MSWasm: Soundly Enforcing Memory-Safe Execution of Unsafe Code

Alexandra Michael Anitha Gollamudi Jay Bosamiya

Aidan Denlinger Evan Johnson Craig Disselkoen

Conrad Watt Bryan Parno Marco Patrignani

Marco Vassena Deian Stefan

University of Massachusetts Lowell

UNIVERSITY OF CAMBRIDGE

CMU

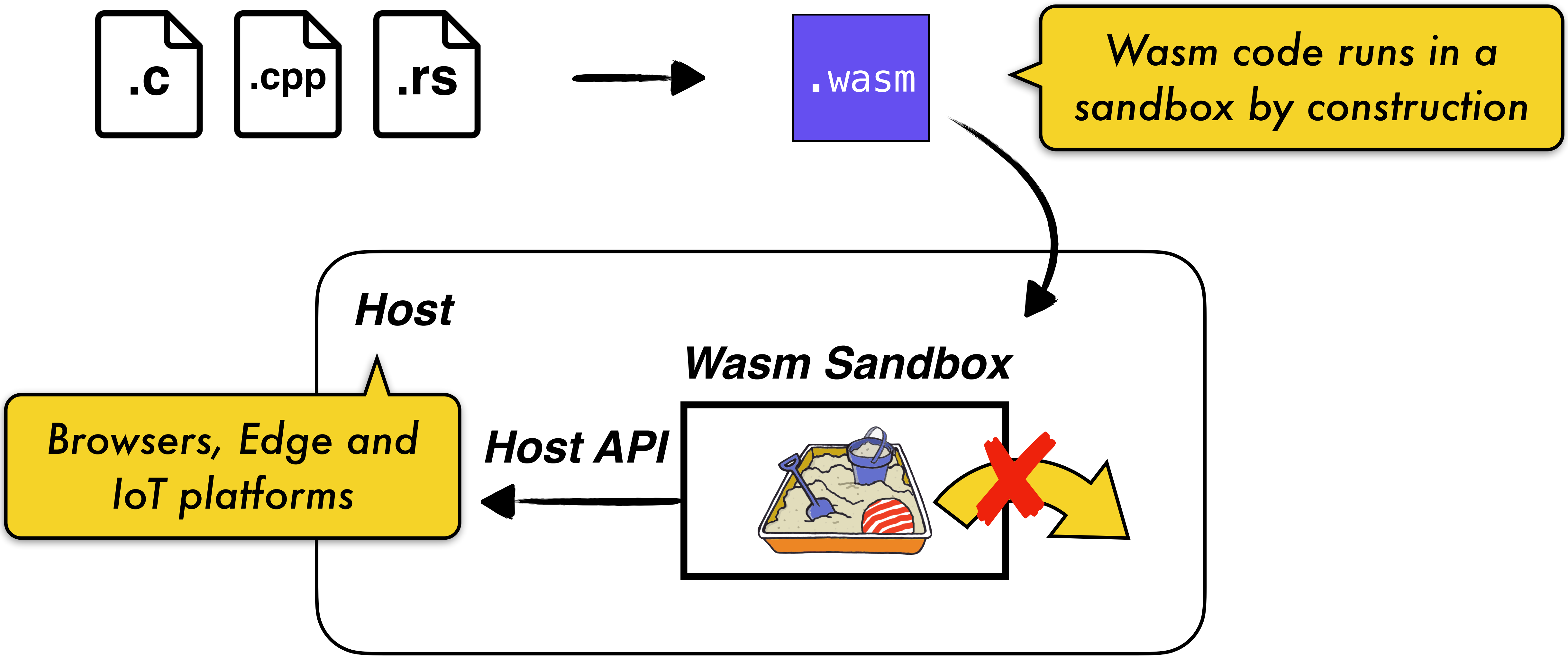
UNIVERSITÀ DI TRENTO

Utrecht University

UC San Diego

WebAssembly

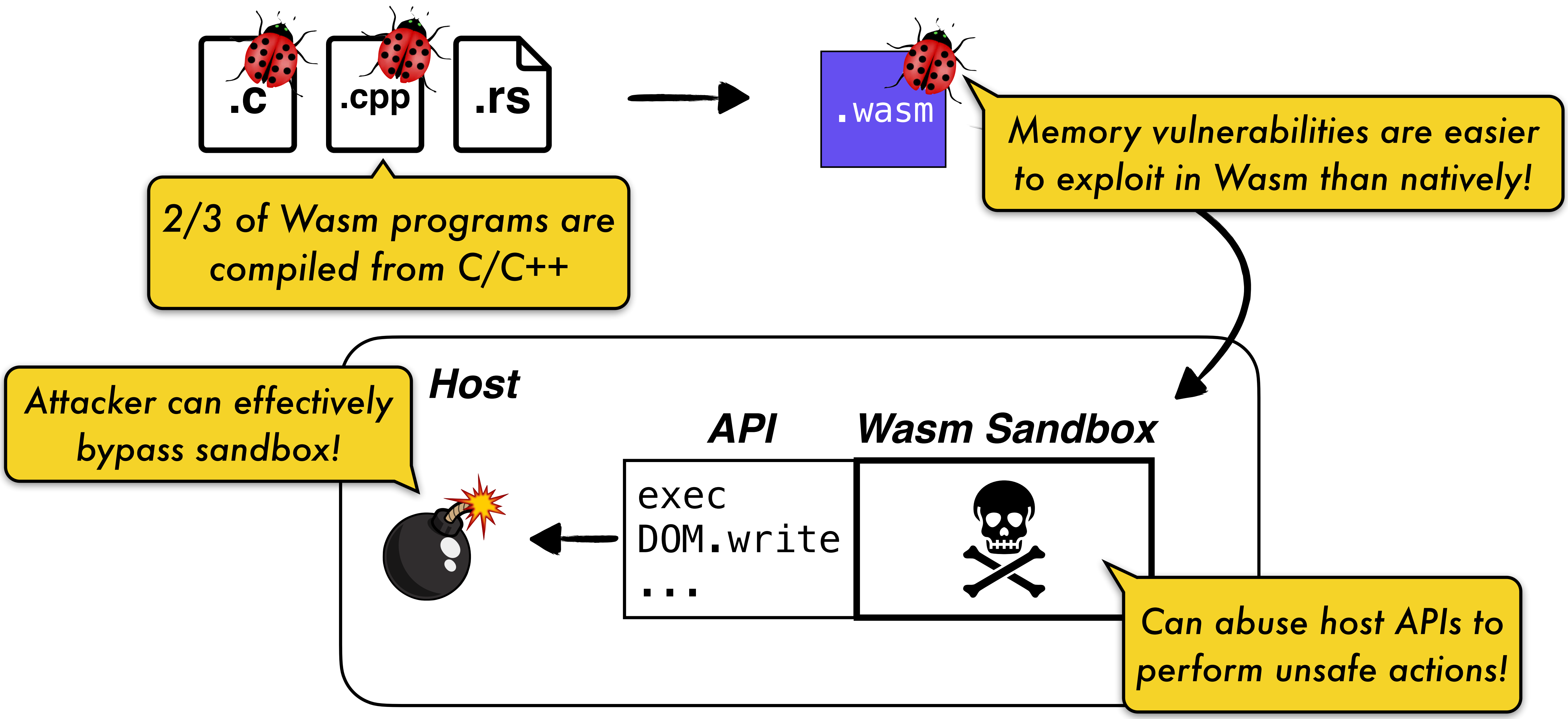
New bytecode language designed to ***run native applications safely***



Wasm programs ***cannot read or corrupt the host's memory!***

Little protection within sandbox!

