# Hardware-Assisted Memory Safety for WebAssembly

Fritz Rehde
Advisor: Martin Fink
Chair of Distributed Systems and Operating Systems
https://dse.in.tum.de/

**15.04.23 – 15.08.23**

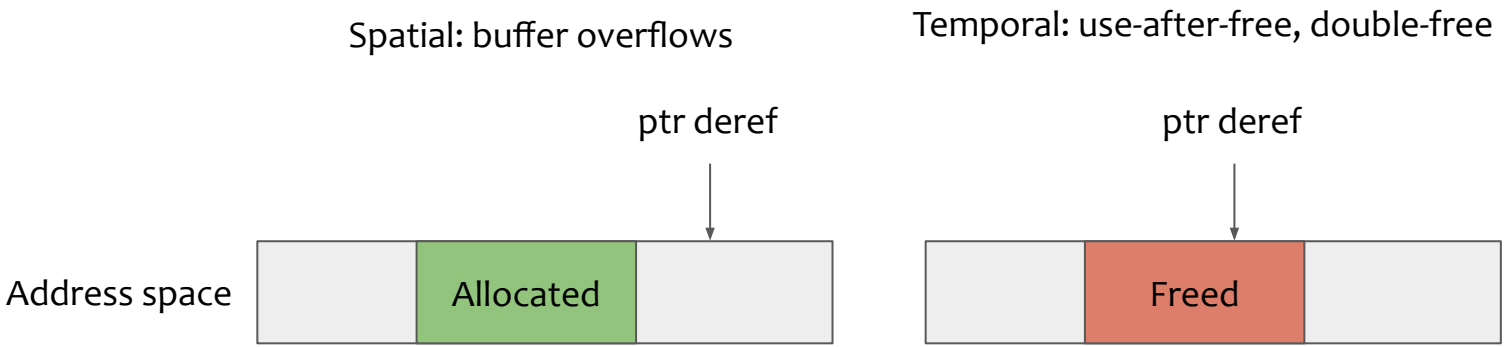
# WebAssembly (WASM)

- “Binary instruction format for a stack-based virtual machine”[1]
- “Compile once, run everywhere” (Portable)
- Environments: (Portable)
  - Web browser (V8)
  - Systems programming (Wasmtime) with WASI
- Compilation target for low-level programming languages, e.g. C (Performant)
- Sandboxing: protect host though bounds checking inside linear memory (Secure)
  - Does not protect module from itself within sandbox!

[1] https://webassembly.org/

# Memory Safety

- Program is memory safe if all memory pointers refer to valid memory when being dereferenced



# Research question

How can we provide memory safety in WebAssembly without incurring significant performance costs?

System design goals:

- **Safety:** Memory safety issues should cause runtime crashes instead of UB
- **Performance:** Minimal performance overhead
- **Usability:** No need to modify input C source code

# Approach

ARM64 Hardware Extensions

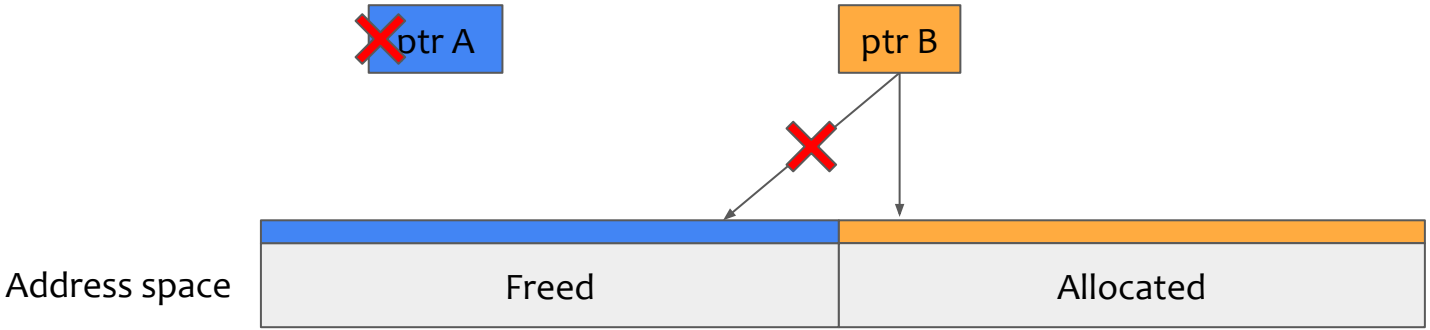
- Store metadata in unused upper bits in pointers

