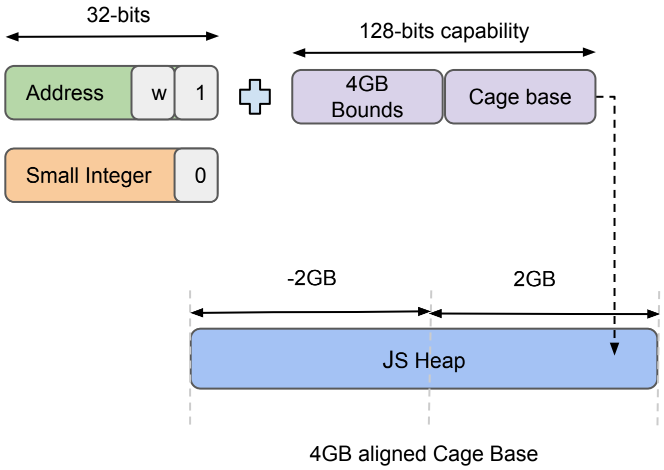
Uncompressed pointer representation

Pointers on the JS heap are integer pointer values of the same width as the native architecture



*Porting remains in the early stages*

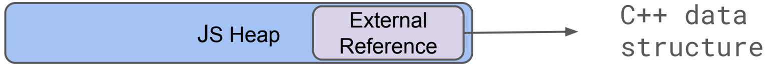
# Uncompressed pointer representation: Morello



*This all looks looks fairly promising, adapting to CHERI C/C++ involves ensure the cage base is a valid capability bounded to the compression cage*

But there are some problems lurking…

In addition to pointers to other JS objects, the v8 Javascript heap also contains external references; typically used to associate JS objects with C++ data structures

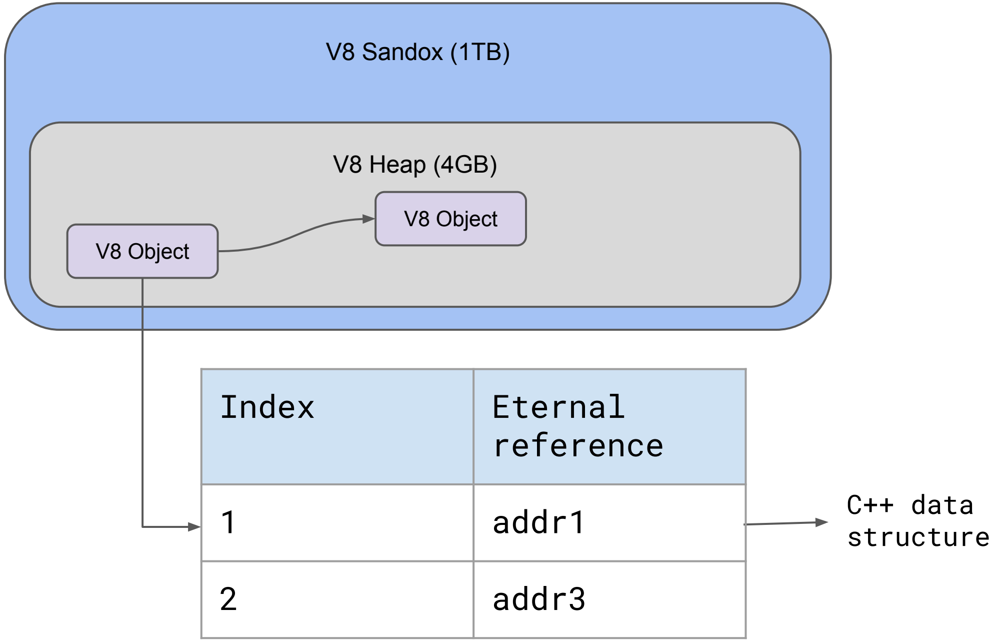


The V8 JS `ExternalReference` type wraps a `void *`, and therefore on Morello is a capability.

However, with uncompressed pointers the JS heap cannot align a value to a stronger value than `alignof(Tagged_t)`, that is 32-bits.

Other archs on v8 allow an unaligned load of this pointer

# V8 Heap Sandbox (`v8_enable_sandbox = true`)



| Index | Eternal reference |
|---|---|
| 1 | addr1 |
| 2 | addr3 |

"All problems in computer science can be solved by another level of indirection" - Butler Lampson

