Cage

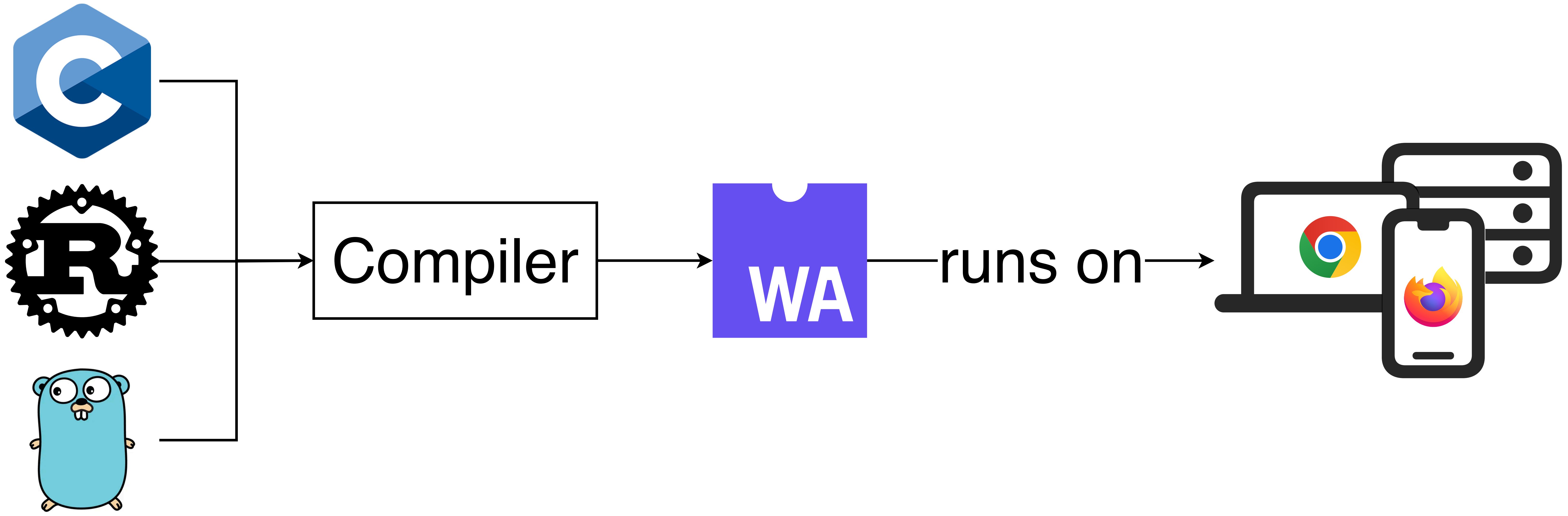
Hardware-Accelerated Safe WebAssembly

Martin Fink, Dimitrios Stavrakakis, Dennis Sprokholt, Soham Chakraborty, Jan-Erik Ekberg, and Pramod Bhatotia

CGO’25 | March 4th 2025 | Las Vegas, Nevada, USA

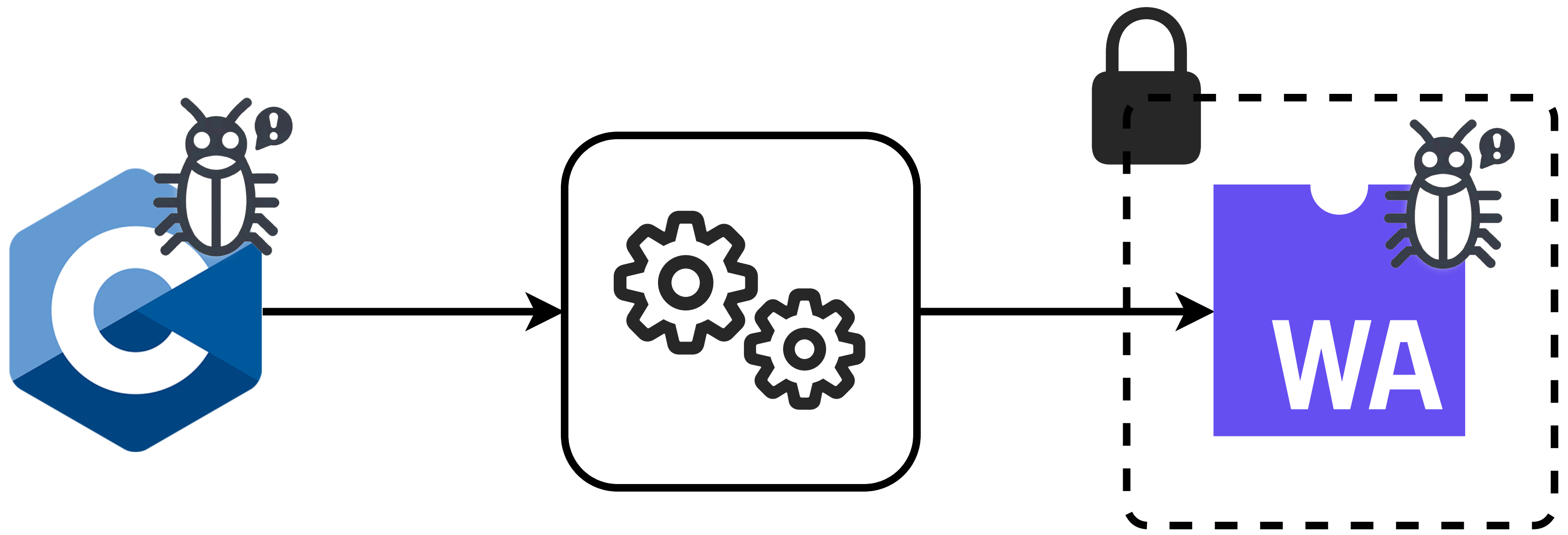
Technical University of Munich | Systems Research Group

WebAssembly



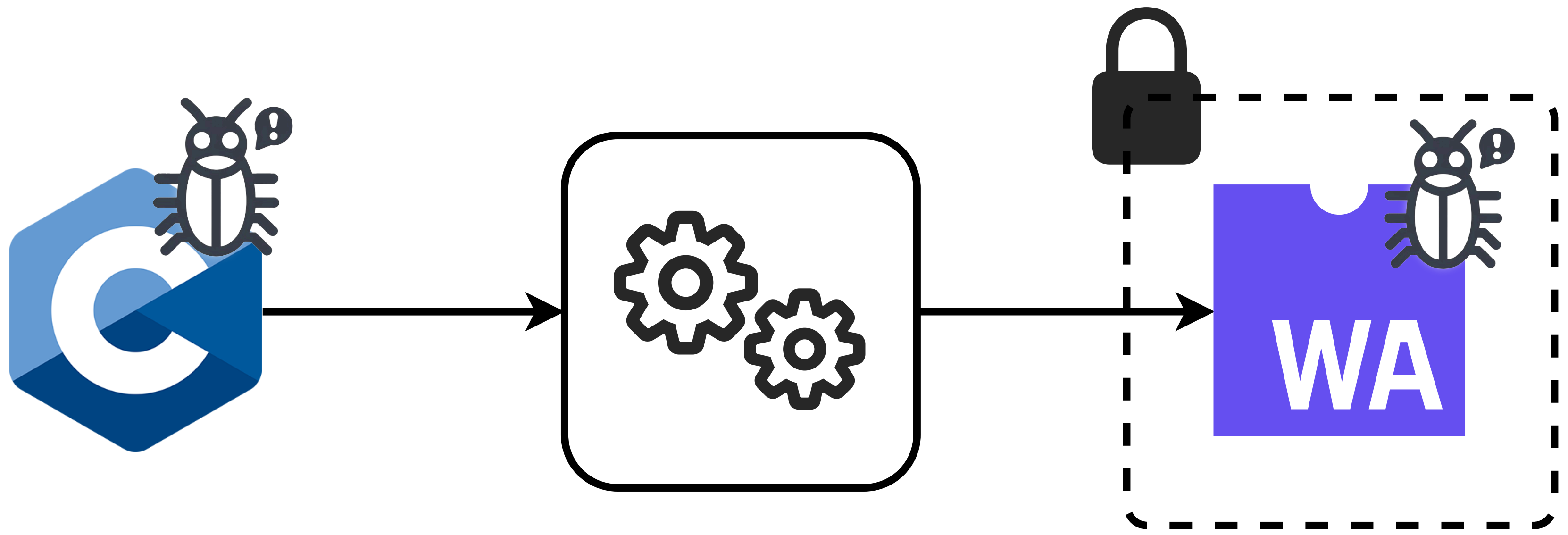
- Versatile compilation target
- Portable and near-native performance
- No direct access to host resources