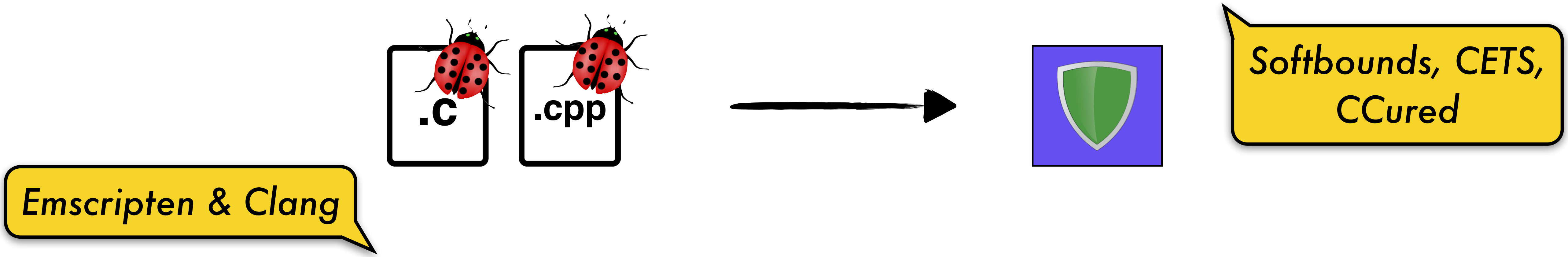
Unsafe programs *remain unsafe when compiled to Wasm*



For example, ***buffer overflows*** *can be turned into* ***XSS attacks*** *[Lehmann et al. 2020]*

Existing Solutions

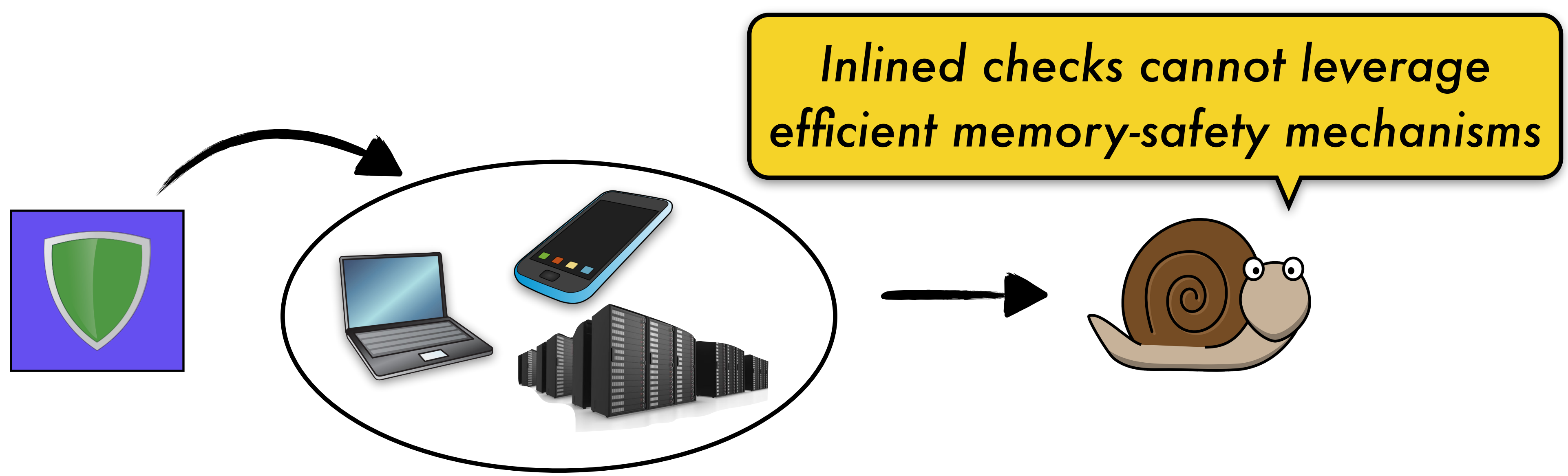
Insert memory-safety checks during compilation:



Industrial compilers do not and they ***should not****!*

No robustness:

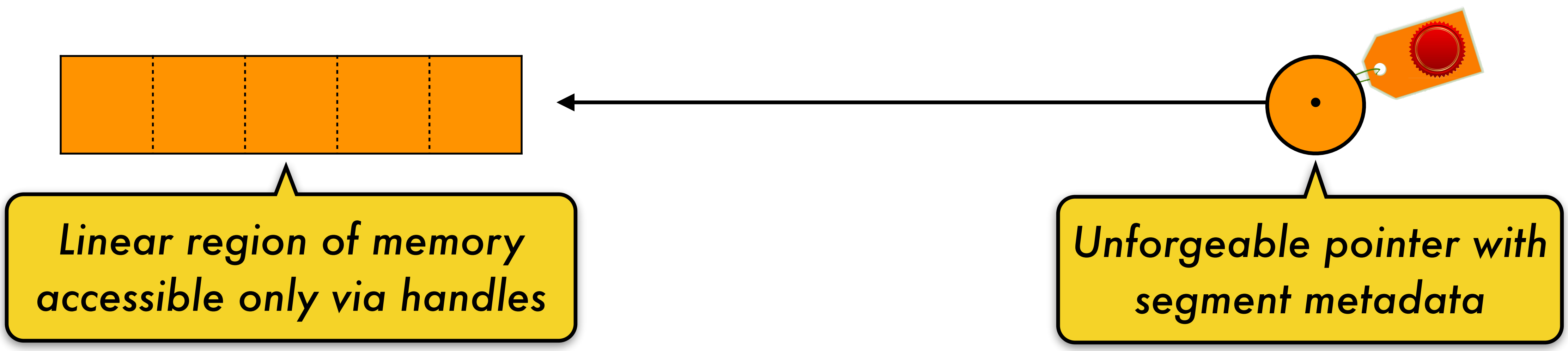
Performance:



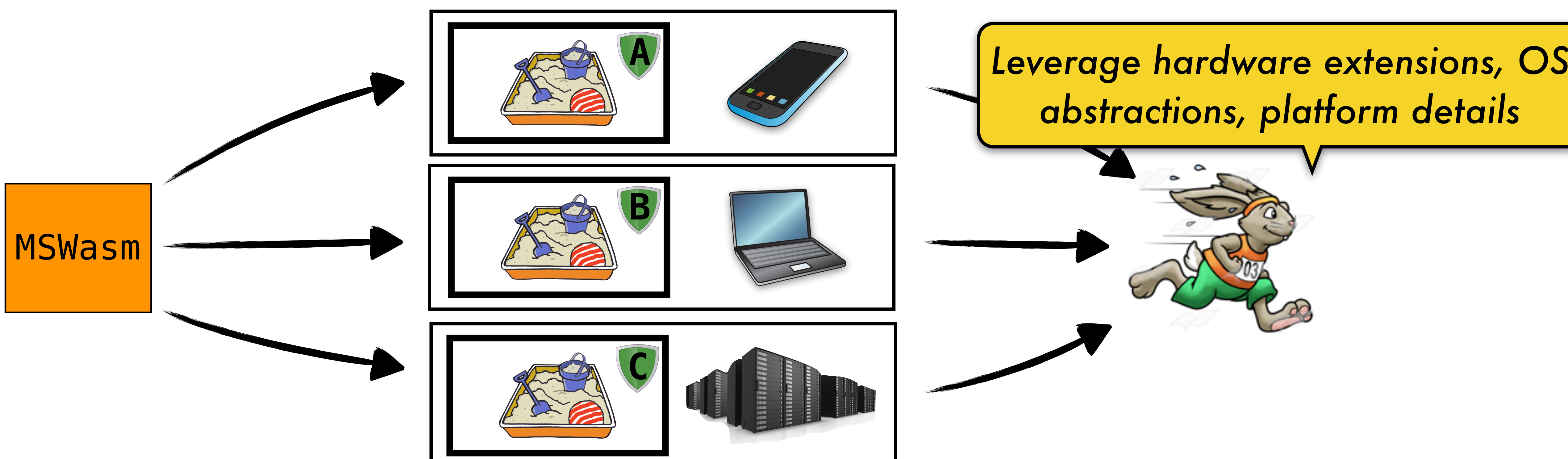
Our Solution: MSWasm

MSWasm extends Wasm with ***memory-safety language abstractions***

Segments **Handles**

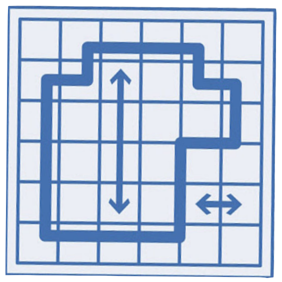


MSWasm backends can ***enforce memory safety robustly & efficiently****!*



Contributions of this work

MSWasm Formal Specification



Well-typed MSWasm programs are robustly memory safe

Color-based Memory Safety

Language- and mechanism-independent definition

Sound C-to-MSWasm Compilation

Memory-safe execution of unsafe code

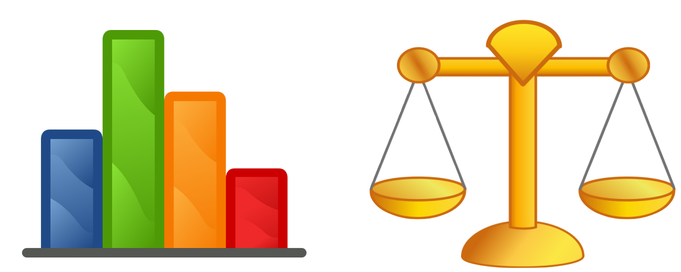
4 MSWasm Compilers

GraalVM

arm Morello

Easy to support different enforcement mechanisms

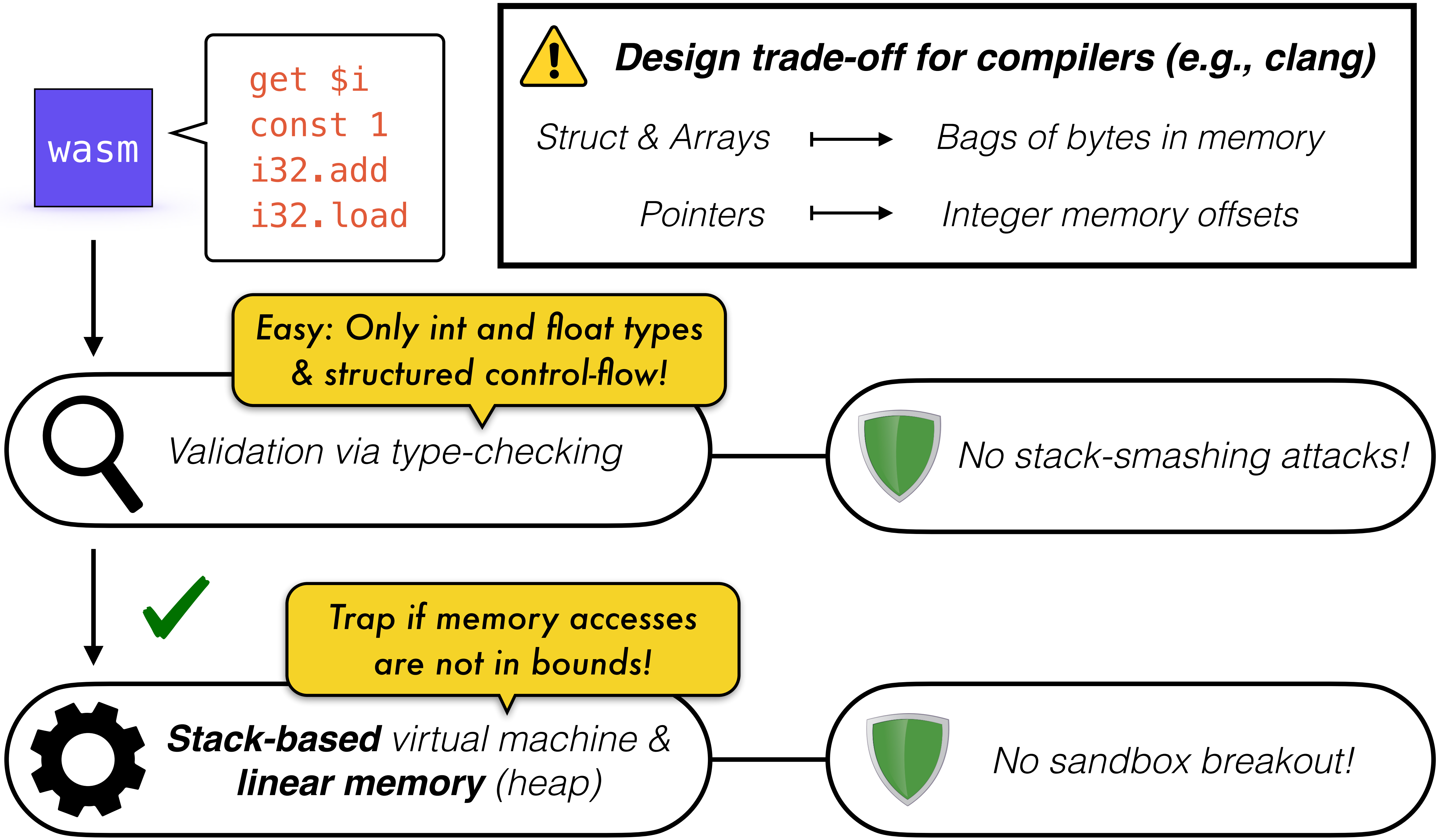
Evaluation on PolyBenchC



General design enables performance-security tradeoffs

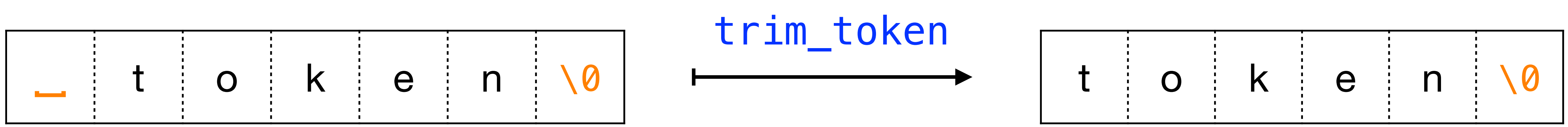
Wasm Basics

Low-level bytecode designed as a ***safe compilation target***



Sandboxing without Memory Safety

Vulnerable function trim_token adapted from libpng 1.6.37:




```c
char *trim_token(char *token) {

  char *trimmed = malloc(1024);

  // Scan token and skip leading whitespace
  // First non-whitespace char and index:
  char next = ...
  int i = ...

  // Copy the rest one char at the time
  for (j = 0; next != \0; j++)
     trimmed[j] = next;
     next = token[++i];

  ...
  return trimmed;
}
```

Possible buffer overflow!

To exploit the vulnerability, call trim_token on a ***string longer than 1024 char after trimming****!*

The vulnerability persists across compilation to Wasm:

C	***Emscripten/Clang***	***Wasm***

```c
trimmed[j] = next;
```

⟼

```wasm