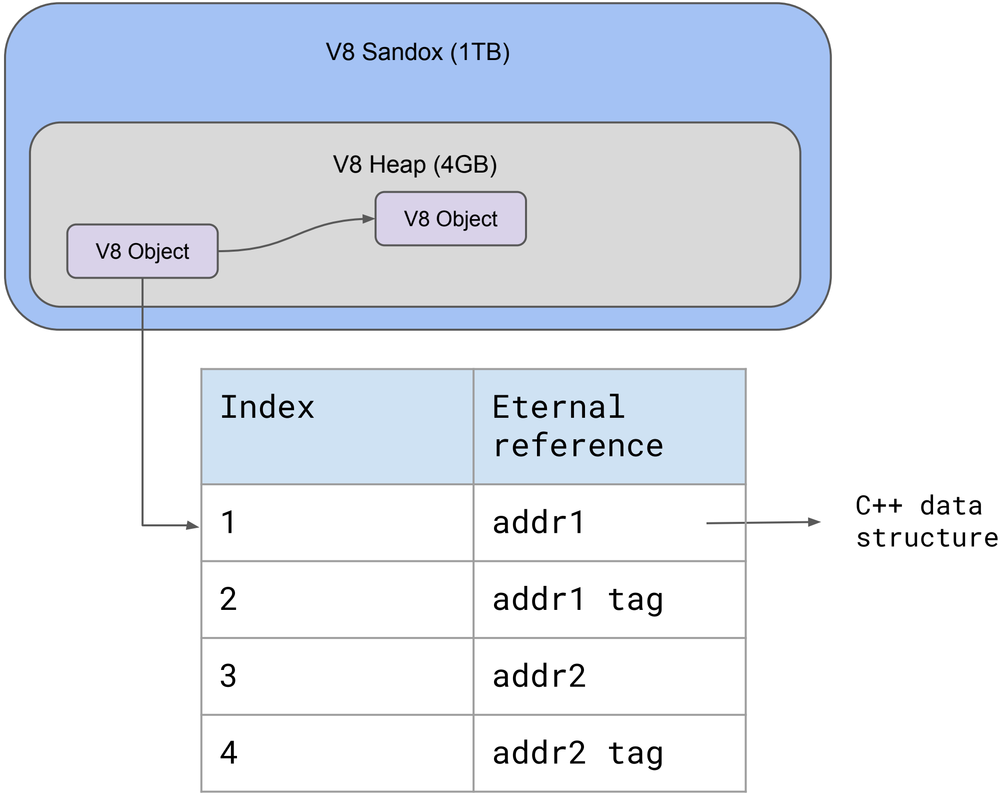
The software sandboxing is designed to isolate V8's (heap) memory such that any memory corruption there cannot "spread" to other parts of the process' memory.

All objects located outside the sandbox ("external entities") are referenced through pointer tables, which are themselves also located outside of the sandbox.

External references contain type information (checked on access) to prevent type confusion attacks: "It should be noted that this construction assumes that setting the top bits of a pointer to a nonzero value causes the pointer to become invalid."

*OK, well that isn't going to work*

# Redesigned V8 Heap Sandbox



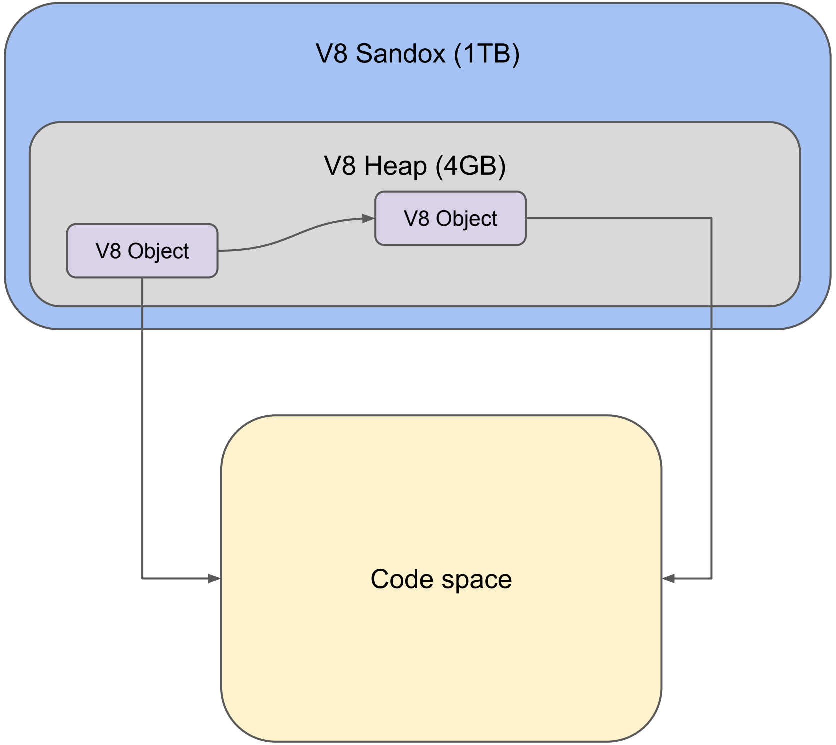
| Index | Eternal reference |
|---|---|
| 1 | addr1 |
| 2 | addr1 tag |
| 3 | addr2 |
| 4 | addr2 tag |

Simplest possible approach, store the capability and the tag separately in adjacent indexes in the external pointer table.