Security Guarantees of WebAssembly



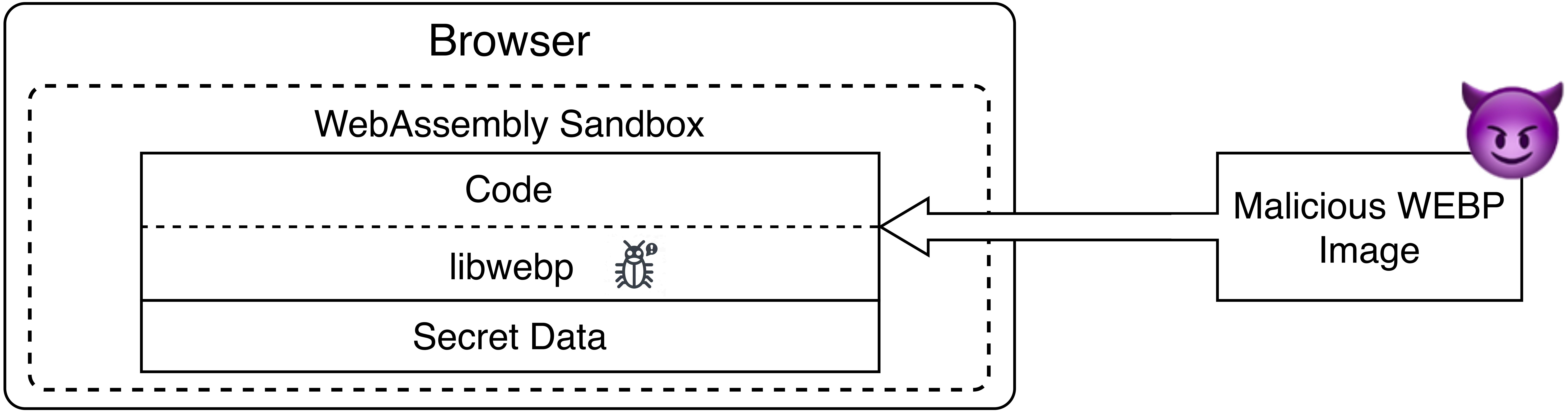
- Provides a sandboxed execution environment

Security Guarantees of WebAssembly



- Provides a sandboxed execution environment
- **No memory safety** guarantees for programs in memory-unsafe languages

Example: A Real-World Vulnerability



- CVE-2023-4863: Heap buffer overflow in libwebp
- Buggy library can be exploited
- WebAssembly does **not** protect against such exploits!

Memory Safety Issues

Google Project Zero

- **72%** of “in the wild” 0-days are memory safety bugs [1]

Microsoft

- **70%** of vulnerabilities in security patches are memory safety violations [2]

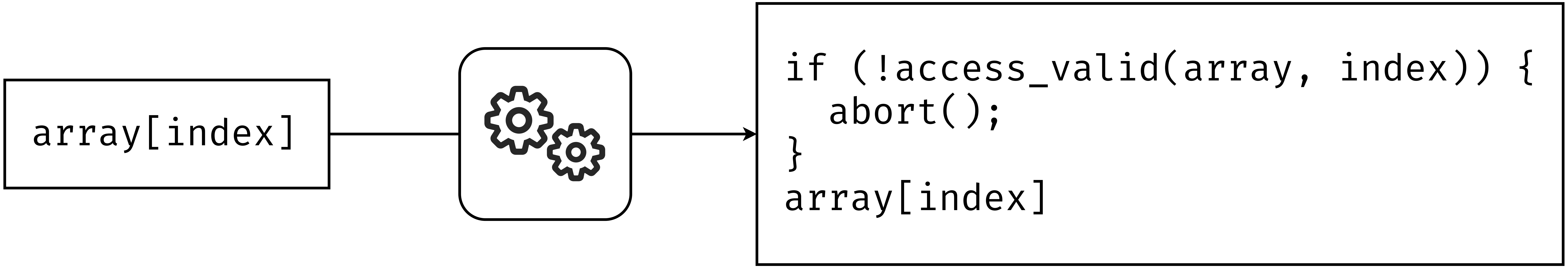
Android

- **24%** of vulnerabilities are memory safety issues (down from 70% in 2019) [3]

[1] Google Project Zero: https://docs.google.com/spreadsheets/d/1lkNJ0uQwbeC1ZTRrxdtuPLCIl7mlUreoKfSIgajnSyY/view
[2] Microsoft: https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code/
[3] Android: https://security.googleblog.com/2024/09/eliminating-memory-safety-vulnerabilities-Android.html

Software-Based Approach

Deterministic Bounds Checking

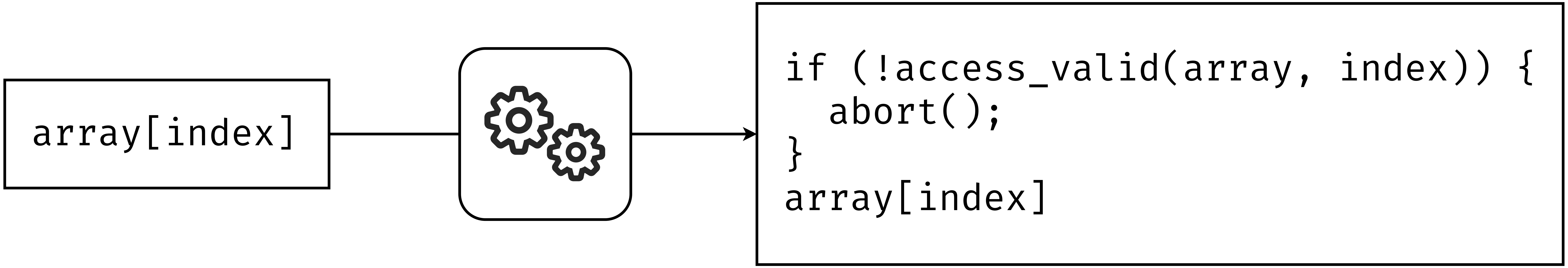


- **Address Sanitizer**: Average slowdown of **73%** [4]

[4] Serebryany, Konstantin, et al. "AddressSanitizer: A fast address sanity checker." 2012 USENIX annual technical conference (USENIX ATC 12). 2012

Software-Based Approach

Deterministic Bounds Checking



- **Address Sanitizer**: Average slowdown of **73%** [4]

Not suitable for production deployment!

[4] Serebryany, Konstantin, et al. "AddressSanitizer: A fast address sanity checker." 2012 USENIX annual technical conference (USENIX ATC 12). 2012

Problem Statement

How can we provide **memory safety** for **WebAssembly** with **low performance** and **memory overheads**?

Design Goals

- **Memory Safety**: spatial and temporal
- **Transparency**: no modification to existing code
- **Portability**: hardware-independent abstraction
- **Security**: WebAssembly modules might be adversarial

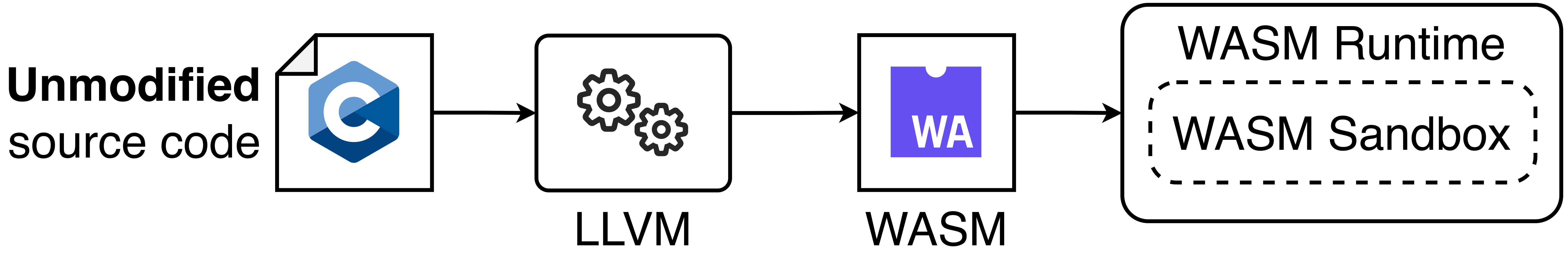
Low Overheads:

- Performance
- Memory Usage
- Sandboxing

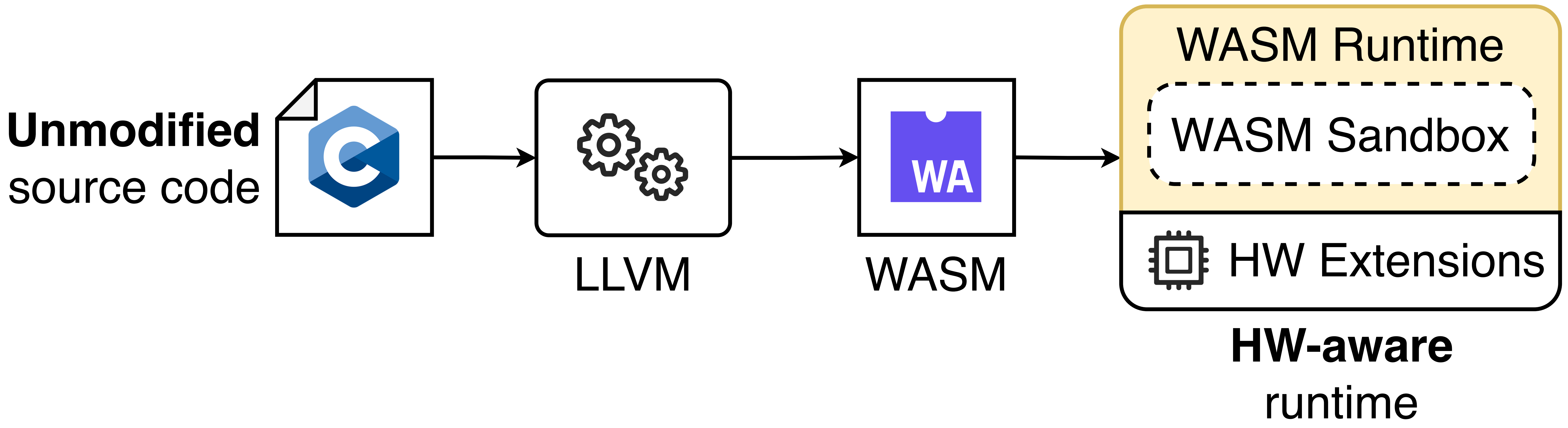
Outline

- Background and Motivation
- **Design**
  - Internal Memory Safety
  - External Memory Safety (Sandboxing)
  - Combining Internal and External Memory Safety
- Implementation
- Evaluation