get $trimmed
get $j
i32.add
get $next
i32.store
```

Buffer must be laid out in linear memory in Wasm

Compute the i32 memory address of trimmed[j]

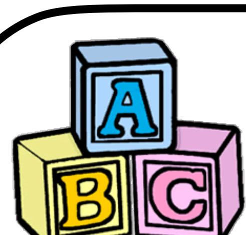
Succeed as long as address is in linear memory!

Vulnerable code cannot break out of the sandbox, ***but****:*

Much easier than natively!

It can ***corrupt*** *and* ***steal sensitive data*** *within sandbox*

Read-only memory & ASLR



Wasm lacks native memory abstractions and protections

MSWasm Design

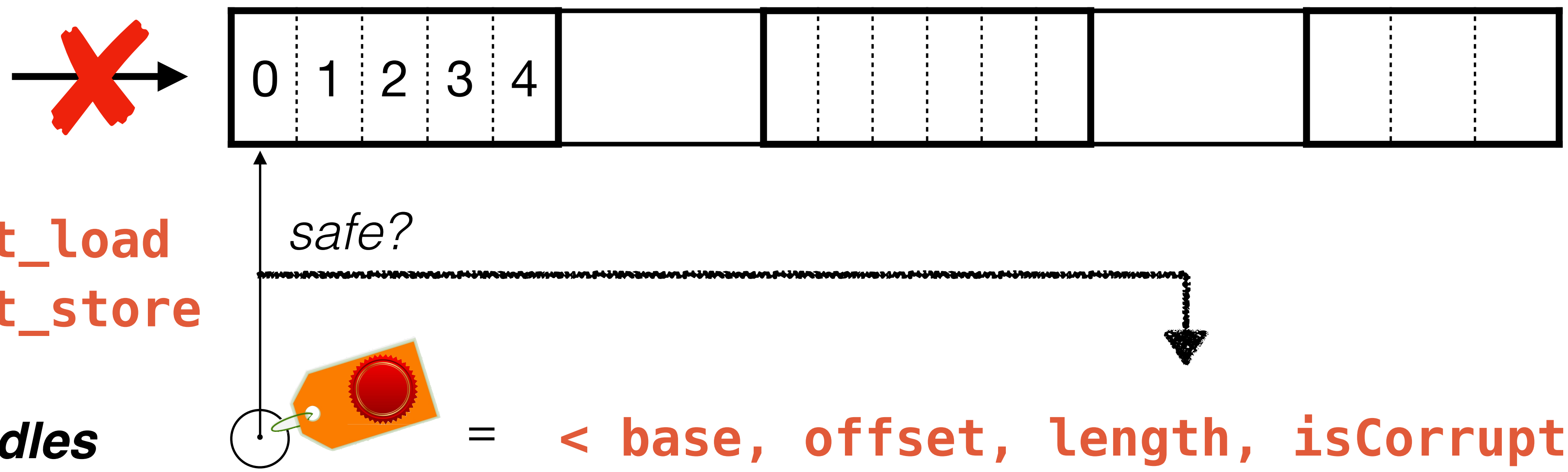
MSWasm provides ***abstractions to enforce memory safety***

New types, values, instructions

Individual Segment

Segment Memory

`load`
`store`



`segment_load`
`segment_store`

Handles = **`< base, offset, length, isCorrupted, id >`**

Spatial Safety

Handle Integrity

Temporal Safety

Other new instructions:

`segment_alloc`

`segment_free`

`handle.add`

`slice` *Shrink portion of segment that a handle can access*

Pointer arithmetic never traps

Emit slice for intra-object memory safety (eg structs)

Enforcing memory safety via compilation

*Compilers can **eliminate latent memory vulnerabilities** by targeting MSWasm*

C-toMSWasm Compiler

```
char *trimmed = malloc(1024);
...
for (j = 0; next != \0; j++)
   trimmed[j] = next;
   next = token[++i];
```

```
const 1024
segment_alloc
set $trimmed
...
get $trimmed
get $j
handle.add
get $next
i32.segment_store
...
```

Allocate 1024-byte segment & store handle in var $trimmed

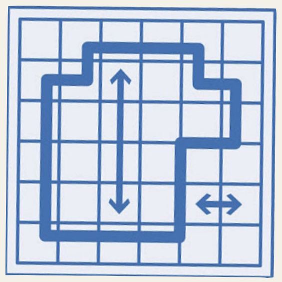
Increment offset of $trimmed & write $next in the segment

Prevent buffer overflow!