When an EPT entry is no longer used, it needs to be freed so it can be reused again later. No mechanism in V8 to execute a finalizer function (which could free any EPT entries) whenever a HeapObject is collected by the GC. As such, the table itself needs to be garbage collected. Slight modifications required to the garbage collection.

Other designs are available see Google's design document for the External Pointer Table: https://docs.google.com/document/d/1V3sxltuFjjhp_6grGHgfqZNK57qfzGzme0QTk0IXDHk/edit

# External code space



With the sandbox enable v8 must be built with an external, code space. Keeping in mind that it is not possible to store a capability on the JS heap, how then does a Javascript object refer to the code space?

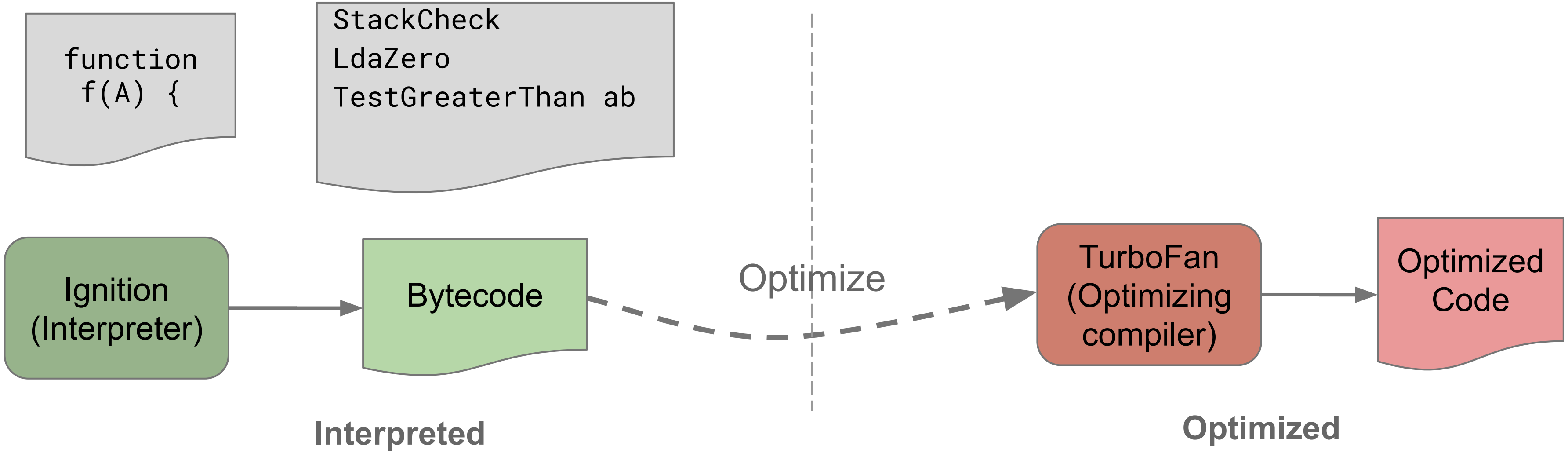
On the version of V8 being ported, the code space used the same pointer compression scheme as was shown for the uncompressed pointer build. Therefore, the code space was itself 4GB in size and 4GB aligned. Objects reference the code space by storing just the top 32-bits, shifting right to create the address

*OK, well that isn't going to work AGAIN*

New compression scheme added for the code space to preserve capabilities, code space aligned to the minimum OS page size, with single cage base or cage base per Isolate

# Code generation

V8's compilation pipeline has evolved significantly in recent years. From a high-level it is now fairly straightforward, though another mid-tier optimizing compiler Maglev somewhat complicated this picture



```
function
 f(A) {
```

```
StackCheck
LdaZero
TestGreaterThan ab
```

*Bytecode handlers are generated at build time, so even the interpreter needs working code generation!*

# Code generation backend

Changes to code generation include:

- Add new aarch64c (Morello) instructions such as capability specific load/stores and additions.
- Introduce capability width registers:
  - Ensure that the correct instruction is generated when operating on capability width registers.
  - Allow allocation of temporary capability width registers.
  - Update macros to use capability register; either where specific or temporary X (64-bit width) registers are used.
  - Update runtime checks to support capability width registers.
  - Introduce named capability registers such as `csp`.
  - Update named registers such as the V8 frame register `fp` to capability registers.
  - Update builtlins written in assembly to use capability registers; either where specific or temporary X (64-bit width) registers are used.
  - Update call descriptors to use capability registers for parameters where needed.
- Preserve `PointerRepresentation` of operands until code generation (in order to identify operations on pointer values).