Key Ideas

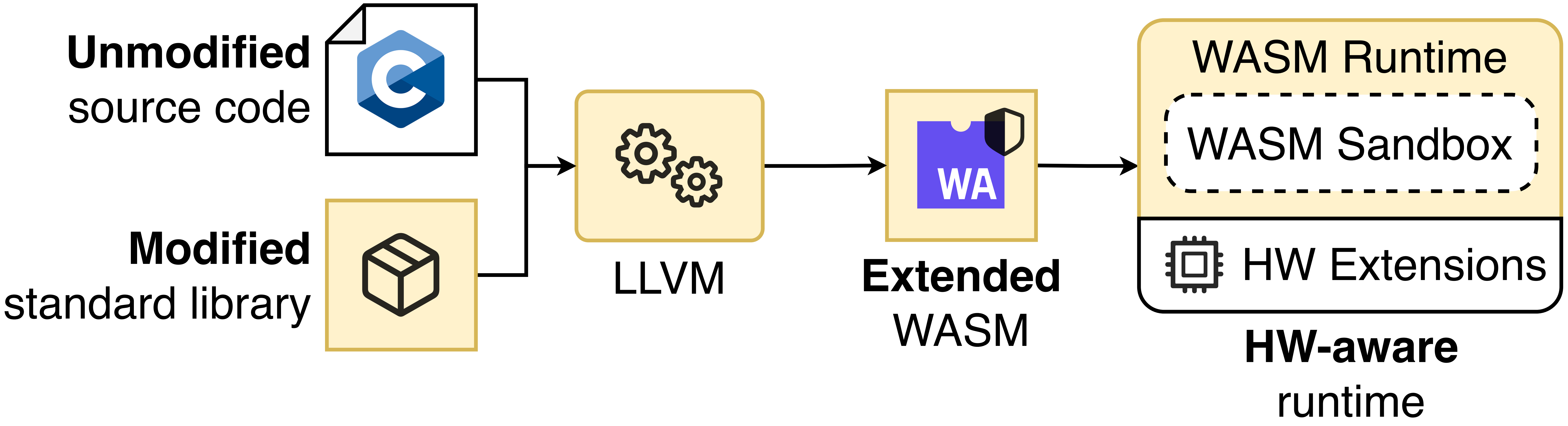

Unmodified
source code
LLVM
WA
WASM
WASM Runtime
WASM Sandbox

Key Ideas



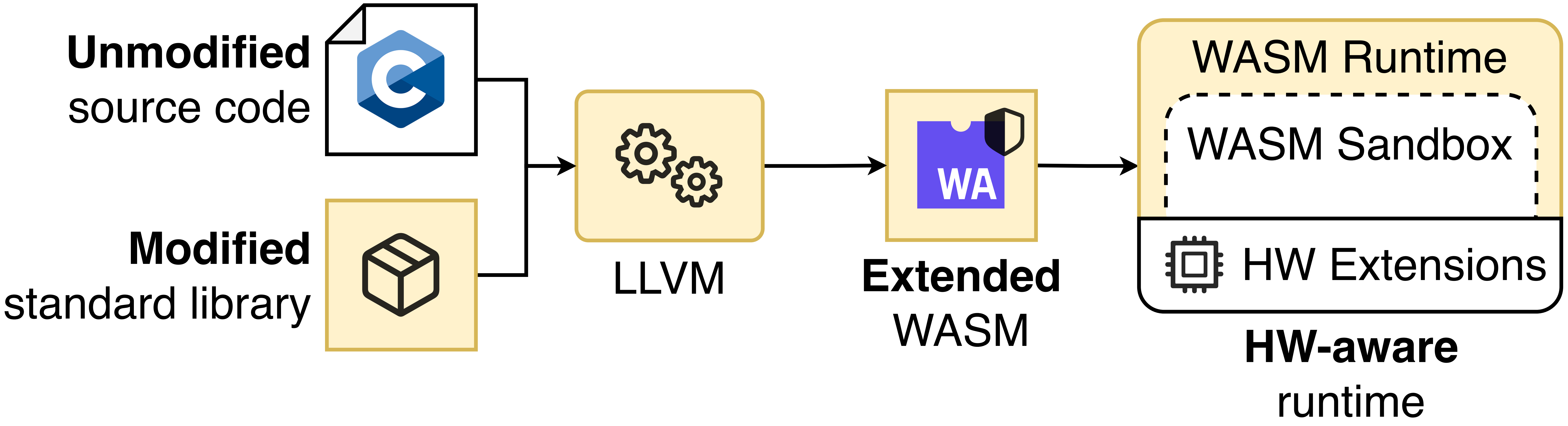
- Hardware-acceleration in CPUs (e.g., Arm MTE)

Key Ideas



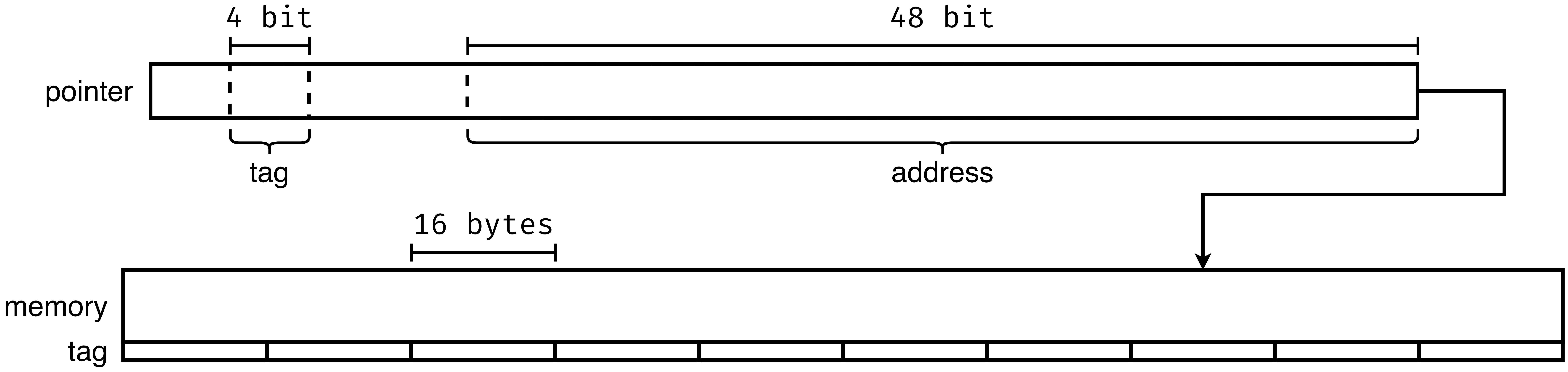
- Hardware-acceleration in CPUs (e.g., Arm MTE)
- Generic abstraction in WebAssembly: tagged pointers and segments

Key Ideas



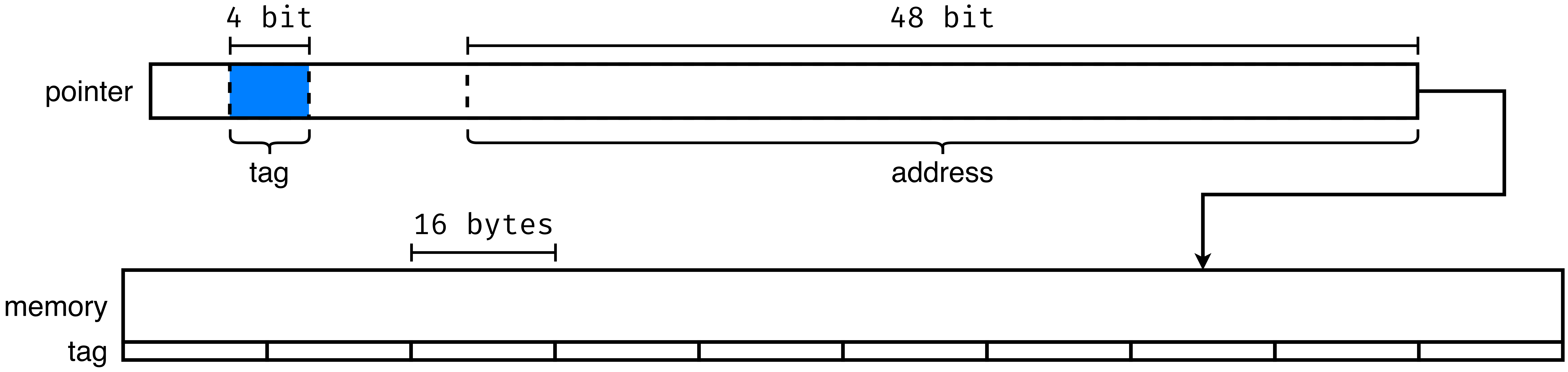
- Hardware-acceleration in CPUs (e.g., Arm MTE)
- Generic abstraction in WebAssembly: tagged pointers and segments

ARM Memory Tagging Extension (MTE)