# Outline

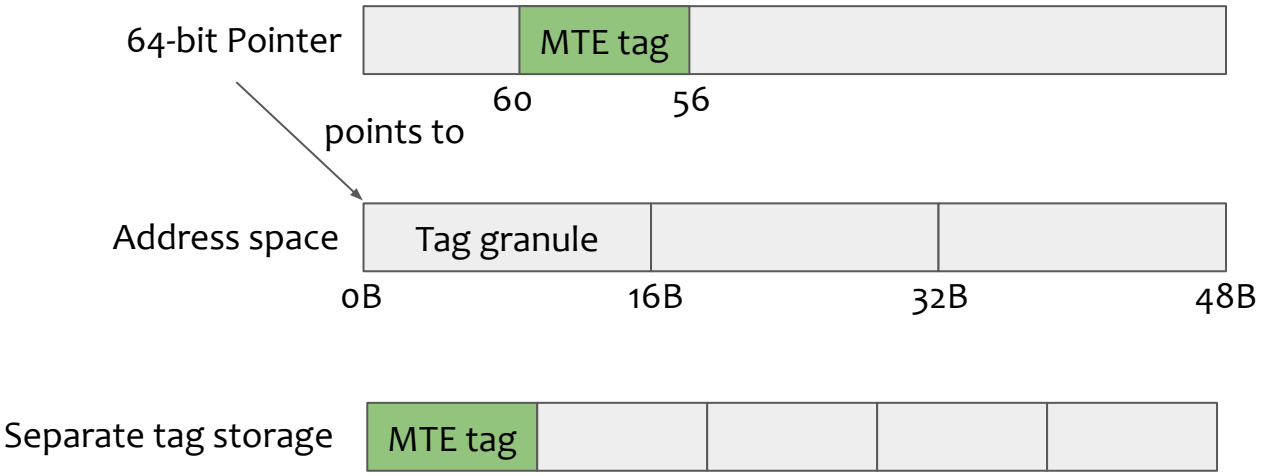
- ~~Motivation~~
- Background
  - Memory Tagging Extension (MTE)
  - Pointer Authentication (PAC)
- Design
- Evaluation

# ARM64 Hardware Extensions: MTE

- Memory Tagging Extension (MTE) in ARMv8.5:
  - Ensures spatial and temporal memory safety
  - Pointers and memory locations are tagged (“colored”)
  - Tags are compared on memory access



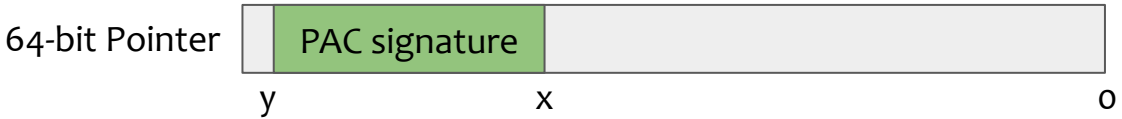
# ARM64 Hardware Extensions: MTE




| | |
|---|---|
| `IRG(ptr) -> tagged_ptr` | "Insert Random Tag" into pointer |
| `STG(tagged_ptr, ptr)` | "Store Allocation Tag" in memory at 16-byte granularity |

# ARM64 Hardware Extensions: PAC

- Pointer Authentication in ARMv8.3:
  - Protects against malicious pointer overrides (e.g. ROP, JOP)
  - Pointer Authentication Code (PAC): cryptographic signature embedded in pointer




with y - x = 55 - <linux virtual address size>[1]