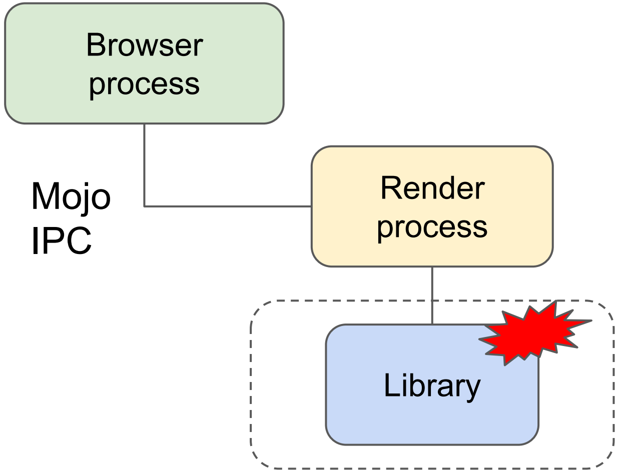
# Current status of v8 and future directions

- Uncompressed pointers:
  - Successfully ported a minimal viable v8, this can execute some Javascript and builtins.
  - More work required to resolve remaining tags faults during code generation.
  - More thorough testing!
- Compressed pointers
  - Adapt the v8 Torque DSL to pad data structures with capability values (now completed by Domagoj Stolfa, University of Cambridge)
  - Fix initial stack allocation (on going).
  - …
- Analysis of security and performance of uncompressed and compressed pointer representations.
- HW enforced sandboxing:
  - “In principle, the sandbox could be implemented with hardware support: similar to the userland-kernel split, V8 would execute some mode-switching instruction when entering or leaving sandboxed code”
  - CHERI-based compartmentalisation of existing design.
  - Re-visit the design in presence of fine-grained, performant compartmentalization.

# Chromium is currently WIP

- Initial challenging to even build Chromium
  - Together with KCL Chromepartments team, we managed to coerce the Chromium tools to fetch Chromium sources and dependencies.
  - Chromium can then be built straightforwardly with ninja; though some manually steps are still required.
- The `chrome` target builds over fifty thousand object, currently approximately 40% build.
  - With varying degrees of confidence.
- Ported third-party dependencies include:
  - Perfetto, system profiling, app tracing and trace analysis framework
  - Graphics libraries such as skia, dawn and angle.
- Memory safe Chromium prerequisite to explore compartmentalisation…

Future directions:



*For example, Library compartmentalisation can restricts access to IPC and therefore further attack surfaces in the Browser process*

# Qualitative porting experience

- Assertion of overly strong invariants is common.
  - For example, asserting the size of datatypes, that fail due to the stronger alignment requirements for capabilities.
- The arm64 `Assembler` and `MacroAssembler` include many checks that only fail at runtime.
  - Frequently check that register sizes are 32-bit or 64-bit, these fail for capability registers at runtime
- Many ad-hoc hecks for 64-bit pointers, else of course it must be 32-bit.
- Compiling V8 is slow:
  - V8's use of inlining (at the least in the ported version) significantly increases compilation times, many minutes for one object isn't uncommon.
  - Changes to the arm64 backend result in one thousand objects being rebuilt, sometimes that make sense other times it is a little confusing.
  - `ccache` and `distcc` do help at least a bit.
- Builtins written in assembler are painful to adapt:
  - Require extensive changes to operand registers; however it is not always clear whether a given value is a capability.
  - Difficult to debug when a capability tag is lost.
- Code uses `u/intptr_t` types to store an integer type that is either a 32-bit or 64-bit width depending on the architecture.
  - This use is widespread within Google.
  - Results in capability aware types being used where they are not needed (carrying cost), and/or confusion about where pointers/integers are being used.

# Conclusions

- Porting of Chromium to Morello is complicated by both its scale and complexity.
- Its is noticeable harder to make progress as FreeBSD is an unsupported platform.
    - Build tools don't work on FreeBSD without some effort.
    - Opaque patch set maintained in FreeBSD ports.
- We didn't expect to engage with porting of v8 early in the project, but found this to be a prerequisite for Chromium.
- Though we expected porting a language runtime to be difficult, v8 to be difficult to proved even we expected:
    - Design fricitions with CHERI required more extensive changes .
    - Guidance that the v8 JIT could be disabled proved to be wrong, nothing works without working code generation for purecap.
    - Overly strong invariants common in the code base.
    - Incorrect usage of `u/intptr_t` types problematic for CHERI.
    - Much more handwritten assembler than envisaged for a language runtime!
- Significant value in demonstrating that CHERI-based compartmentalization can be used to strengthen or redesign the Chromium security model.
    - But lots of (ongoing) work to get to this point…

# Q & A