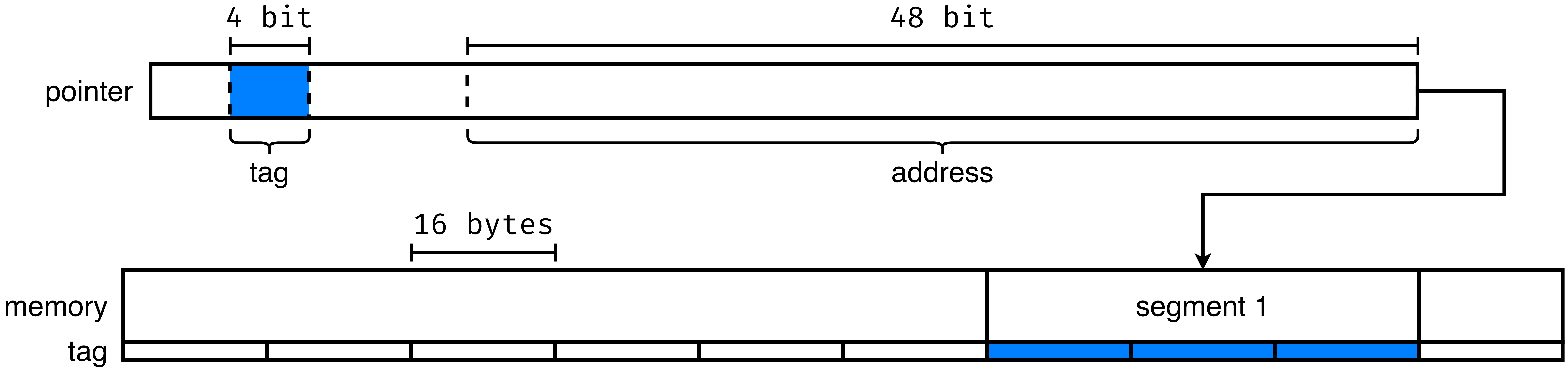
- **4 bit tag** in unused address bits
- **16 byte granularity**
- Tag mismatch is caught by hardware

ARM Memory Tagging Extension (MTE)



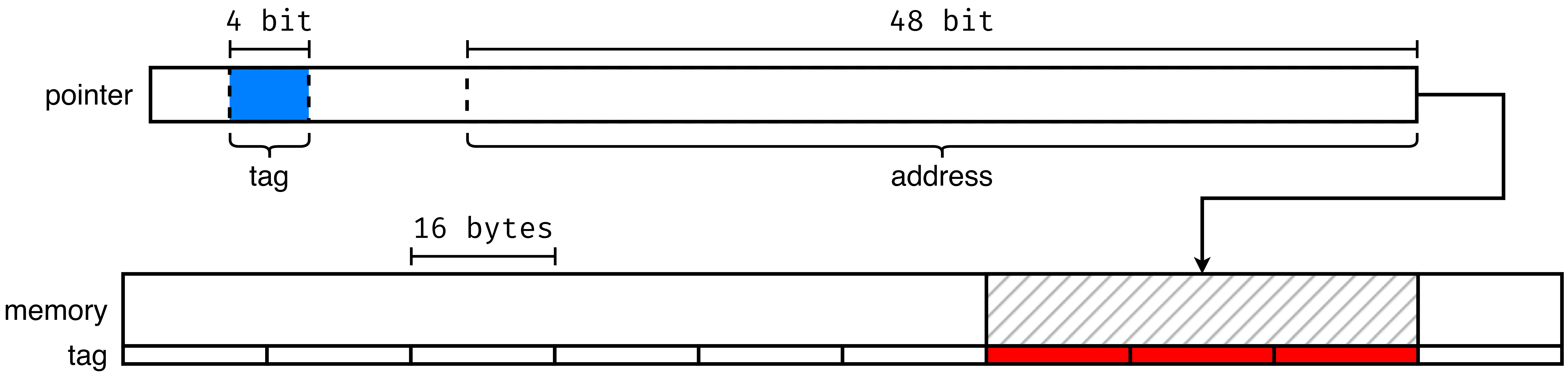
- **4 bit tag** in unused address bits
- **16 byte granularity**
- Tag mismatch is caught by hardware

ARM Memory Tagging Extension (MTE)



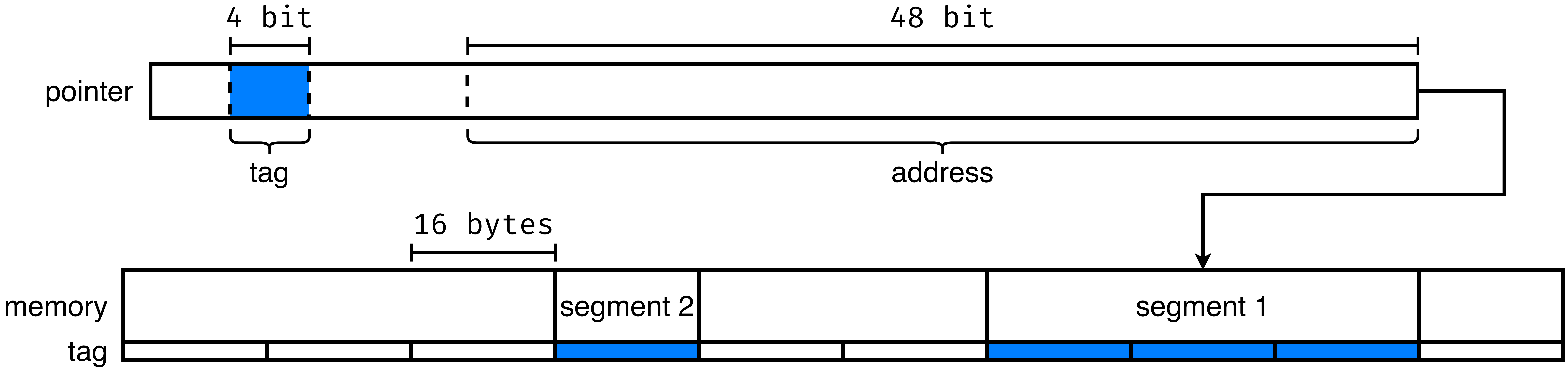
- **4 bit tag** in unused address bits
- **16 byte granularity**
- Tag mismatch is caught by hardware

ARM Memory Tagging Extension (MTE)



- **4 bit tag** in unused address bits
- **16 byte granularity**
- Tag mismatch is caught by hardware

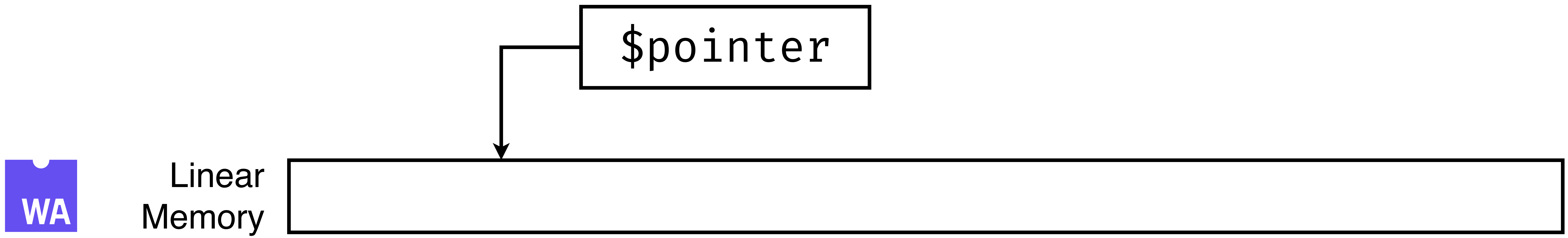
ARM Memory Tagging Extension (MTE)



- Probabilistic Memory Safety
- 16 distinct tags → **tag collisions**

Memory Segments

```
char *pointer = malloc(32);
```

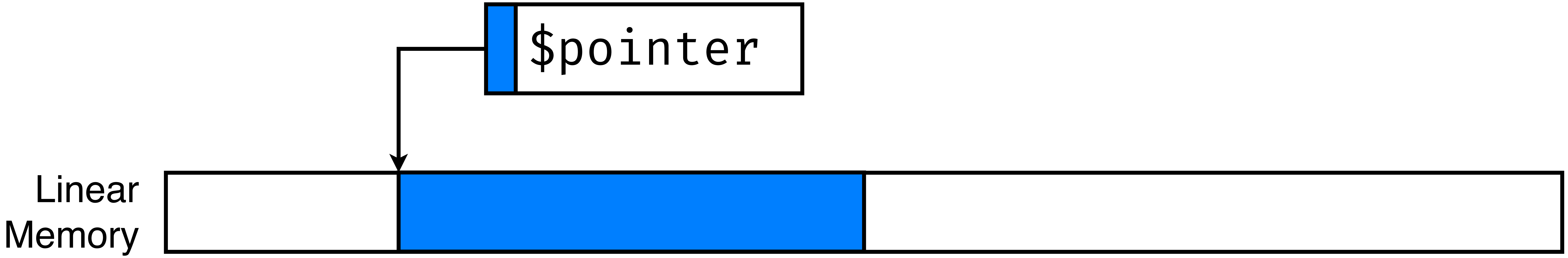


Memory Segments

Memory Segments and Tagged Pointers

```
char *pointer = malloc(32);
```

```
segment.new $ptr $len
```