*If trimmed is incremented past its bound, segment_store **traps***

This Talk

MSWasm Design



In the paper: type system & operational semantics

Color-based Memory Safety

Sound C-to-MSWasm Compilation

4 MSWasm Compilers

GraalVM™

arm Morello

Evaluation on PolyBenchC

This Talk

MSWasm Design

Color-based Memory Safety

Reason about memory safety & show that MSWasm specs are sound!

Formal Results

4 MSWasm Compilers

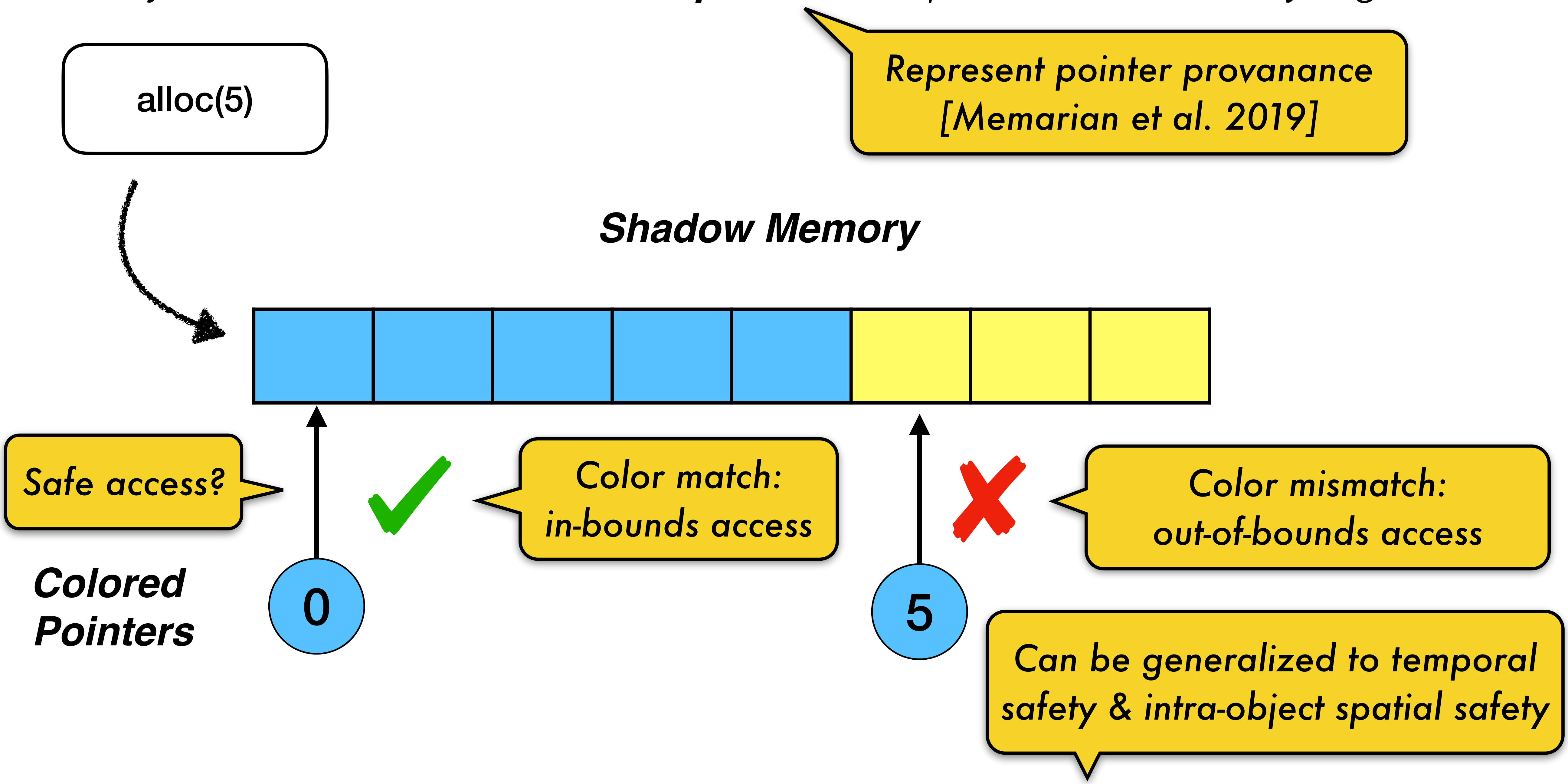
GraalVM

arm Morello

Evaluation on PolyBenchC

Reasoning about Memory Safety using Colors

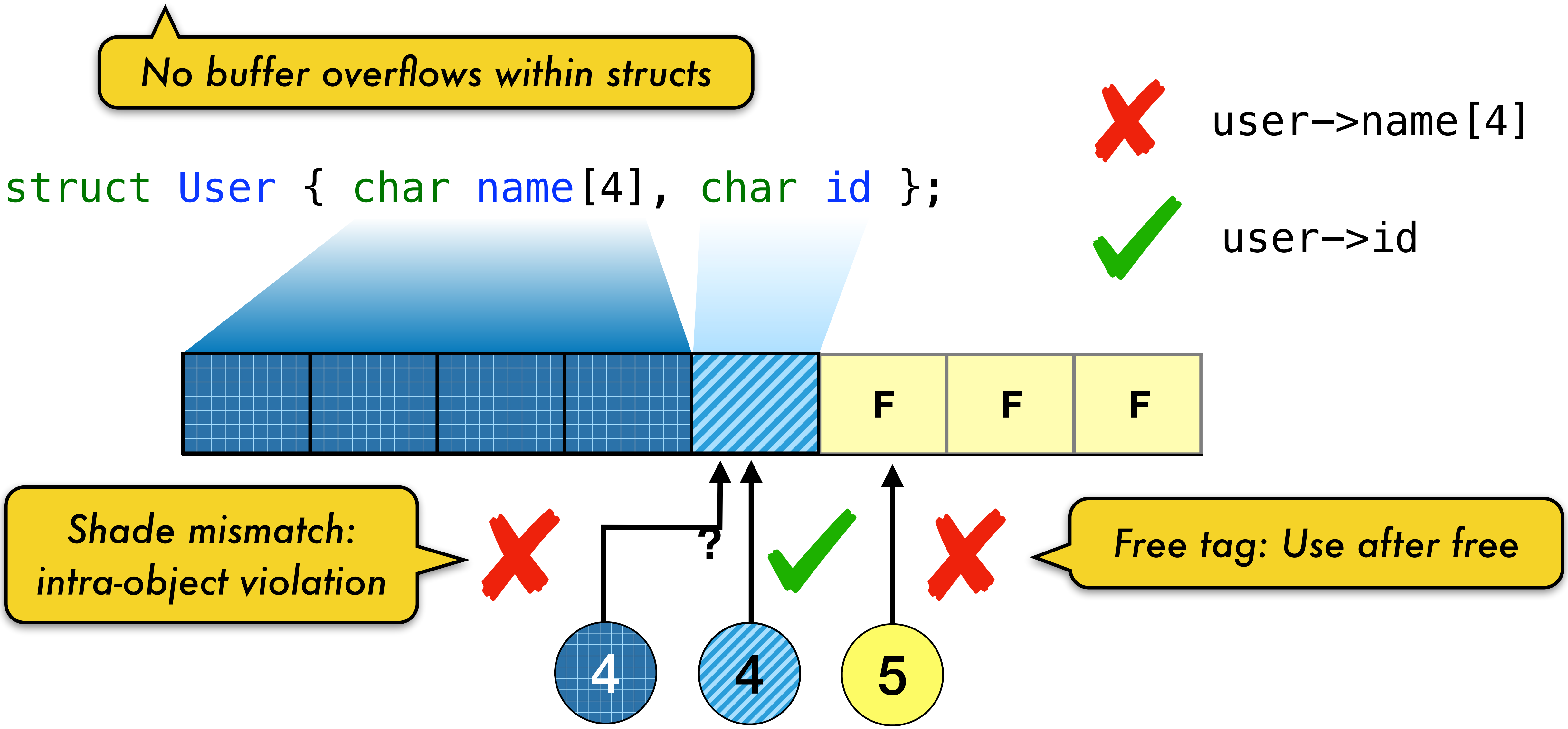
*Memory allocations associate a **unique color** to pointer and memory region:*