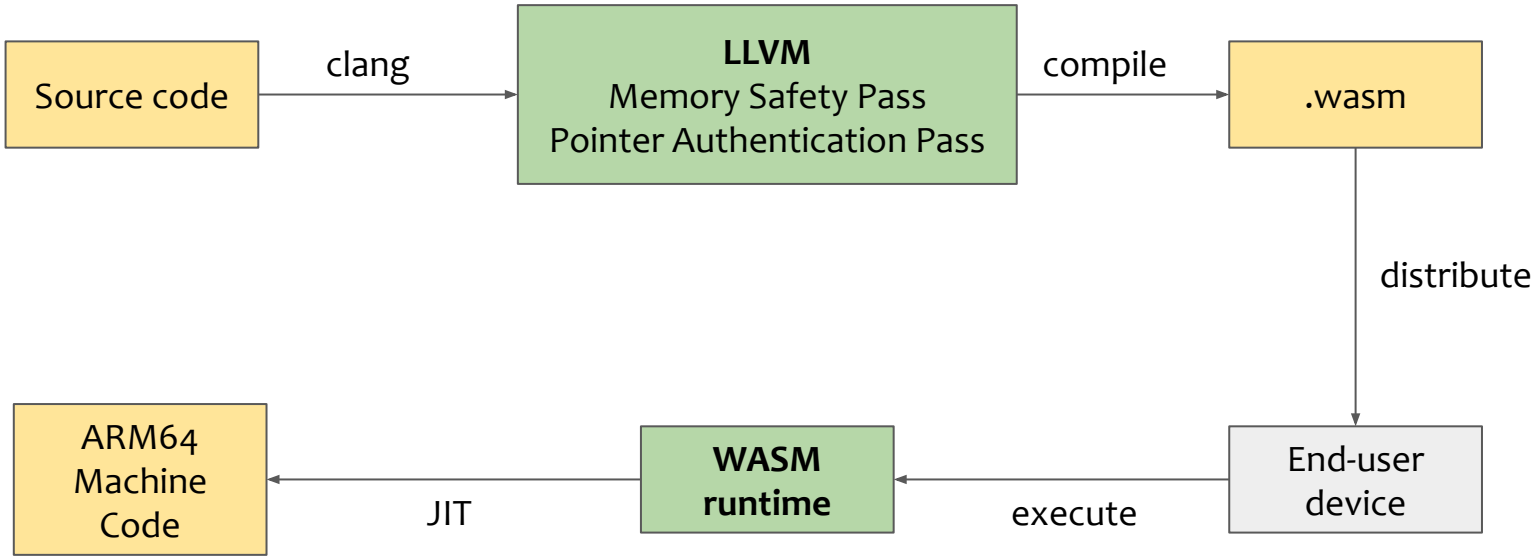
| | |
|---|---|
| `PACDA(ptr) -> signed_ptr` | Generate PAC, and sign pointer with it |
| `AUTDA(ptr) -> authed_ptr` | Authenticate pointer, remove PAC if succeeded |

[1] https://www.kernel.org/doc/html/v6.5/arch/arm64/pointer-authentication.html

# Outline

- ~~Motivation~~
- ~~Background~~
- Design
- Evaluation

# System overview



# Outline

- ~~Motivation~~
- ~~Background~~
- Design
  - New WASM Instructions
  - LLVM Passes
  - Wasmtime Additions
- Evaluation

# Design: new WASM instructions

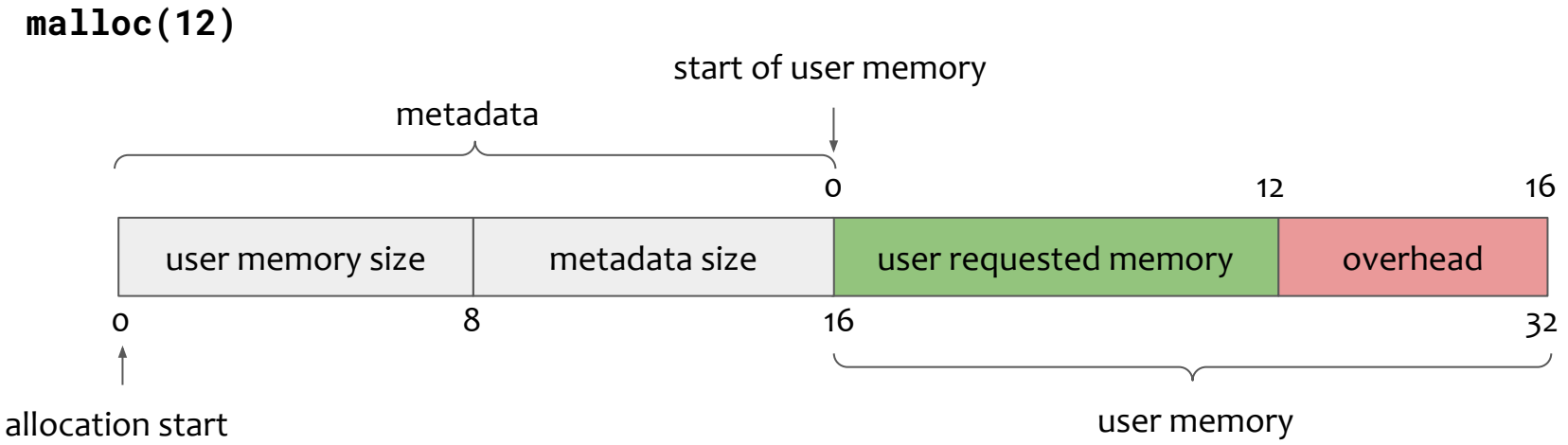
- Introduce *segments*: protected memory region, only accessible with matching pointer