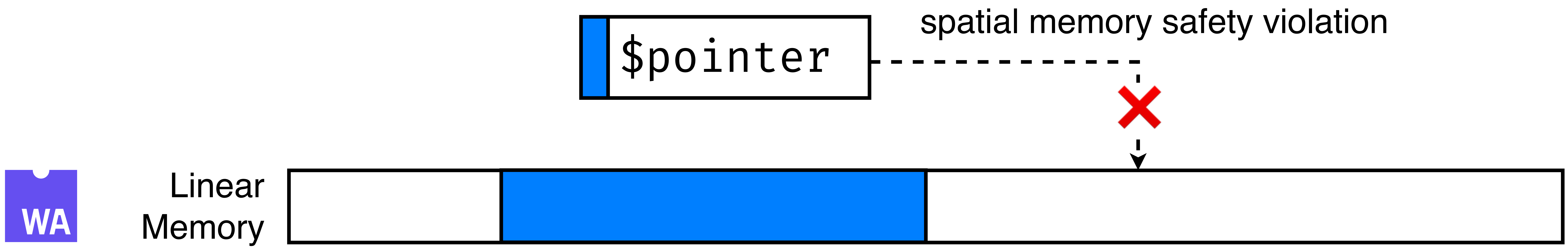


Memory Segments

Spatial Memory Safety Violations

```
char *pointer = malloc(32);
pointer[40];
```

```
segment.new $ptr $len
```

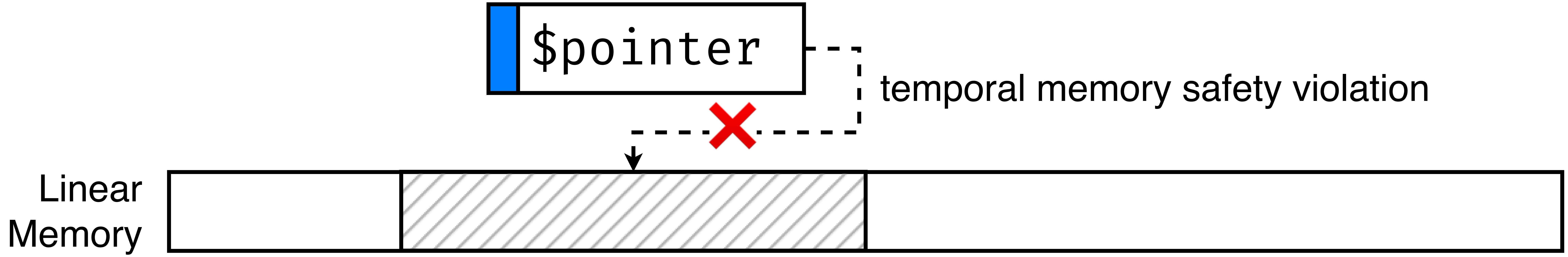


Memory Segments

Temporal Memory Safety Violations

```
char *pointer = malloc(32);
free(pointer);
pointer[24];
```

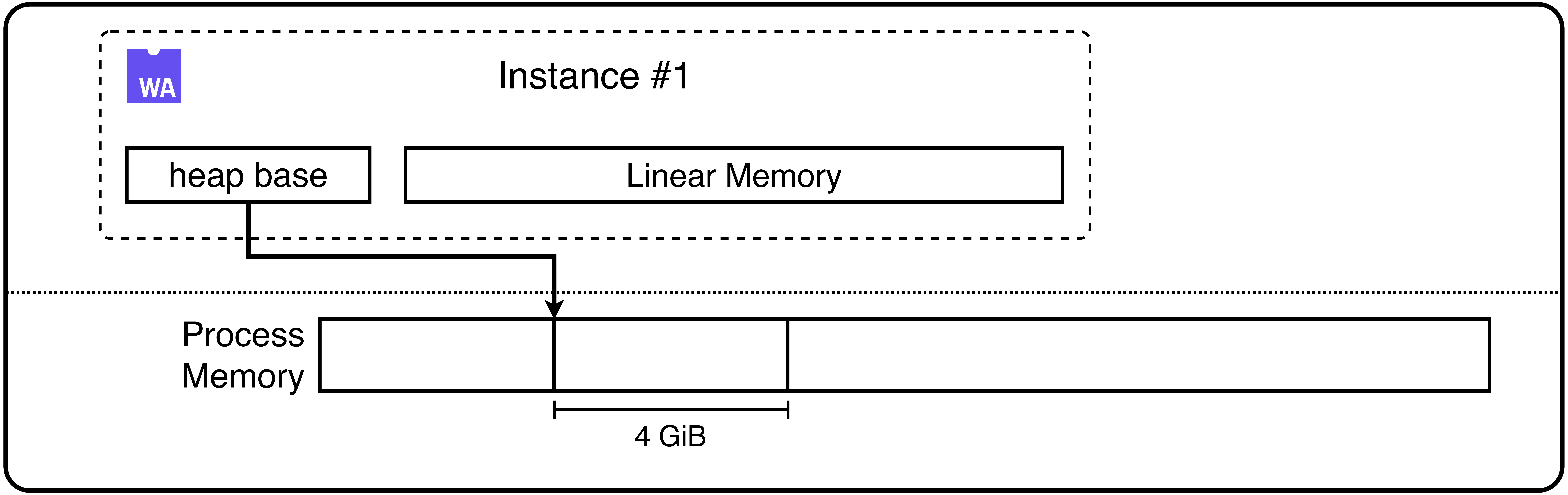
```
segment.free $ptr $len
segment.set_tag $ptr $tag $len
```



Outline

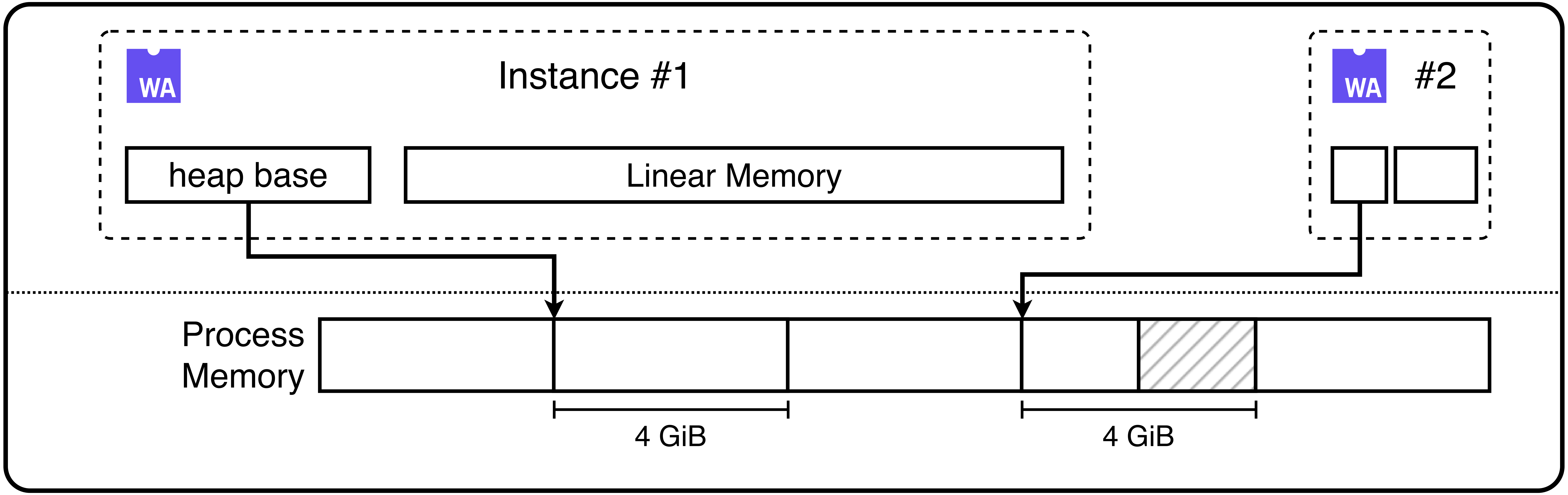
- Background and Motivation
- **Design**
  - Internal Memory Safety
  - **External Memory Safety (Sandboxing)**
  - Combining Internal and External Memory Safety
- Implementation
- Evaluation

External Memory Safety



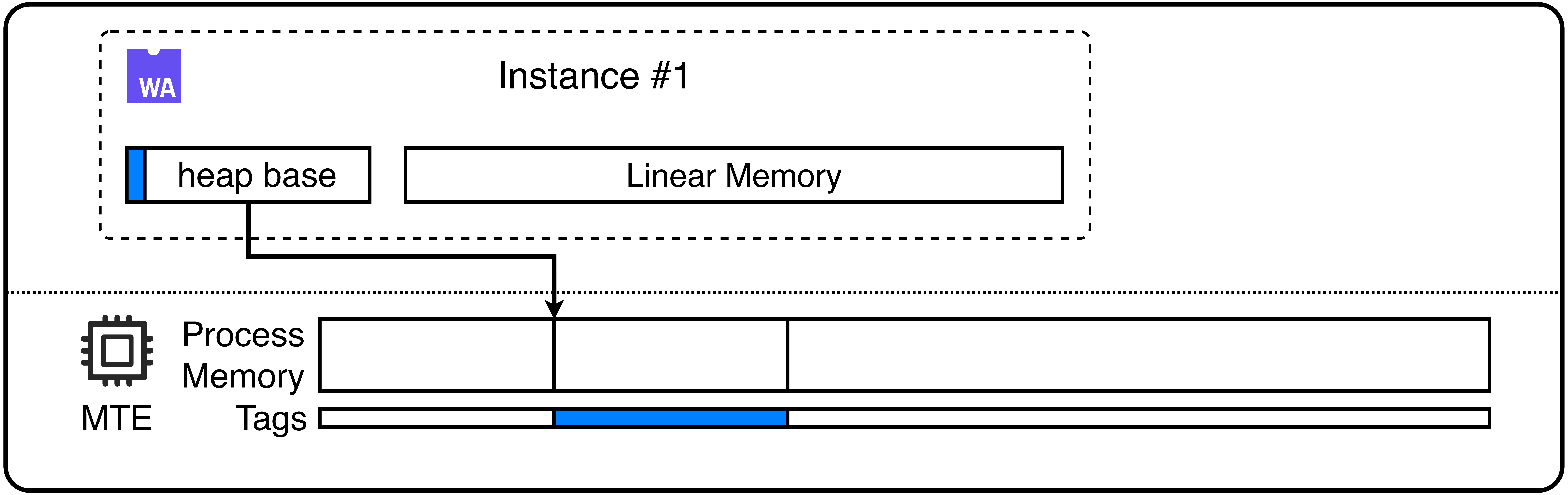
- Sandboxing using **guard pages**
- Allocate 2^{32} = 4 GiB of virtual memory per sandbox
- Only possible for 32-bit WebAssembly

