USENIX SECURITY '26

An Integer Overflow Endgame

Compiler and Architecture Support for Default-On Integer Overflow Mitigation

Zheng Zhang, Qi Ling, Kian Kasad — Purdue University · Brendan Ryan Sweeney — UT Austin
Deepak Gupta — Meta Platforms Inc. · Tal Garfinkel — Google · Kazem Taram — Purdue University



August 12, 2026

Overflows Are a Big Problem

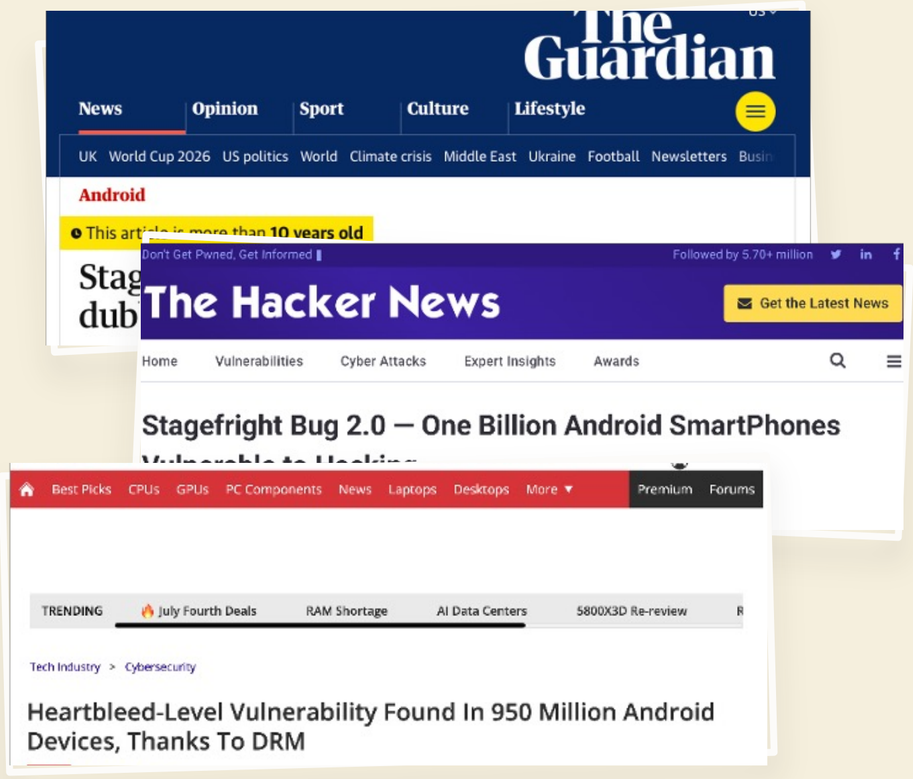
30 years since the Ariane 5 rocket crash — **we still haven't solved integer overflow.**

Source: ESA

Overflows Are a Big Problem

950M

Android devices exposed by **Stagefright**



We Know Where Overflows Happen

Signed overflows are Undefined Behavior in C/C++ — always a bug.

Unsigned overflows follow common patterns or can be annotated.

Modern safe languages (Swift, Rust) support explicit overflow annotations.

Overflow checks are off by default because they are too expensive.

Why Checks Are Too Slow (Compiler)

```
int func(int a, int b) {
  int c = a + b;
  return c;
}
```

Why Checks Are Too Slow (Compiler)

```
  int func(int a, int b) {
    int c = a + b;
+   int c_overflow = is_overflow(a, b);
+   if (c_overflow) goto handler;
    return c;
  }
```

Why Checks Are Too Slow (Compiler)

```
  int func(int a, int b) {
    int c = a + b;
+   int c_overflow = is_overflow(a, b);
+   if (c_overflow) goto handler;
    return c;
  }
```

Breaks Optimizations

- ✕ **InstCombine**
- ✕ **Reassociate**
- ✕ **Vectorization**
- ✕ **etc.**

Why Checks Are Too Slow (Hardware)

AT A GLANCE, IT LOOKS CHEAP

```
add eax, esi
jo  .handler
```

- **One instruction — the CPU already tracks overflow**
- **A predictable branch — almost never taken**

Why Checks Are Too Slow (Hardware)