| WASM instruction | Implementation in WASM runtime |
|---|---|
| `segment.new(index[1], size) -> tagged_index` | 1. Generate new (random) tag (IRG)<br>2. Tag index with new tag<br>3. Tag memory with new tag in 16-byte granules (STG) |
| `segment.free(index, size)` | Tag memory with free tag 0 (STG) |
| `pointer_sign(index) -> signed_index` | Sign index (PACDA) |
| `pointer_auth(index) -> authed_index` | Authenticate index (AUTDA) |

[1] An index is an i64

# Design: LLVM Memory Safety pass

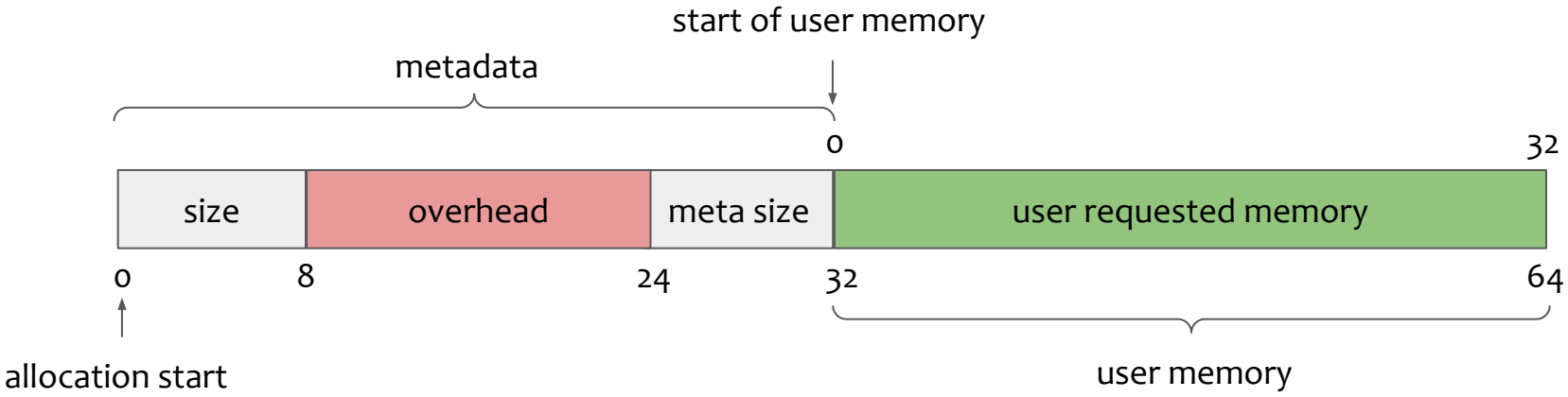
- Wrappers around aligned_alloc, malloc, calloc, realloc and free
- MTE requires 16-byte alignment, but we also need to handle custom alignment



# Design: LLVM Memory Safety pass

- Wrappers around aligned_alloc, malloc, calloc, realloc and free
- MTE requires 16-byte alignment, but we also need to handle custom alignment

**`aligned_alloc(32, 32)`**



# Design: LLVM Pointer Authentication pass

- Before storing a pointer value, sign it (with pointer_sign)
- After loading a pointer value, authenticate it (with pointer_auth)
- Rules for implementation:
  - **Signed** pointers **must be** authenticated before being dereferenced
  - **Non-signed** pointers **may not be** authenticated before being dereferenced
- Problem:
  - WASI-libc only provides interface (header files)
  - WASM runtime provides implementations
- Solution:
  - Cannot sign/authenticate pointers, which come from or are passed to “external functions”
  - External functions: function-, module- and program-granularity (WIP: LTO)