External Memory Safety



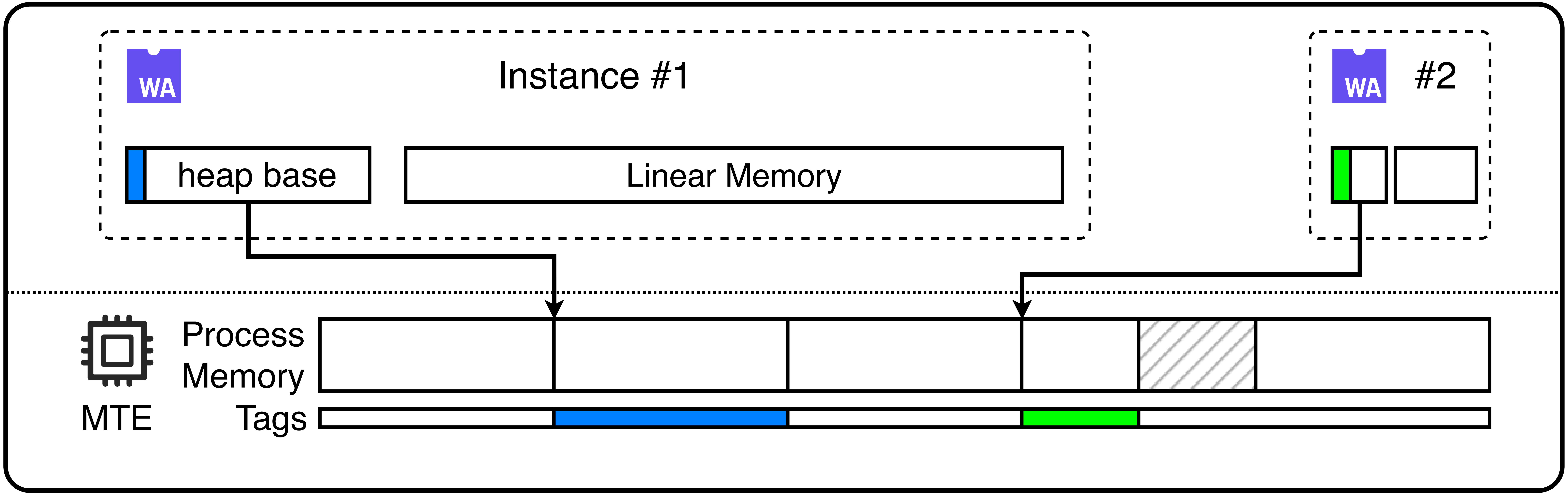
- Sandboxing using **guard pages**
- Allocate 2^{32} = 4 GiB of virtual memory per sandbox
- Only possible for 32-bit WebAssembly

External Memory Safety



- Assign distinct tag for each sandbox
- Perform access relative to tagged base pointer

External Memory Safety

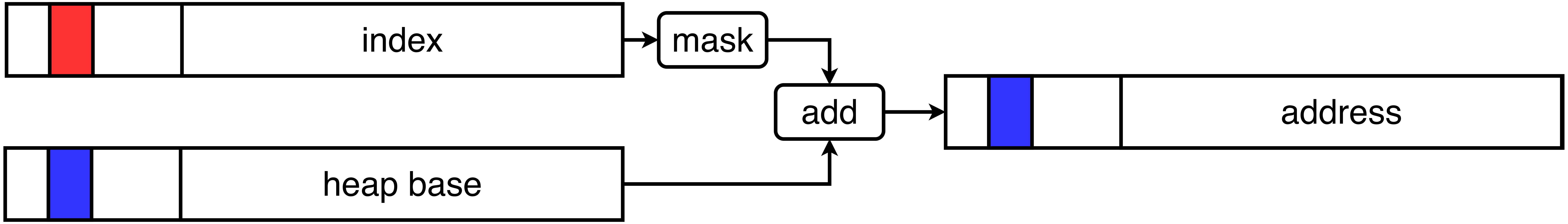


- Assign distinct tag for each sandbox
- Perform access relative to tagged base pointer

Outline

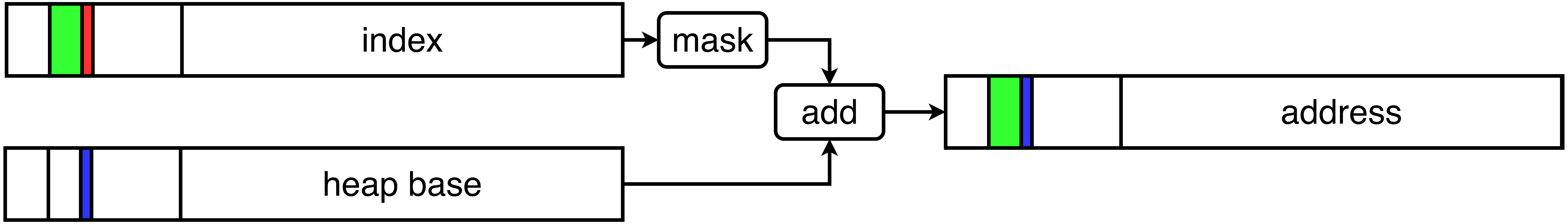
- Background and Motivation
- **Design**
  - Internal Memory Safety
  - External Memory Safety (Sandboxing)
  - **Combining Internal and External Memory Safety**
- Implementation
- Evaluation

Combining Memory Safety and Sandboxing



- Split tag bits
  - Up to four bits for sandboxing
  - Remaining bits for memory safety within the sandbox
- On address translation, mask out runtime-reserved bits

Combining Memory Safety and Sandboxing



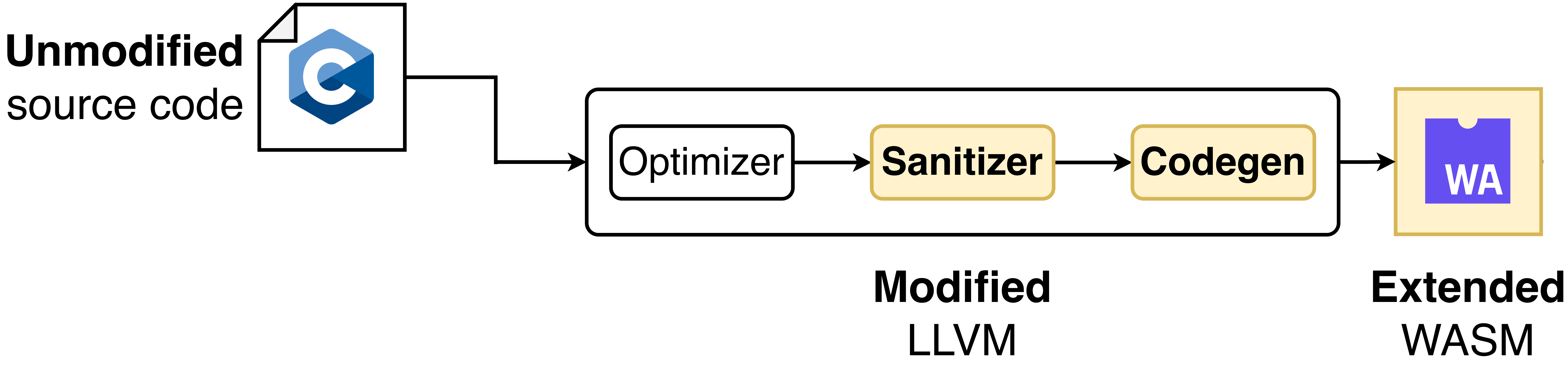
- Split tag bits
  - Up to four bits for sandboxing
  - Remaining bits for memory safety within the sandbox
- On address translation, mask out runtime-reserved bits