BUT… IT'S NOT JUST ONE

Every checked arithmetic operation in the program needs its own `jo`



program instructions

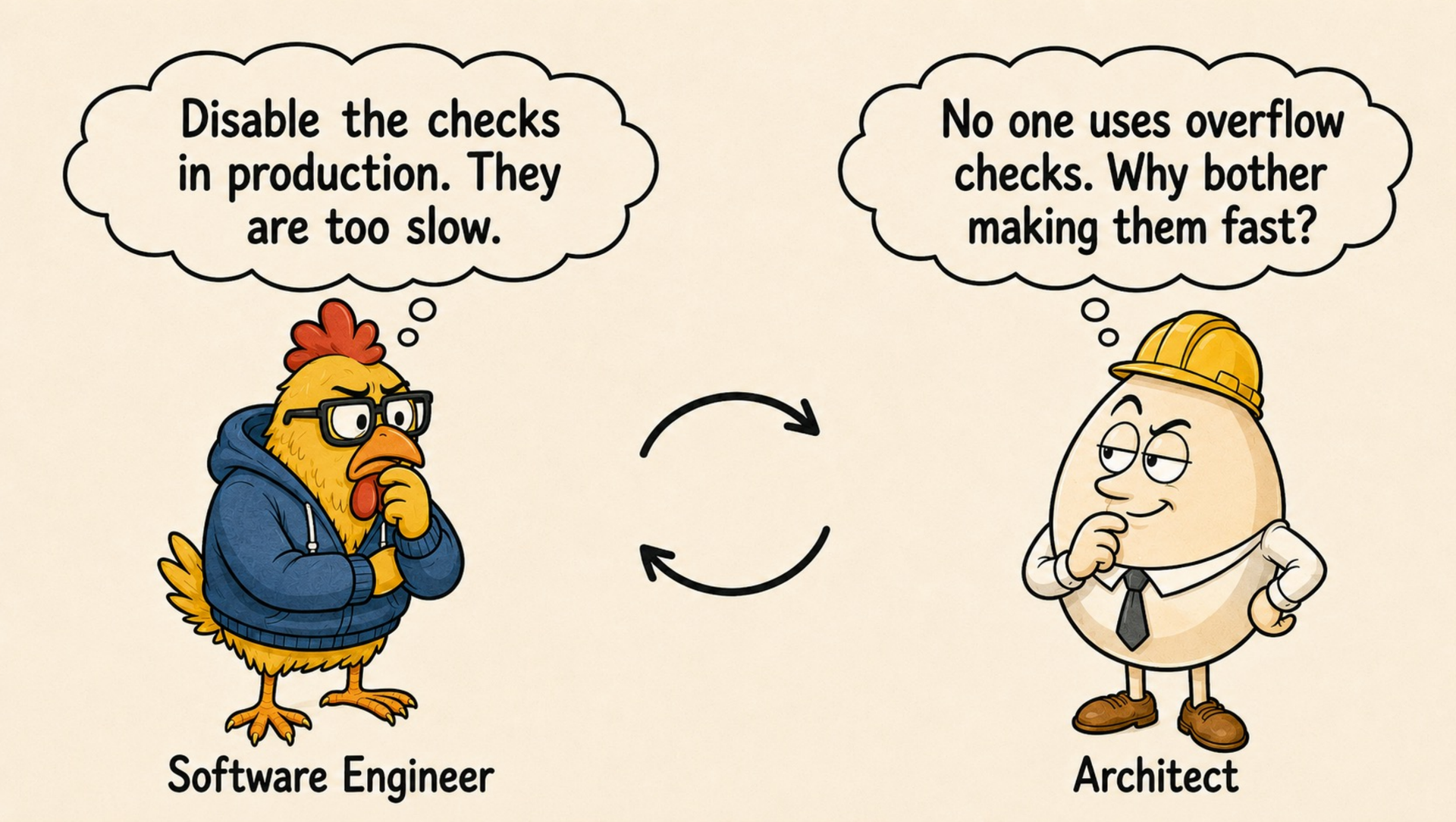
arithmetic instructions

1 in ~6

instructions is arithmetic

SPEC 2017 · x264

The Chicken-and-Egg Problem

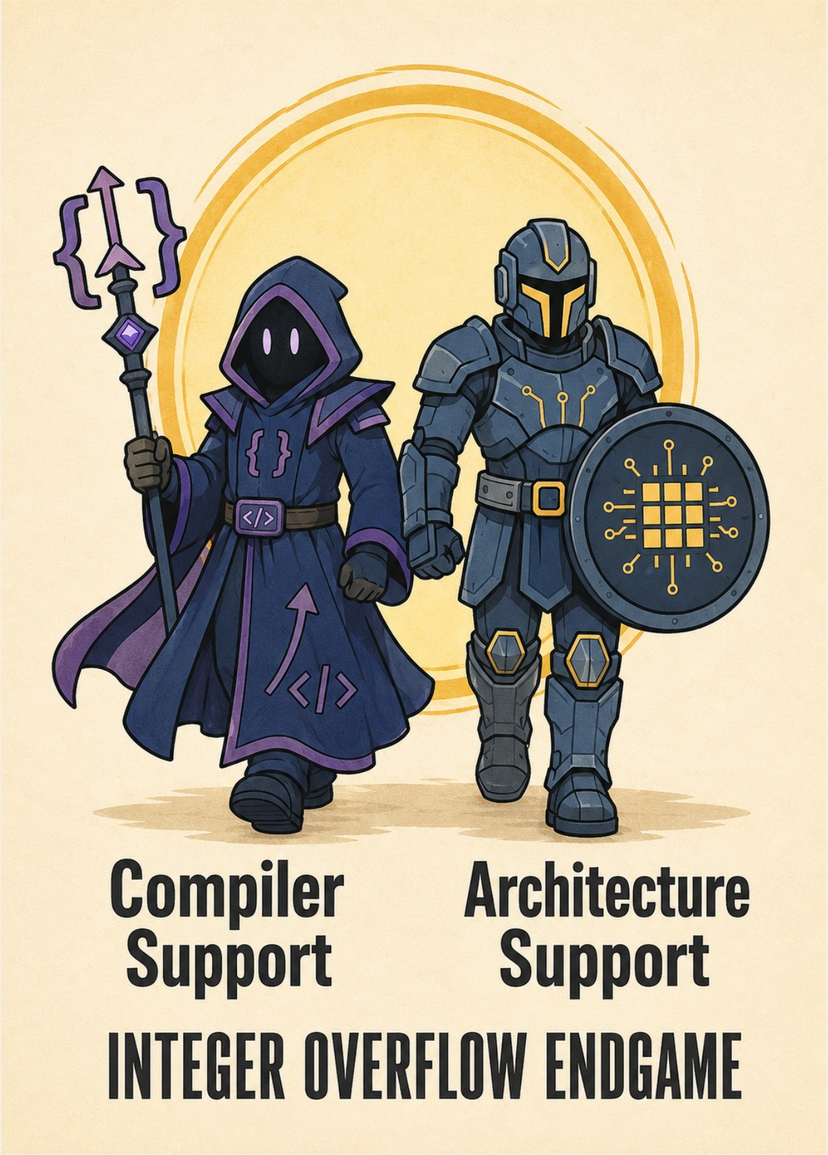


The End Game

Eliminate Compiler Overheads
Late Check Insertion (LCI)