# Design: WASM runtime

- Modify WASM runtime *Wasmtime*[1] (Bytecode Alliance; Rust)
- Generate MTE and PAC instructions if target supports them
- Problem:
  - Random tag generation: neighbouring segments (stack) might have same tag
  - Buffer overflow not detected (probability: 1/15 = 6.67%)
- Solution[2]: (WIP)
  - Use IRG for first segment
  - Generate next tags by incrementing

[1] https://wasmtime.dev/
[2] https://arxiv.org/abs/2204.03781

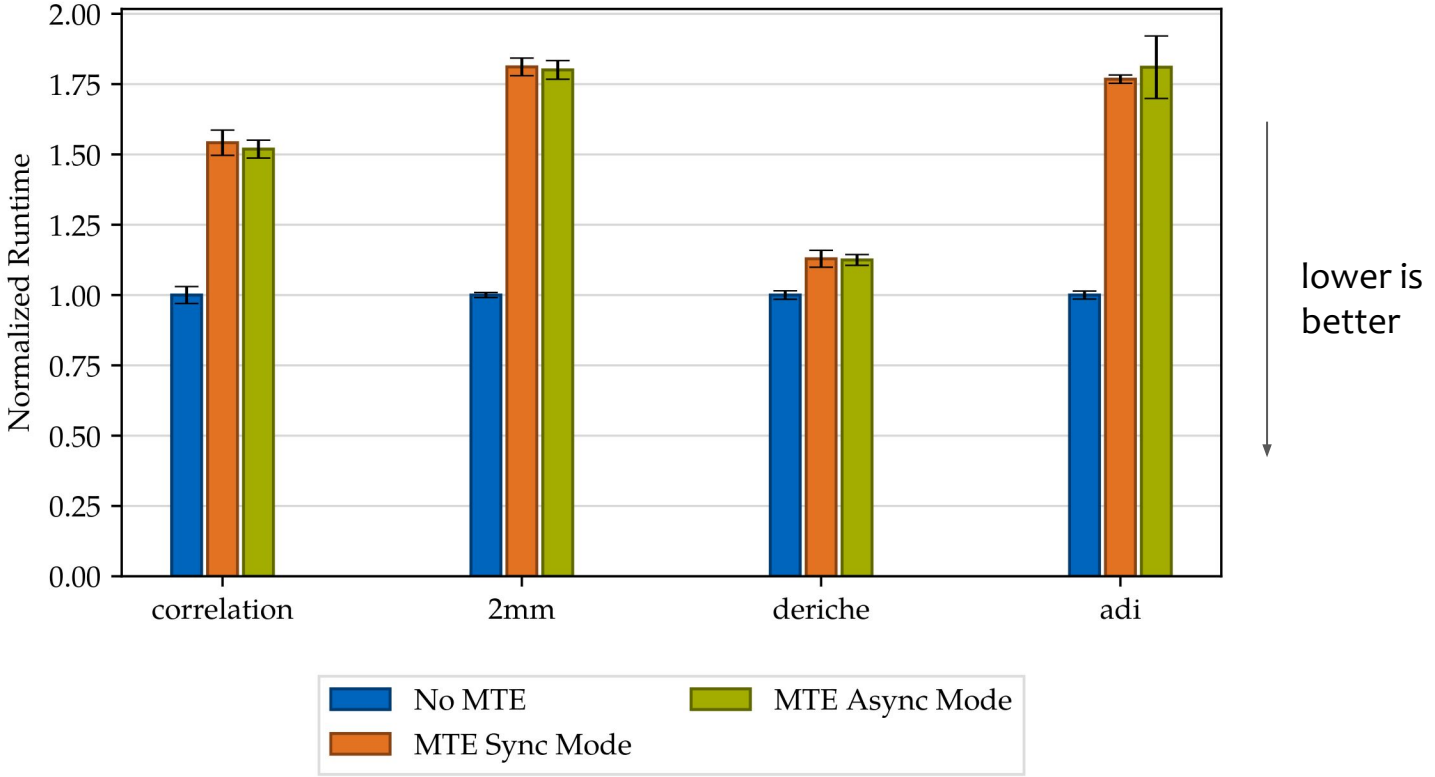
# Outline

- ~~Motivation~~
- ~~Background~~
- ~~Design~~
- Evaluation

# Evaluation: MTE

- Experimental setup:
  - AMD EPYC 7713P CPU (64 cores, 128 threads)
  - 515 GiB DDR4 RAM
  - x86, NixOS 23.05
- MTE not available on real hardware => QEMU