*This is sufficient to detect **spatial memory-safety violations***

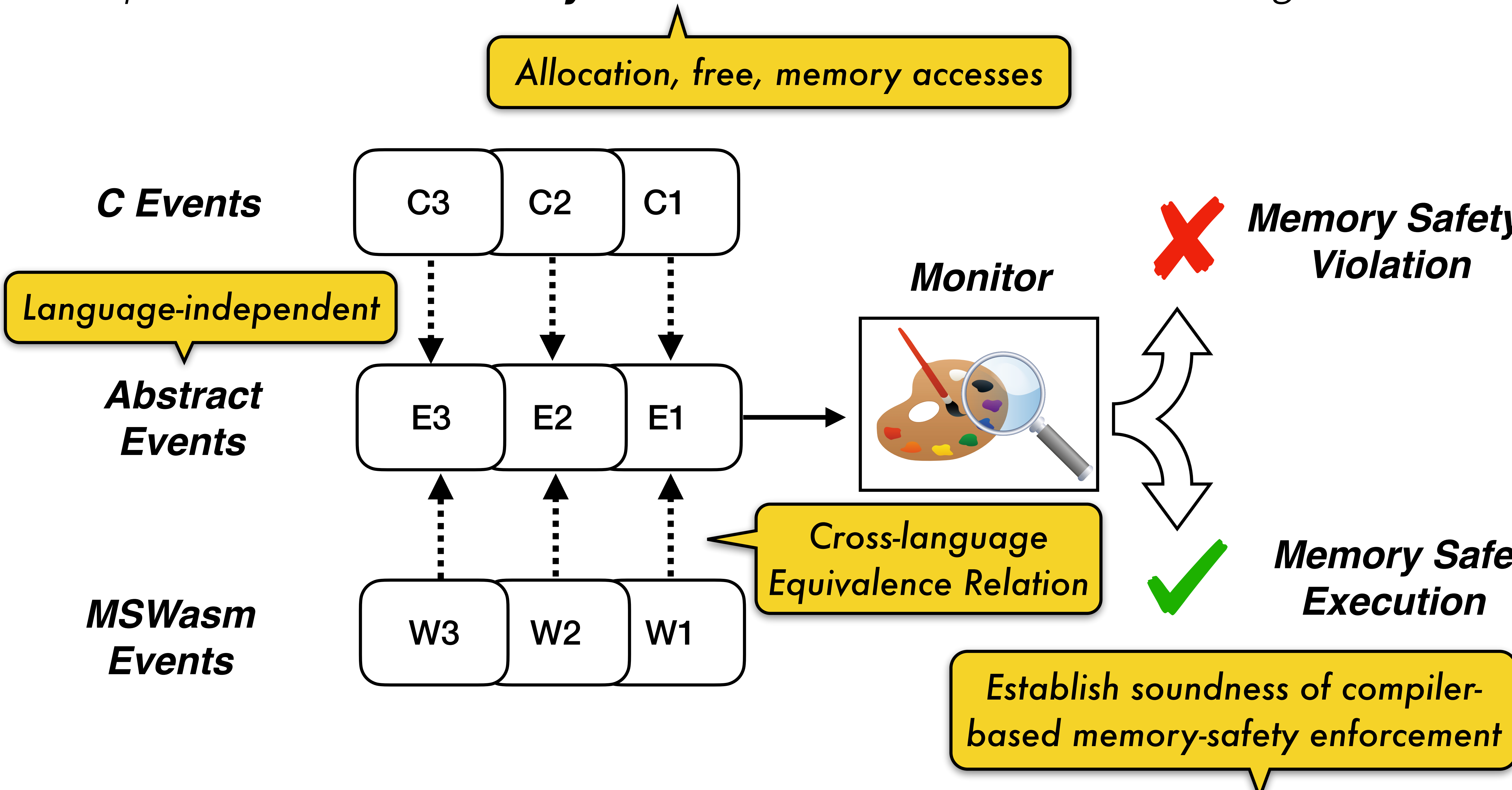
Color-Based Memory Safety

Spatial Safety	*Location and pointer **colors must match***
Temporal Safety	*Add **tags** to mark **free** memory locations*
Intra-object Safety	*Decorate pointers and locations with **shades***



Color-Based Memory-Safety Monitor

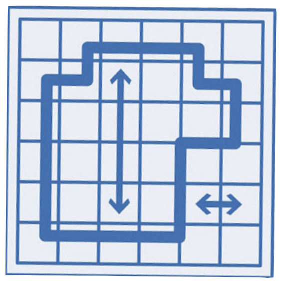
*Inspect a trace of **memory events** and **detect violations** using colors:*