Eliminate Hardware Overheads
ISA extensions for efficient checks

Near-zero cost, default-on overflow mitigation



Outline

✓ Motivation

Characterizing Costs of Overflow Checks

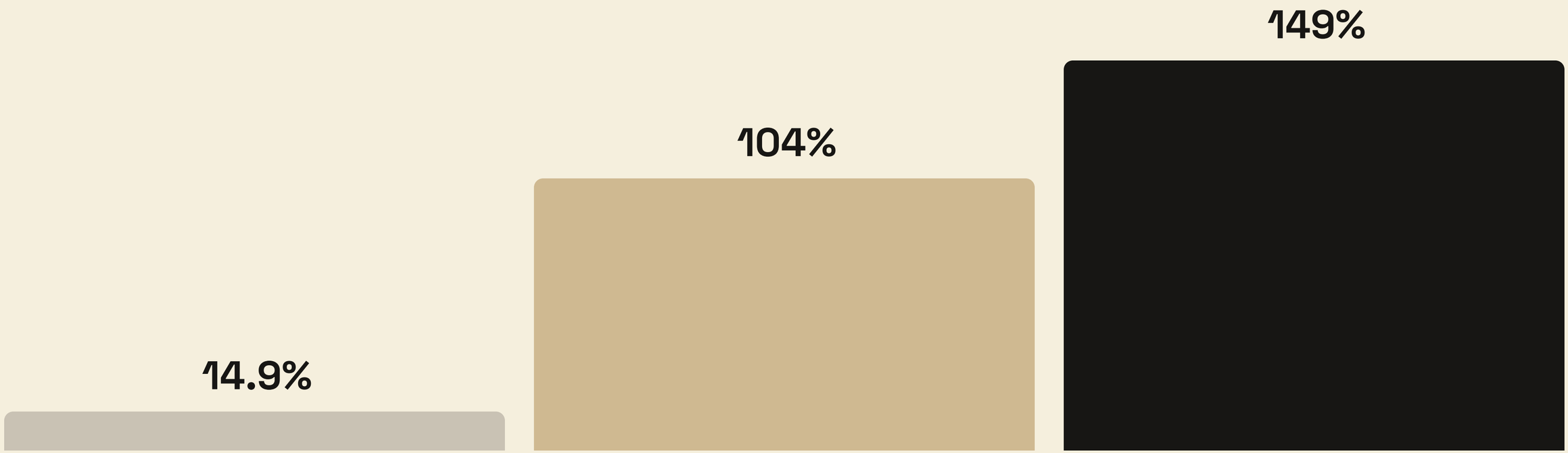
Integer Overflow End Game

— Compiler Solution

— ISA Extension

Overflow Checks Are Indeed Costly

Using UBSan — the default integer overflow checking in LLVM



↺ REMINDER

Sources of Overhead

Breaking Optimizations

Compiler

Extra Instructions

Hardware

Next: isolate each cost

Source 1: Compiler Overhead

THE EXPERIMENT — ISOLATE THE COMPILER COSTS

01

Keep the checks, drop the branches

02

Measure what is left

56% **of UBSan's overhead comes from the compiler alone.**

Source 2: Hardware Overhead

What is left is the cost of the `jo` branches themselves.

> **44%** **of the overhead comes from the branches.**

But *how* do extra branches cost that much?

Front-end pressure?	**Back-end pressure?**	**Branch prediction?**

Topdown Analysis for SPEC2017

OBSERVATION 1

The branches are well predicted

OBSERVATION 2

Retiring µops increase sharply

The sheer number of extra µops contributes the most.

Outline

✓ Motivation

✓ Characterizing Costs of Overflow Checks

Integer Overflow End Game

Compiler Solution

— ISA Extension

How Does UBSan Prevents Optimizations?

Optimization	Example transformation blocked
Reassociate	`x=a+b-c+d+1; y=2-(a+b-c+d); x+y → 3`

How Does UBSan Prevents Optimizations?

Optimization	Example transformation blocked
Reassociate	`x=a+b-c+d+1; y=2-(a+b-c+d); x+y → 3`
Constant propagation	`1+2 → 3`
Instruction combining	`x^(x&(-x)) → x&(x-1)`
Strength reduction	`i*4+j → lea eax, [rsi+4*rdi]`
Vectorization	Branches/intrinsics break auto-vectorization completely

Relaxing Overflow Semantics

UBSan aims to detect overflows in all **language-level** arithmetic operations, as a sanitizer.

Relaxing Overflow Semantics

UBSan aims to detect overflows in all **language-level** arithmetic operations, as a sanitizer.

Example: dead code elimination

```
int func(int x, int y) { int z = x + y; return 0; }
```

z is never used → dead code

Relaxing Overflow Semantics

UBSan aims to detect overflows in all **language-level** arithmetic operations, as a sanitizer.

Example: dead code elimination

```
int func(int x, int y) { int z = x + y; return 0; }
```

z is never used → dead code

```
; x86 without UBSan
xor eax, eax
ret
```

z is dead → the add disappears

```
; x86 with UBSan
add eax, esi
jo  .LBB0_1
...
```

the check keeps the add alive

Relaxing Overflow Semantics

UBSan aims to detect overflows in all **language-level** arithmetic operations, as a sanitizer.

Example: dead code elimination

```
int func(int x, int y) { int z = x + y; return 0; }
```

z is never used → dead code

```
; x86 without UBSan
xor eax, eax
ret
```

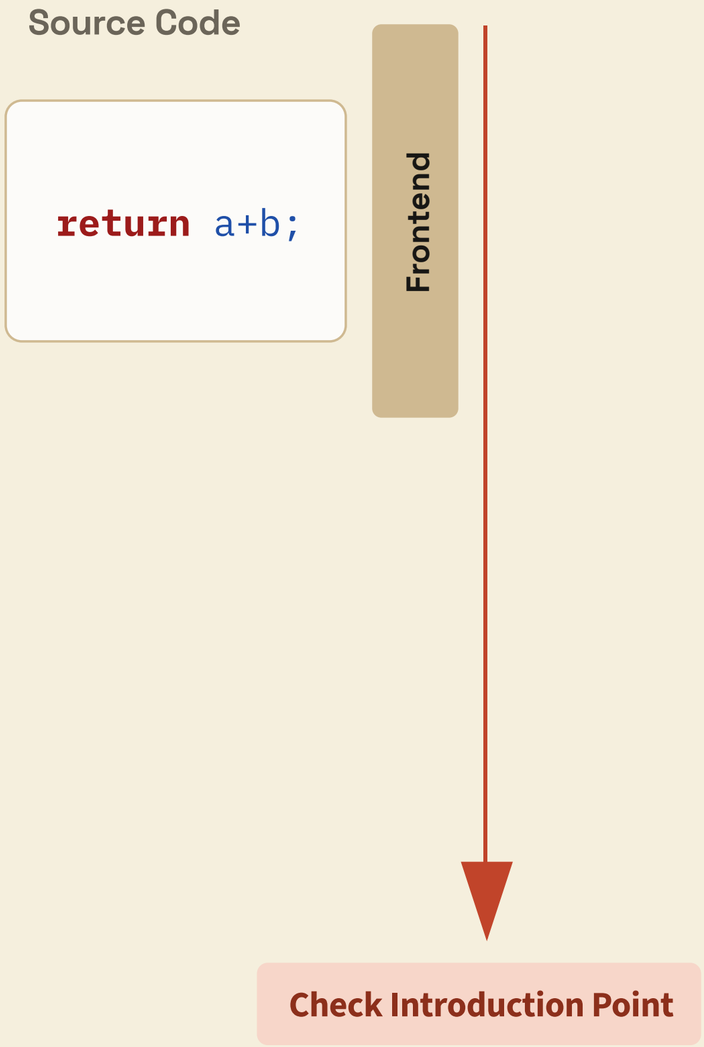
z is dead → the add disappears

```
; x86 with UBSan
add eax, esi
jo  .LBB0_1
...
```