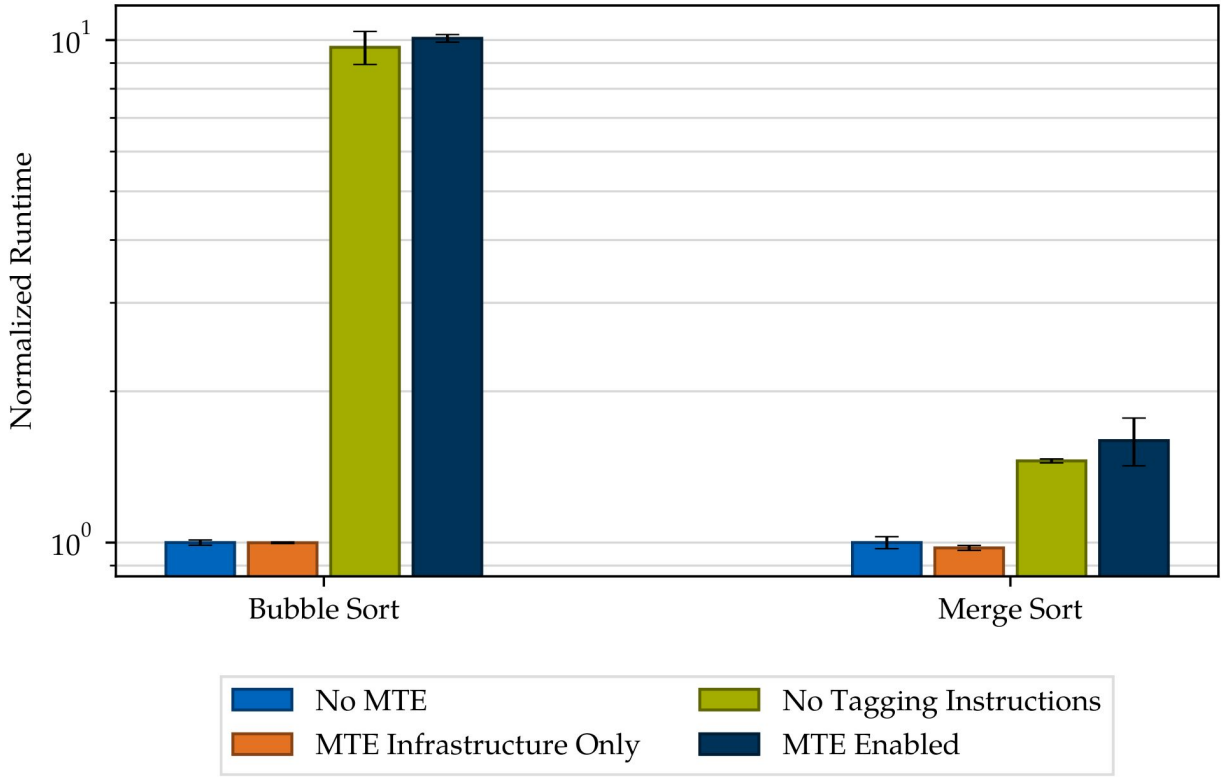
# Evaluation: PolybenchC[1]




[1] https://github.com/MatthiasJReisinger/PolyBenchC-4.2.1 (dataset: medium)