*The monitor allows reasoning about **memory safety for different languages***

This Talk

MSWasm Design



Color-based Memory Safety

Sound C-to-MSWasm Compilation

4 MSWasm Compilers

GraalVM™

arm Morello

Evaluation on PolyBenchC

This Talk

MSWasm Design

Color-based Memory Safety

Sound C-to-MSWasm Compilation

4 MSWasm Compilers

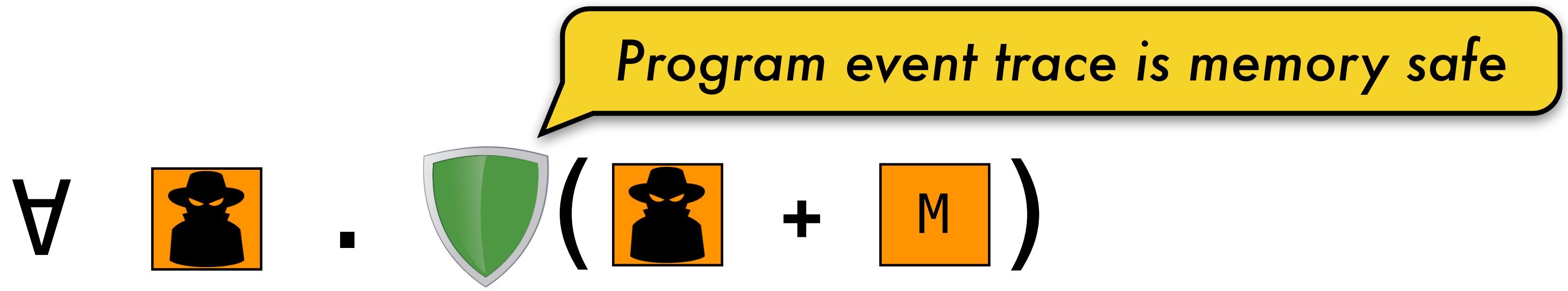
GraalVM

arm Morello

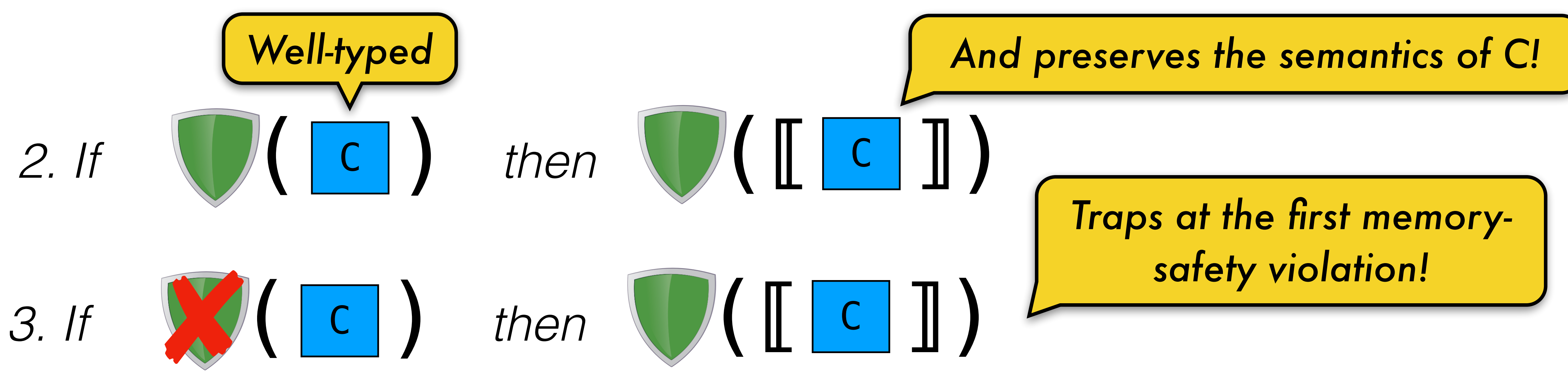
Evaluation on PolyBenchC

Formal Results

1. Any well-typed MSWasm module ***M*** *is* ***robustly memory safe****:*

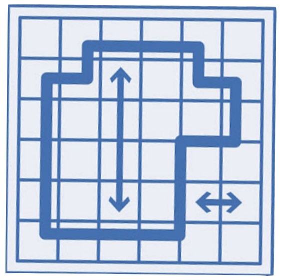


C-to-Wasm compiler enforces ***memory-safe execution of unsafe code****:*



This Talk

MSWasm Design



Color-based Memory Safety

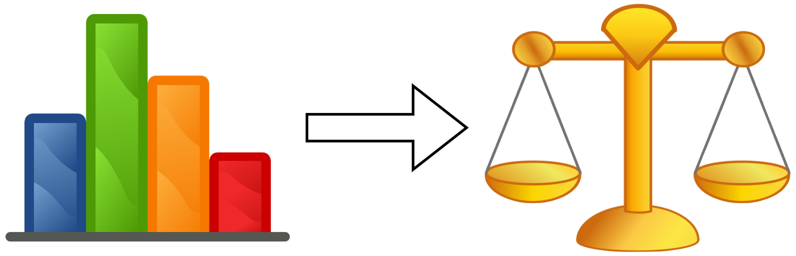
Formal Results

4 MSWasm Compilers

GraalVM™

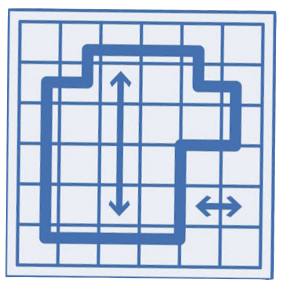
arm Morello

Evaluation on PolyBenchC



This Talk

MSWasm Design



Color-based Memory Safety

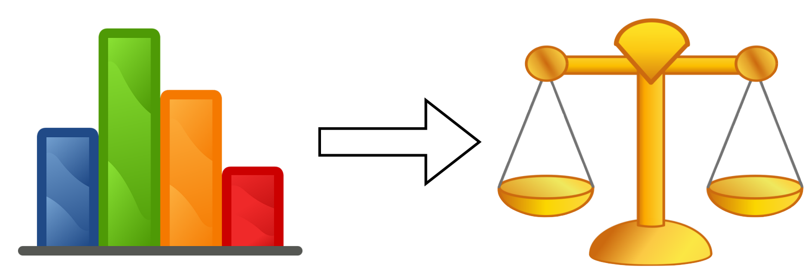
Formal Results

4 MSWasm Compilers

GraalVM

arm Morello

Evaluation on PolyBenchC



MSWasm Implementation

C-to-MSWasm compiler *based on Cheri fork of Clang/LLVM*

Reuse Cheri-LLVM "fat pointer" IR representation for handles

General design makes it easy to support different mechanisms

S = ***Spatial Safety***
T = ***Temporal Safety***
H = ***Handle Integrity***

MSWasm Backends	Based on	Type	Enforcement	Memory Safety
Rust	*rWasm*	*AOT*	*SW / 64-bit*	**STH, ST / S**
GraalVM™	*GraalWasm*	*JIT*	*SW*	**ST**
arm Morello	*Cheri LLVM*	*AOT*	*HW*	**SH**