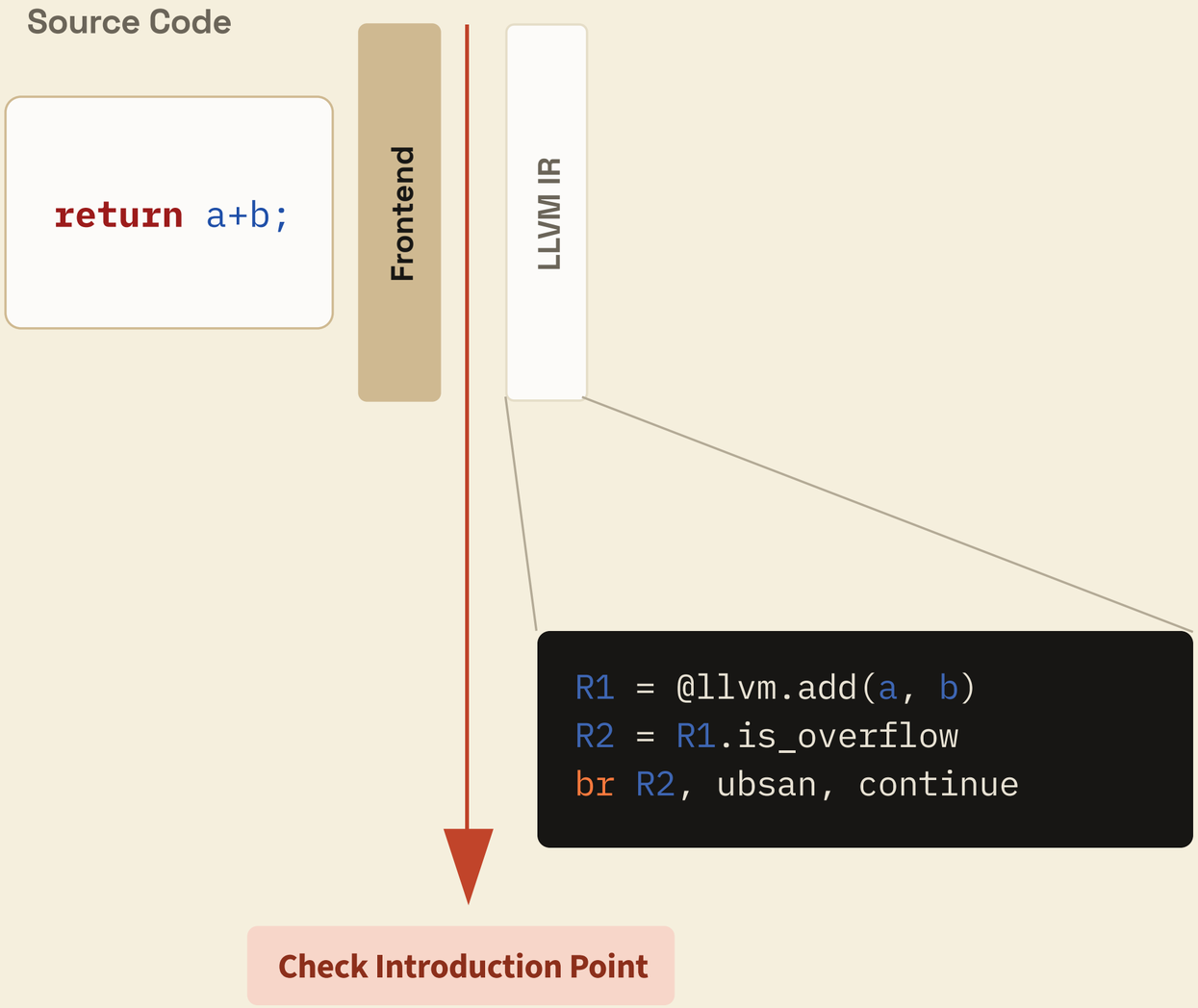
the check keeps the add alive

For mitigation, we only care about executed operations.