# Evaluation: Sorting Algorithms

Sort array of size 40,000

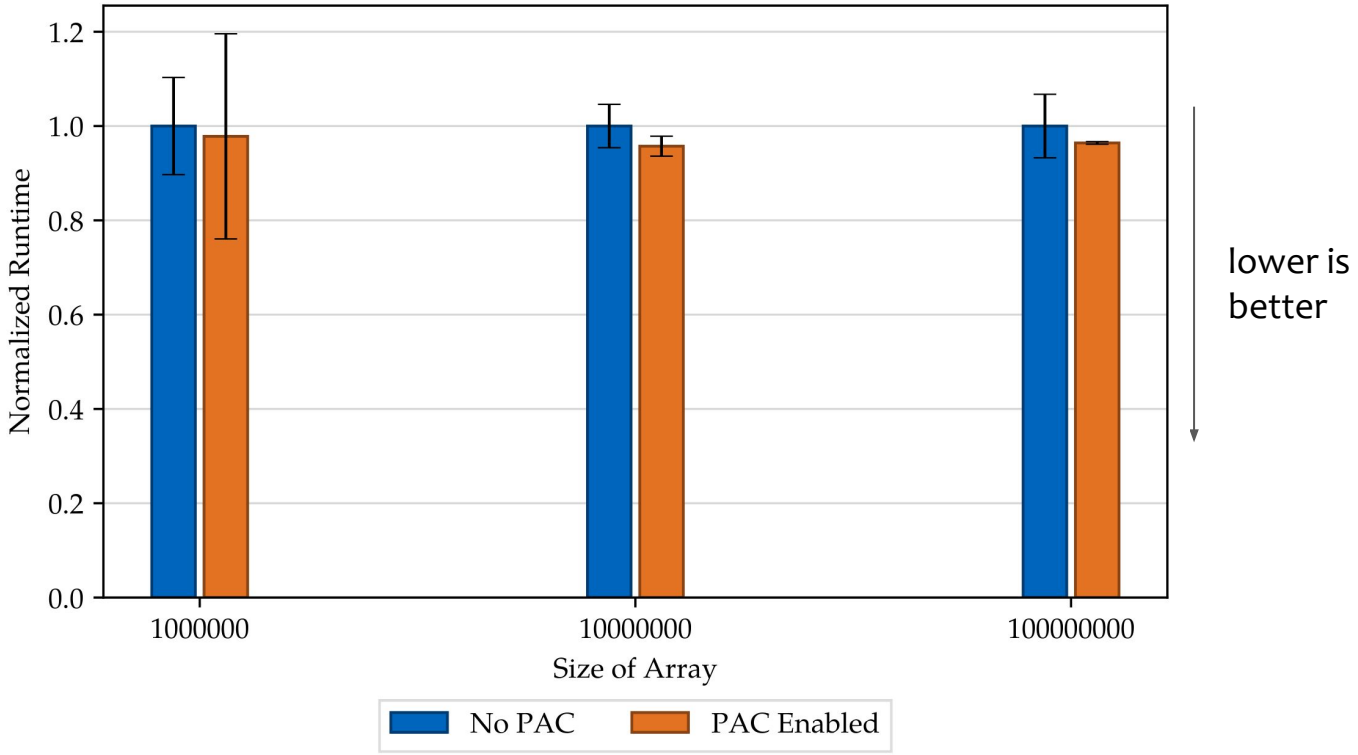


# Evaluation: PAC

- Experimental setup:
  - Apple M1 Pro
  - 10 cores (8 performance; up to 3220 MhZ, 2 low power)
  - 192 KB instruction cache, 128 KB data cache, 24 MB shared L2 cache
  - 16 GB LPDDR5-6400
- Supports PAC natively (through Apple's HVF hypervisor through QEMU)

# Evaluation: PAC Overhead

Store n pointers in pointer array (PACDA), load n pointers from pointer array (AUTDA)



# Summary

- Ensure memory safety in WASM:
  - New memory safety WASM instructions
  - LLVM IR passes to insert them
  - Wasmtime to insert ARM64 MTE and PAC instructions
- PAC: limited safety, but minimal overhead
- MTE: high safety, but real-world performance still unknown