Outline

- Background and Motivation
- Design
- **Implementation**
- Evaluation

Implementation

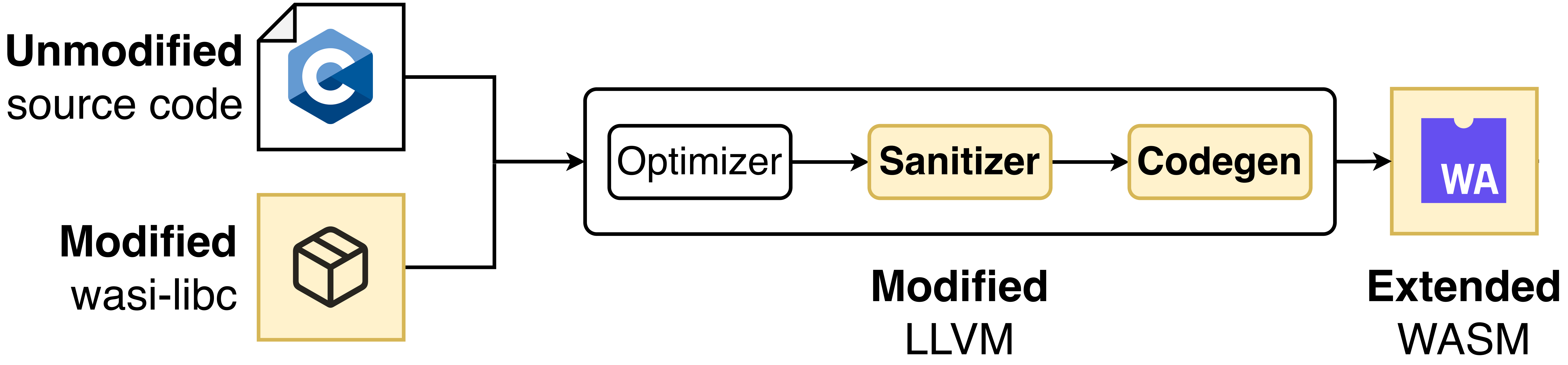
Implementation



Compiler Toolchain

- LLVM 17
- Sanitizer passes

Implementation



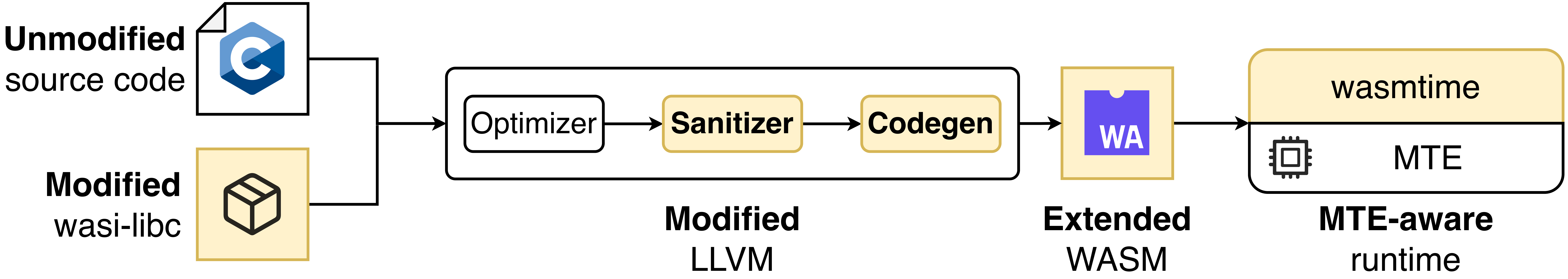
Compiler Toolchain

- LLVM 17
- Sanitizer passes

Libc

- wasi-libc
- 64-bit WASM
- Memory-safe allocator

Implementation



Compiler Toolchain

- LLVM 17
- Sanitizer passes

Libc

- wasi-libc
- 64-bit WASM
- Memory-safe allocator

WASM Runtime

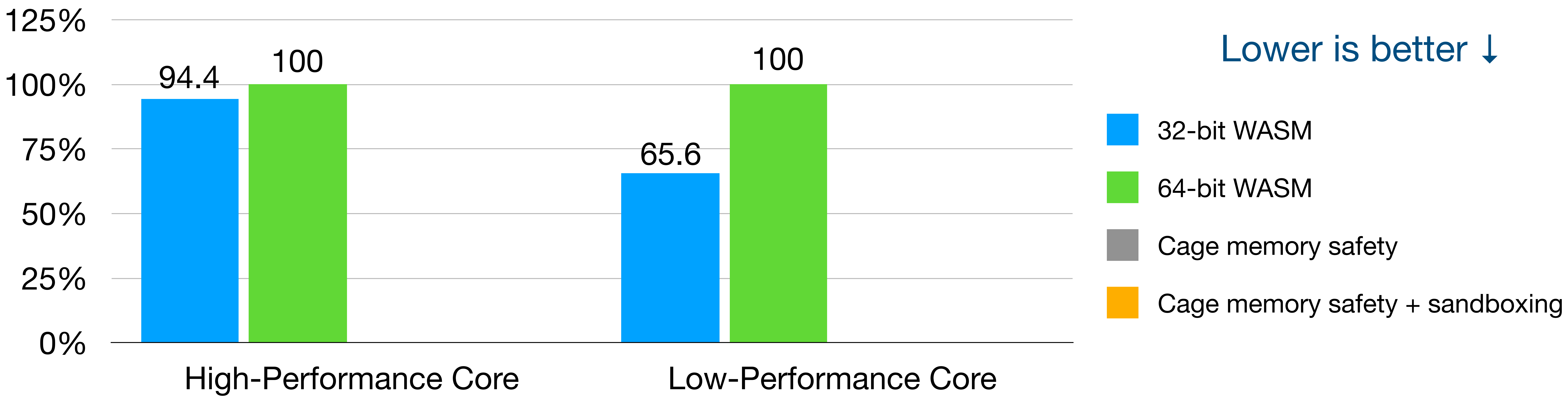
- wasmtime 16
- MTE-based memory safety
- MTE-based sandboxing

Outline

- Background and Motivation
- Design
- Implementation
- **Evaluation**

Runtime Overheads

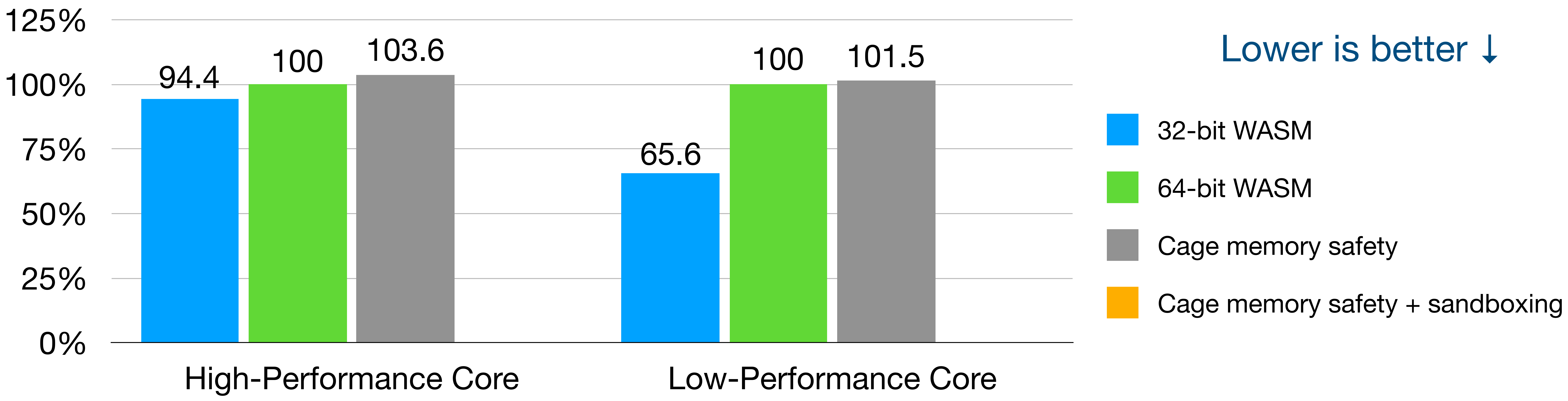
PolyBench/C on Google Pixel 8



Baselines: 32-bit WASM **34.5—5.6% faster** than 64-bit WASM

Runtime Overheads

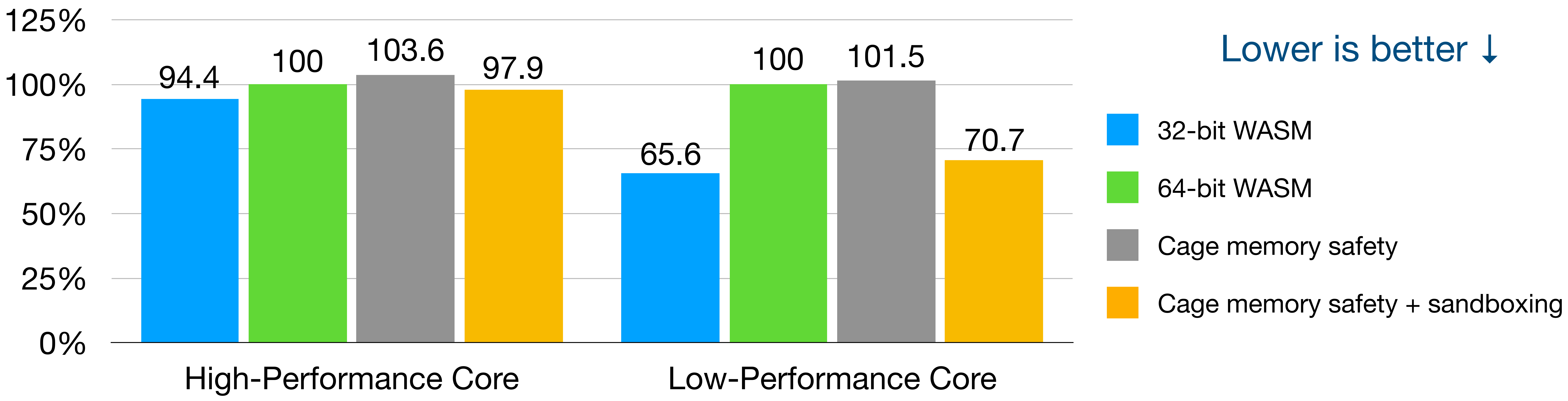
PolyBench/C on Google Pixel 8



Memory Safety: **1.5—3.6%** overhead
Address Sanitizer: Runtime overheads of **> 70%**