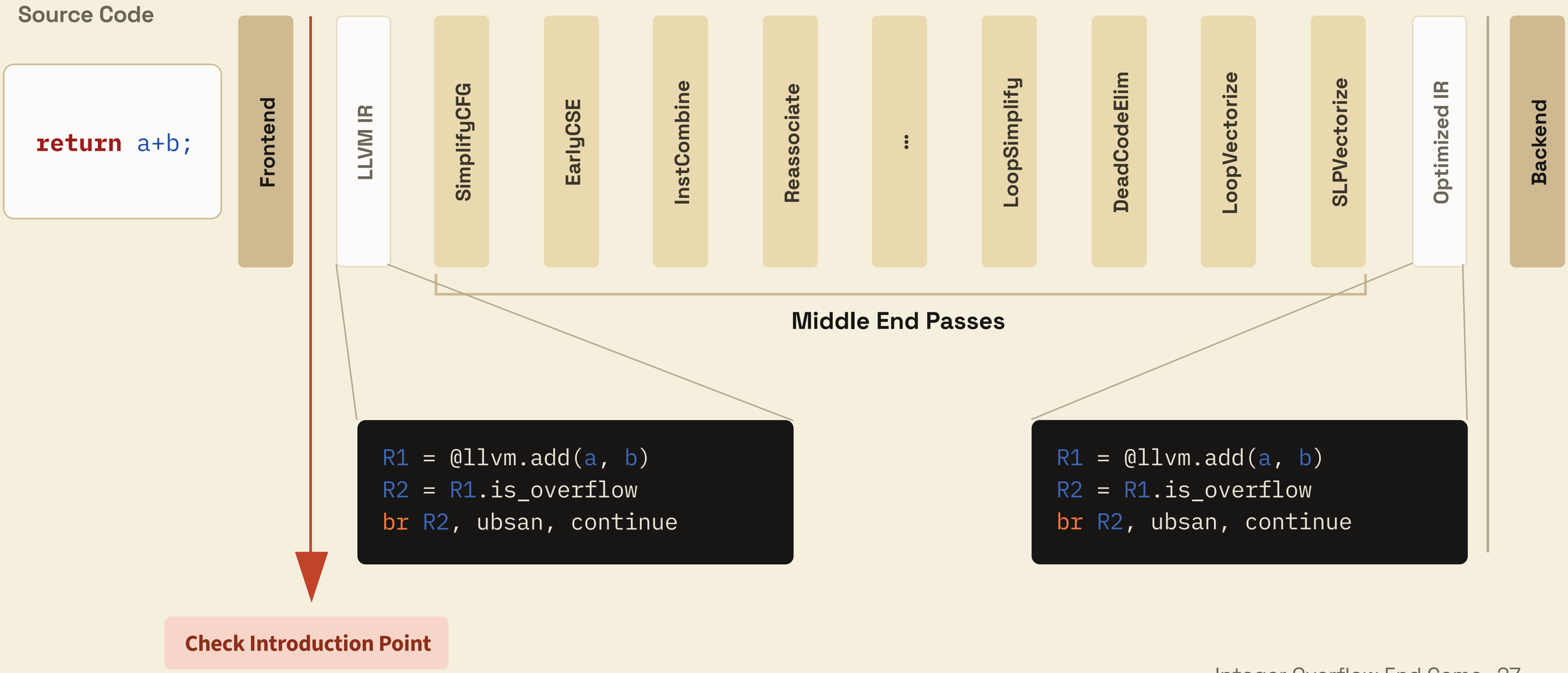
UBSan Process



UBSan Process



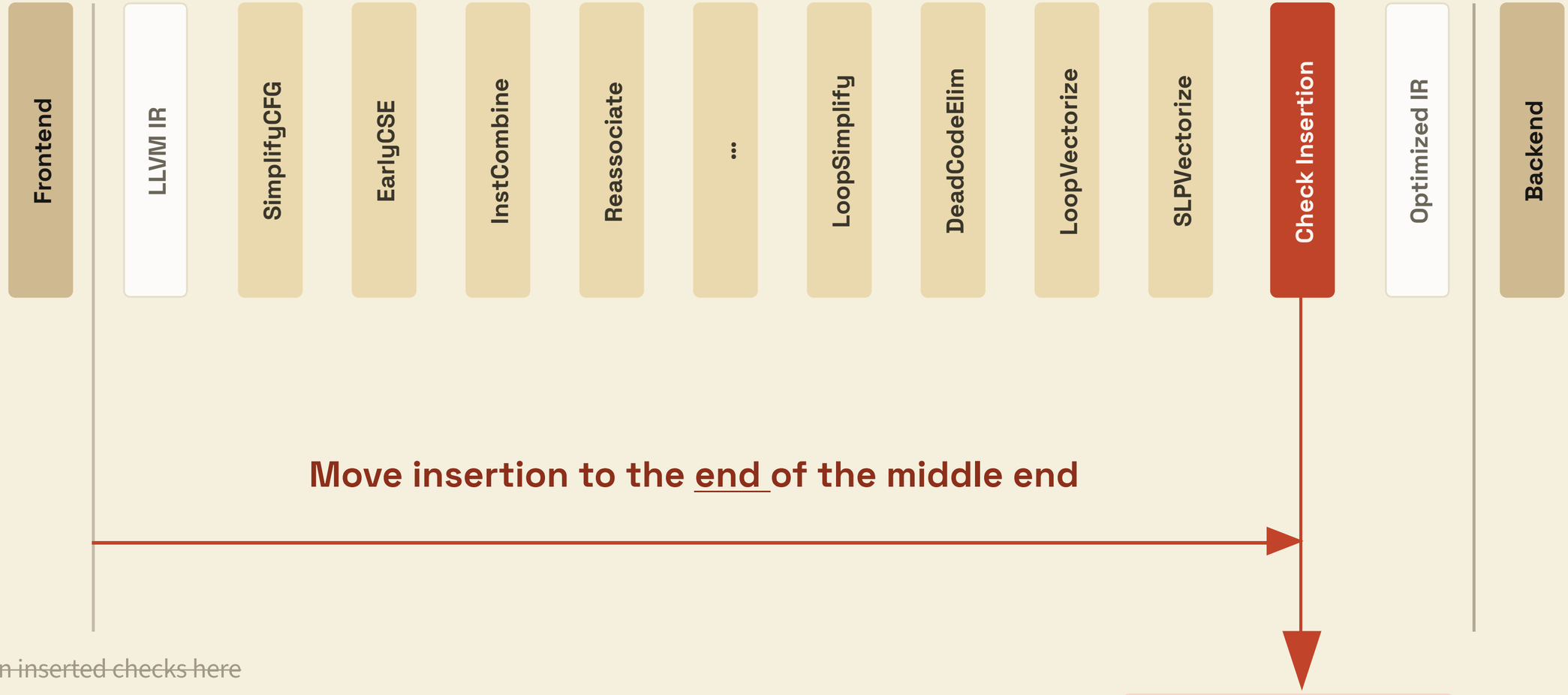
UBSan Process



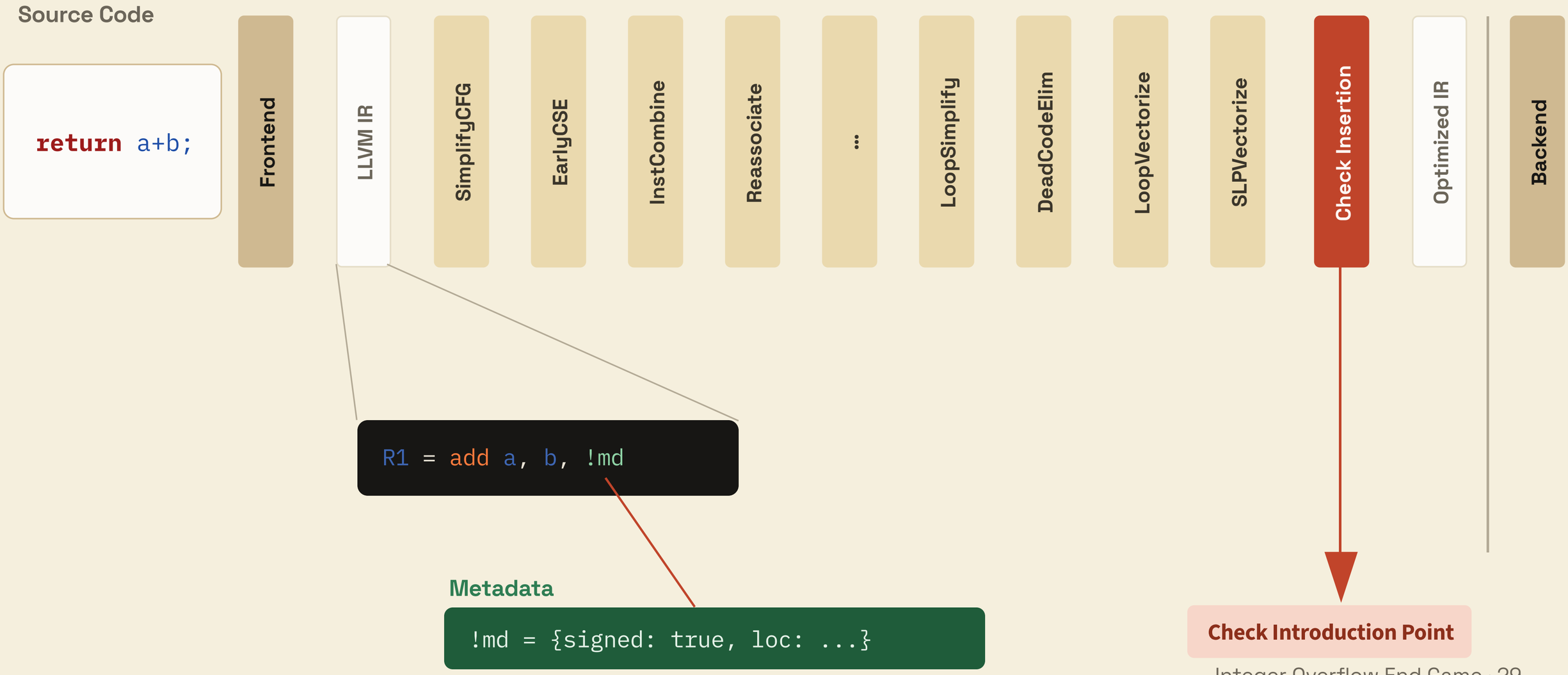
Late Check Insertion (LCI) Approach

Source Code

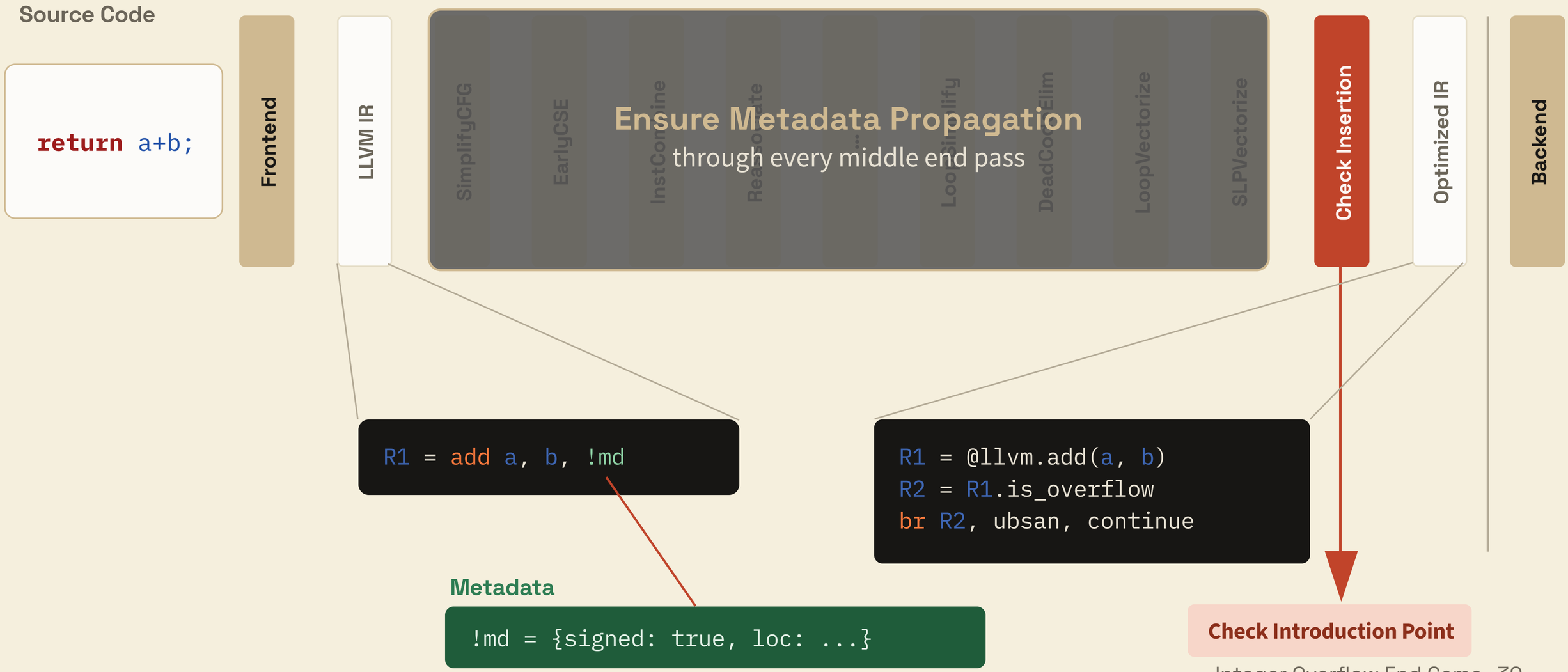
```
return a+b;
```



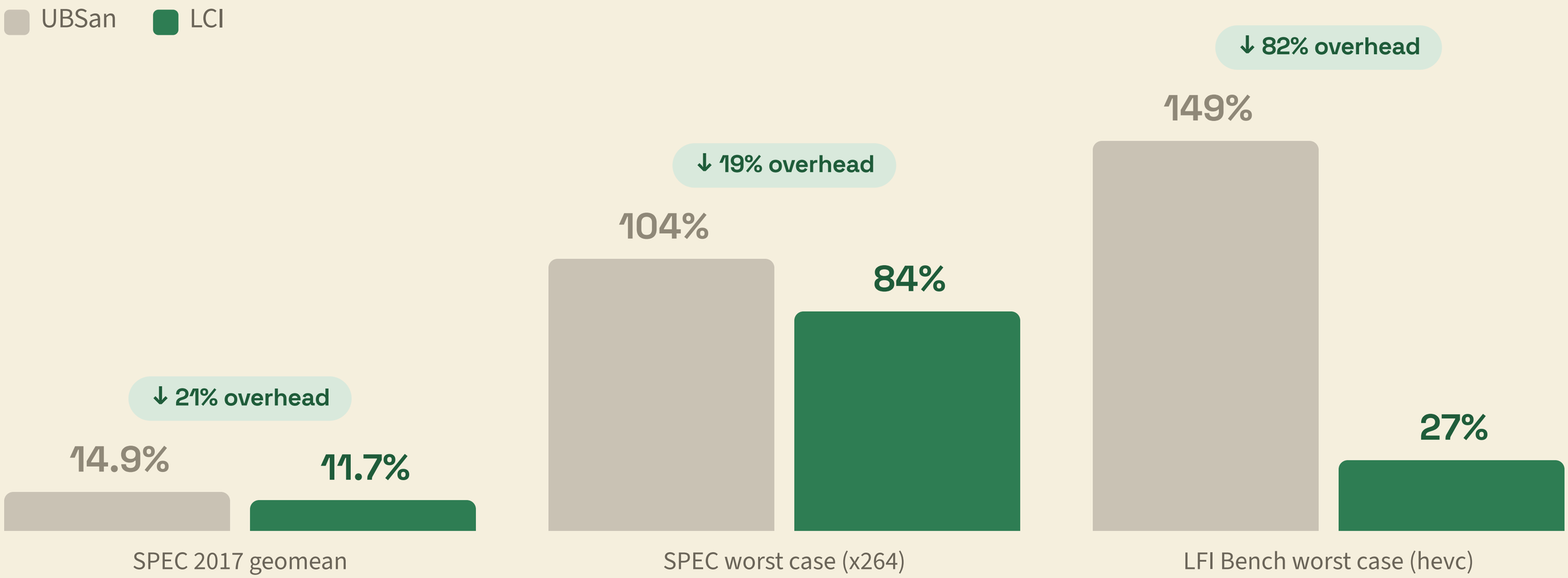
Late Check Insertion (LCI) Approach



Late Check Insertion (LCI) Approach



LCI Results (x86)



Outline

✓ Motivation

✓ Characterizing Costs of Overflow Checks

Integer Overflow End Game

✓ Compiler Solution

ISA Extension

Limitations of Current ISAs

No current ISA can say “add these, signed, and check for overflow” in one instruction.

1 No sign information