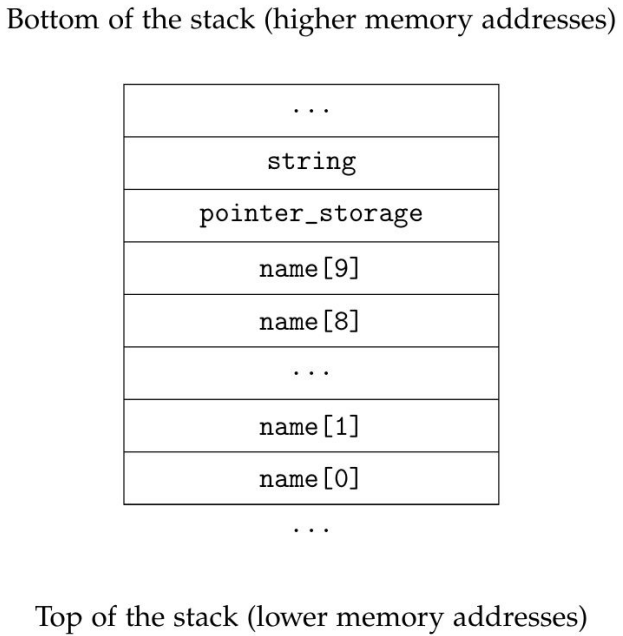
**Try it out!**

https://github.com/TUM-DSE/llvm-memsafe-wasm
https://github.com/TUM-DSE/wasmtime-mte

# Backup

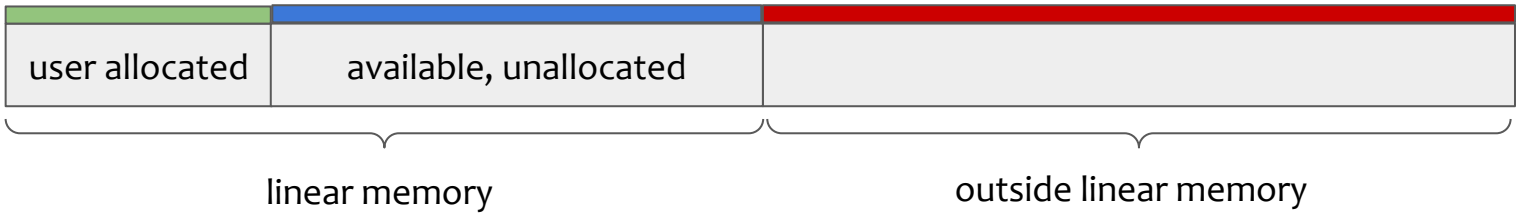
# Example: PAC protection

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    char *string = "Hello World!";
    char **pointer_storage = &string;
    char name[10];

    printf("What is your name?\n");
    scanf("%s", name);  // potential buffer overflow
    printf("Hello user %s!\n", name);

    char *loaded_string = *pointer_storage;  // failed authentication
    printf("String protected with PAC: %s\n", loaded_string);

    return 0;
}
```



# Work in Progress: Optimizing bounds checks using MTE

- Linear memory access: check index is in-bounds, to protect runtime/host
- Our idea to eliminate this overhead using MTE:
  - Tag outside linear memory with default free tag
  - Tag entire linear memory and freed user memory with linear memory free tag
  - Tag allocations inside with randomly generated tag



# Additions in Wasmtime

- Memory region tagging optimizations:
  - ST2G: “Store Allocation Tags”, tag 2 granules at once (32 bytes)
  - Dynamic size: ST2G loop, conditional STG
  - Static size: Loop unrolling threshold (= 160 bytes = 5 x ST2G)
- Error message on MTE trap:
  - Extend signal handler: catch SIGSEGV, check si_code for SEGV_MTESERR or SEGV_MTEAERR

# PAC Analysis in LLVM pass

Pointer cannot be signed/authenticated if:

- “Pointer value comes from elsewhere”:
  - Value was passed as parameter to the current function
  - Value was loaded from any memory location
  - Value is a global value
  - Value is return value of any function
- “Pointer value has other uses”:
  - Value is recursively passed as function parameter to an external function
- Aliases have to be checked as well

# Related Work

- Providing Memory Safety in WebAssembly:
  - MSWasm[1]: *segments* only accessible through *handles* (fat pointer); software-based; performance overhead
  - GC proposal[2]: introduce native GC instead of PLs having to ship own GC (large binaries)
- Providing Memory Safety using ARM64 Hardware Extensions
  - HWASan[3]: software-based tags using Top Byte Ignore (TBI); code size overhead
  - Deterministic Tagging[4]: LLVM analysis to differentiate between safe and unsafe memory allocations => tag only unsafe => performant

[1] https://arxiv.org/abs/2208.13583
[2] https://github.com/WebAssembly/gc
[3] https://source.android.com/docs/security/test/hwasan
[4] https://arxiv.org/abs/2204.03781