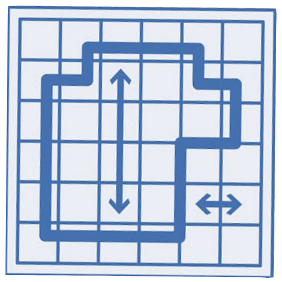
Baggy Bounds

Could optimize memory safety checks

Capabilities

This Talk

MSWasm Design



Color-based Memory Safety

Formal Results

4 MSWasm Compilers

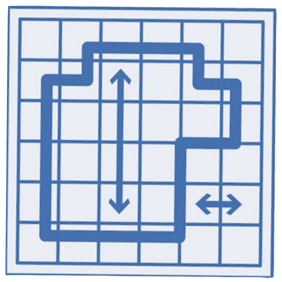
GraalVM

arm Morello

Evaluation on PolyBenchC

This Talk

MSWasm Design



Color-based Memory Safety

Formal Results

4 MSWasm Compilers

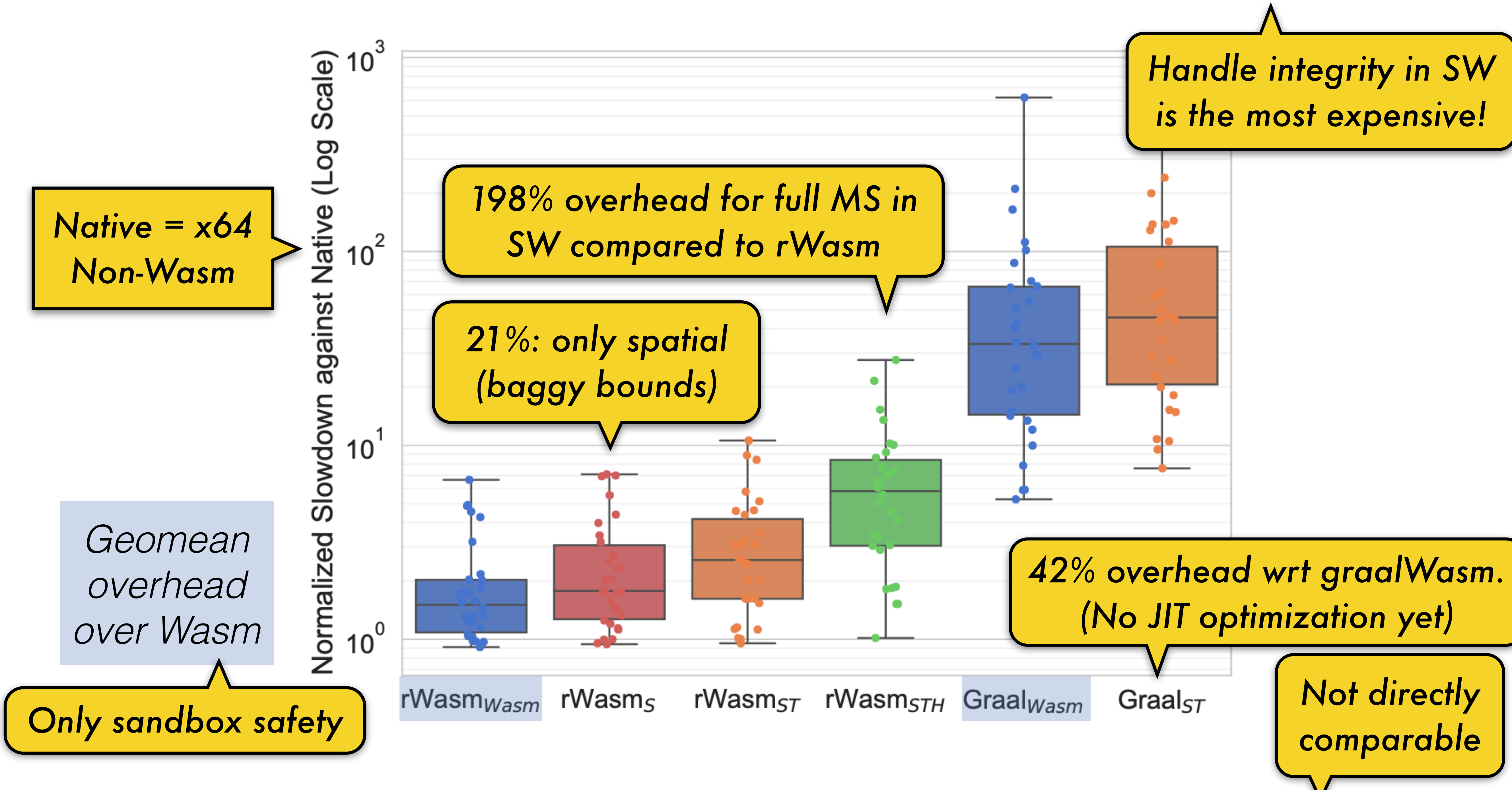
GraalVM

arm Morello

Evaluation on PolyBenchC

Evaluation of MSWasm on PolyBenchC

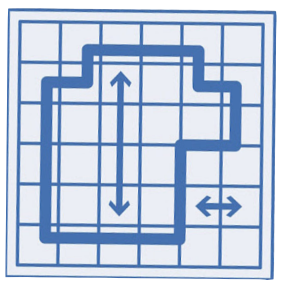
Each safety enforcement techniques comes with a runtime performance cost:



The overhead for enforcing ***Cheri-SH*** *in HW is* ***39% over native (aarch64)***

MSWasm: Soundly Enforcing Memory-Safe Execution of Unsafe Code

MSWasm Formal Specification



Well-typed MSWasm programs are robustly memory safe

Color-based Memory Safety

Language- and mechanism-independent definition

Sound C-to-MSWasm Compilation

Memory-safe execution of unsafe code

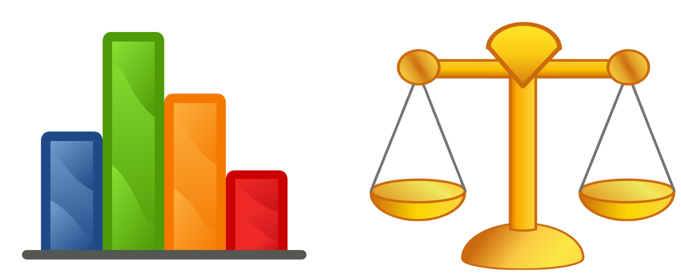
4 MSWasm Compilers

GraalVM™

arm Morello

Easy to support different enforcement mechanisms

Evaluation on PolyBenchC



General design enables performance-security tradeoffs