Signedness appears only in the branch, not in the add.

Limitations of Current ISAs

No current ISA can say “add these, signed, and check for overflow” in one instruction.

1 No sign information

Signedness appears only in the branch, not in the add.

2 No overflow semantics

“Check this one” is unexpressible.

Limitations of Current ISAs

No current ISA can say “add these, signed, and check for overflow” in one instruction.

1 No sign information

Signedness appears only in the branch, not in the add.

2 No overflow semantics

“Check this one” is unexpressible.

3 Reliance on branches

Only branch-on-overflow exists; blocks µop fusion.

Limitations of Current ISAs

No current ISA can say “add these, signed, and check for overflow” in one instruction.

1 No sign information

Signedness appears only in the branch, not in the add.

2 No overflow semantics

“Check this one” is unexpressible.

3 Reliance on branches

Only branch-on-overflow exists; blocks µop fusion.

4 No overflow flags

Like RISC-V.

More instructions → more µops → lower throughput.

opX: One Instruction, Checked

TRAP

Checked arithmetic **traps** instead of branching — the check combines with the arithmetic instruction itself.

One instruction. One µop. No control flow.

Expressing Overflow Semantics in Each ISA

Signed and unsigned checked variants — the real cost is encoding space, and every ISA has room somewhere.

x86

SEGMENT PREFIXES

Exploit unused prefixes — no new opcodes.