Runtime Overheads

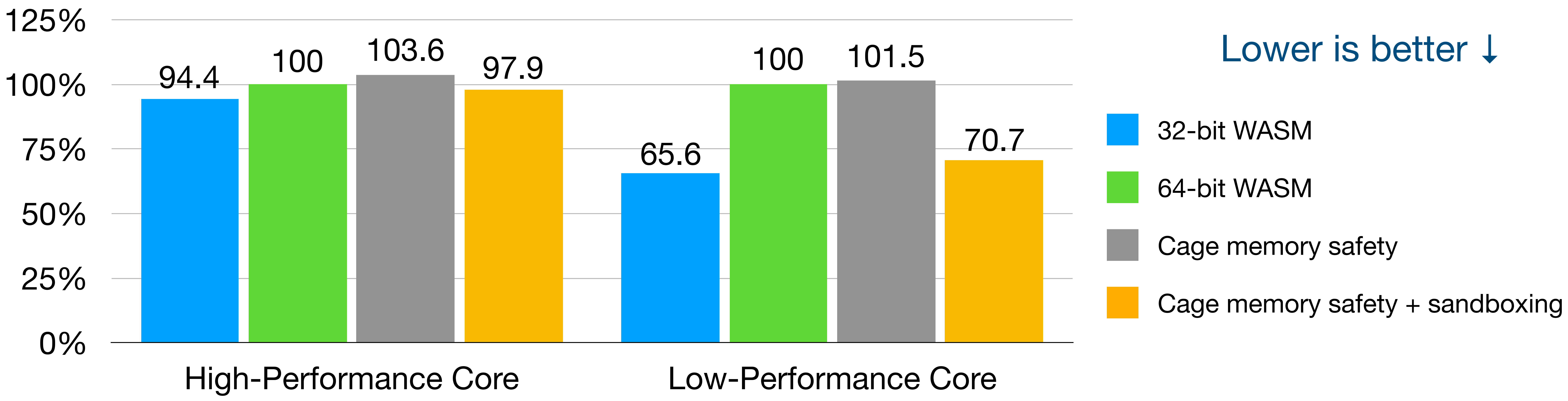
PolyBench/C on Google Pixel 8



Combined with MTE-based sandboxing: 2.1—29.3% speedup

Runtime Overheads

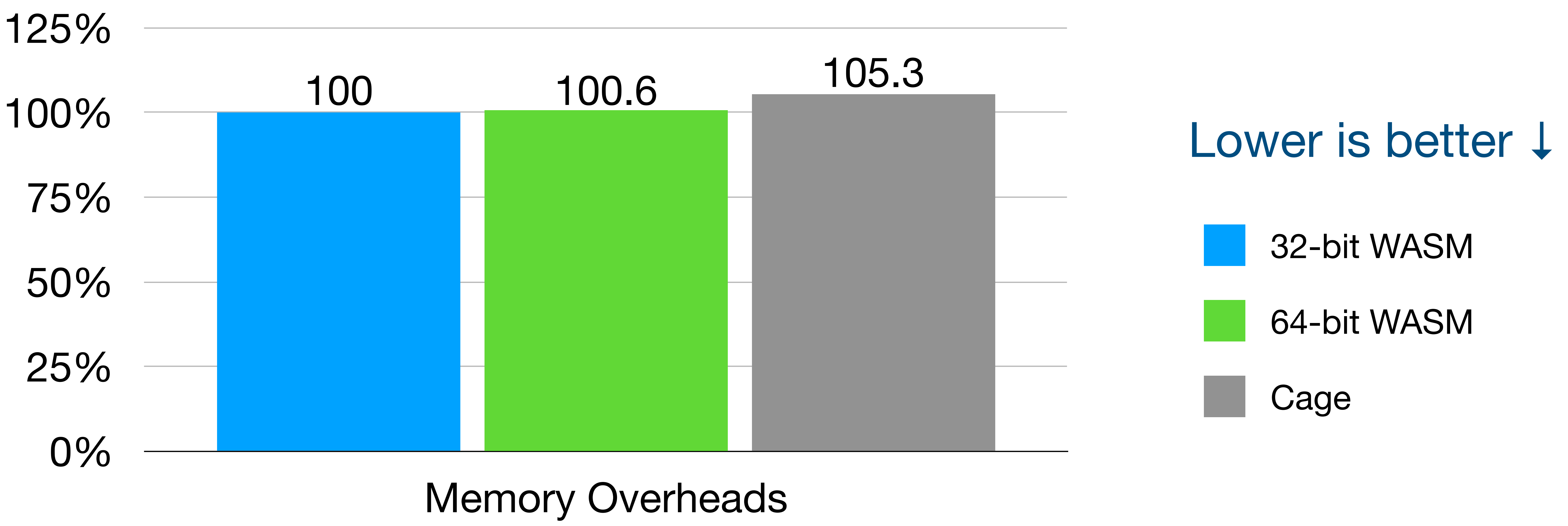
PolyBench/C on Google Pixel 8



Minimal overheads for production deployments, speedups compared to 64-bit WASM!

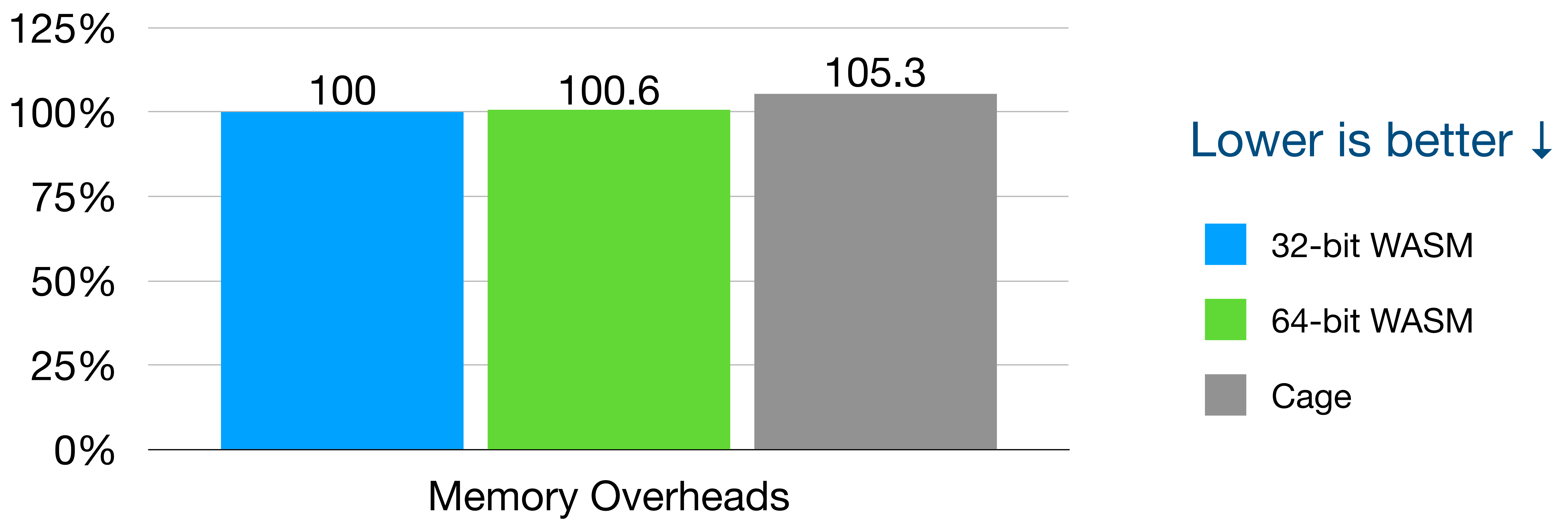
Memory Overheads

PolyBench/C on Google Pixel 8



Memory Overheads

PolyBench/C on Google Pixel 8



Cage introduces minimal memory overheads (~5.3%)
Address sanitizer incurs much larger overheads (2-3x)

Outlook

Growing Importance of Memory Safety

Outlook

Growing Importance of Memory Safety

- Sensitive data is located on mobile devices and in the cloud

Outlook

Growing Importance of Memory Safety

- Sensitive data is located on mobile devices and in the cloud
- Memory-safe languages, testing, and fuzzing are **insufficient**

Outlook

Growing Importance of Memory Safety

- Sensitive data is located on mobile devices and in the cloud
- Memory-safe languages, testing, and fuzzing are **insufficient**
- Memory safety **deployed in production** increases trust

Outlook

Growing Importance of Memory Safety

- Sensitive data is located on mobile devices and in the cloud
- Memory-safe languages, testing, and fuzzing are **insufficient**
- Memory safety **deployed in production** increases trust

Hardware-Assisted Memory Safety

Outlook

Growing Adoption of Memory Safety Extensions

Outlook

Growing Adoption of Memory Safety Extensions

- CPU manufacturers are integrating memory safety extensions
  - Arm MTE, Arm PAC, CHERI, Intel MPK, …

Outlook

Growing Adoption of Memory Safety Extensions

- CPU manufacturers are integrating memory safety extensions
  - Arm MTE, Arm PAC, CHERI, Intel MPK, …
- Widespread deployment in production environments
  - MTE: Google Pixel, Ampere One

Outlook

Growing Adoption of Memory Safety Extensions

- CPU manufacturers are integrating memory safety extensions
  - Arm MTE, Arm PAC, CHERI, Intel MPK, …
- Widespread deployment in production environments
  - MTE: Google Pixel, Ampere One
- Differing tradeoffs
  - Capabilities vs. tagged memory, …

Summary

- **Memory Safety Extension** for 64-bit WebAssembly
- Implementation using **Arm MTE**
- Overheads **<5.6%**, **speedups** when using MTE for sandboxing
- More details, such as formalization, evaluation, and pointer authentication in the paper!

Paper

Source Code