ARM

FUSIBLE PREFIX

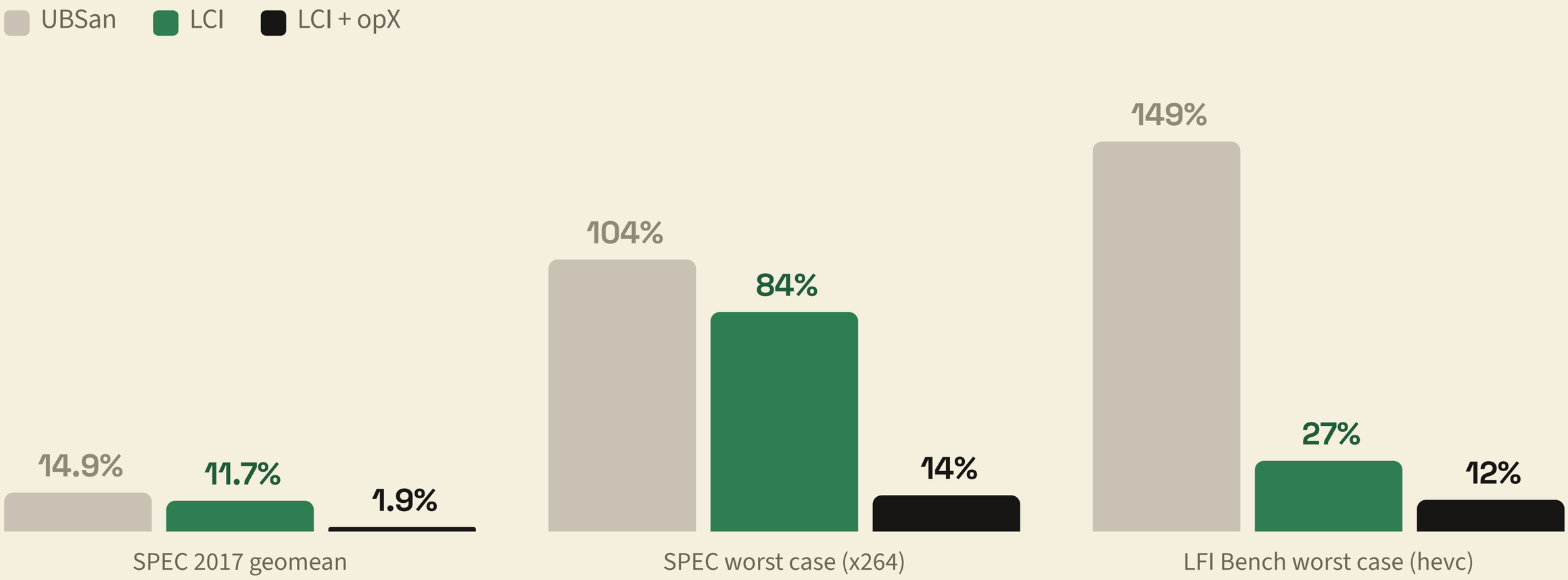
Propose a fusible prefix instruction.

RISC-V

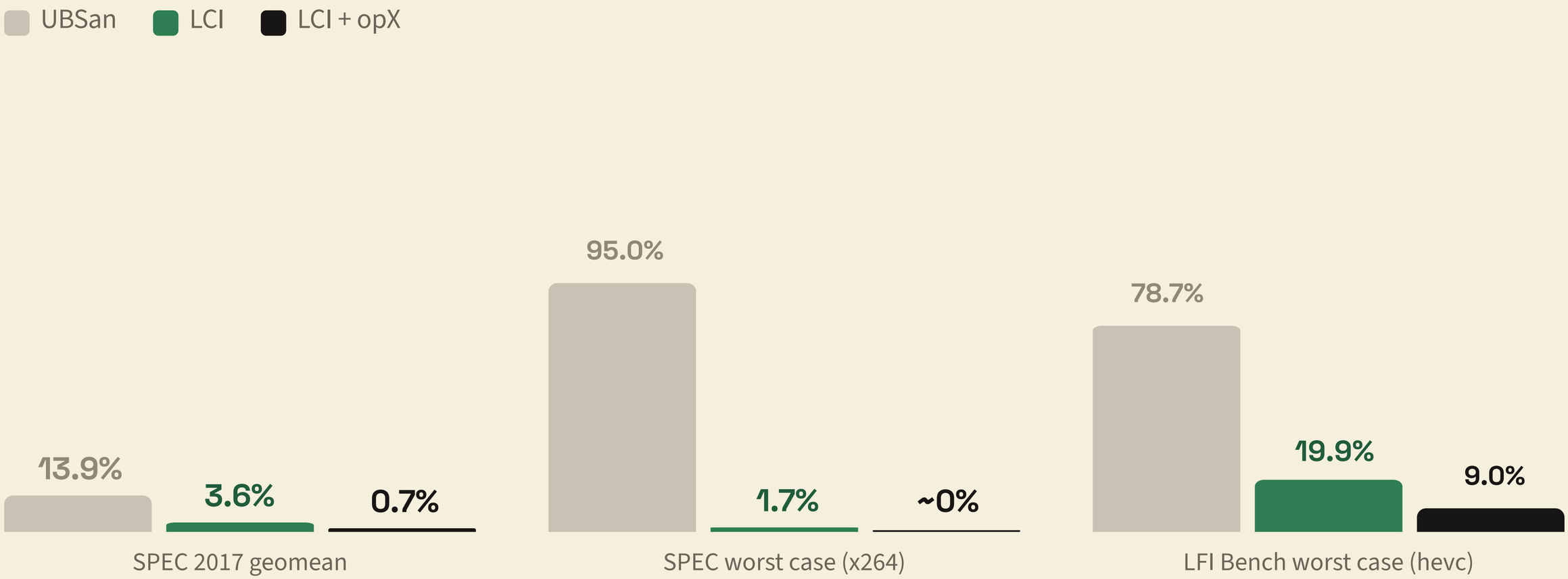
MINIMAL NEW OPCODES

Add a small set of checked variants.

LCI + opX Results (x86)



LCI + opX Results (ARM)



Security Evaluation

15

Integer-overflow CVEs

Selected from CWE-190 to test realistic vulnerability patterns.

15/15

Equivalence checked

Alive2 verified 11/15 functions; the remaining SMT-error cases were verified manually.

100%

Runtime coverage

LCI detects all runtime overflows detected by UBSan in SPEC.

Security result: the lower-cost path (LCI) preserves UBSan-style runtime overflow detection.

Conclusion

COMPILER

LCI

Delay checks so the compiler keeps optimizing.

ISA

opX

Make checks trap-based and cheap.

ENDGAME

Production goal

Near-zero cost default-on overflow mitigation.

Default-on integer overflow mitigation is practical with compiler + ISA support.

Thanks for Listening

Zheng Zhang · Purdue University

Paper

Artifacts

E-mail