An Integer Overflow Endgame: Compiler and Architecture Support for Default-On Integer Overflow Mitigation

Zheng Zhang
Purdue University

Qi Ling
Purdue University

Kian Kasad
Purdue University

Brendan Ryan Sweeney
UT Austin

Deepak Gupta
Meta Platforms Inc.

Tal Garfinkel
Google

Kazem Taram
Purdue University

Abstract

Integer overflow checks are *off-by-default* in most production settings due to their high overheads. Consequently, many serious vulnerabilities remain unmitigated. We analyze the source of these overheads—in compilers including LLVM (UBSan, Rust), and in processors including x86-64, ARM64, and RISC-V—and show how to eliminate them.

Overheads in LLVM stem from how overflow semantics are expressed in LLVM IR. LLVM's current approach, based on intrinsics and branches, interferes with a variety of middle-end optimizations. We develop an alternative approach that tracks overflow semantics with metadata, but otherwise leaves LLVM IR unchanged, eliminating this overhead. Overheads in hardware primarily result from current ISA designs, that require doubling the number of μops for checked vs. unchecked arithmetic, stressing the processor pipeline. We show how simple ISA extensions can alleviate this stress, enabling overflow-checked arithmetic to issue as a single μop.

We evaluate these extensions through compiler-based emulation on real hardware and simulation in gem5. Our work offers a path to significantly reducing the cost of overflow checks in existing compilers, and demonstrates that integer overflow mitigation can be cheap enough for default-on use in future hardware.

1 Introduction

Integer overflows have been a common cause of software failures dating back to the Ariane 5 rocket crash nearly 30 years ago [1]. At present, integer overflow accounts for roughly 4300 CVEs [4]. The Linux kernel has averaged over six critical overflow bugs per year for the past decade [13]. High profile overflow vulnerabilities include Stagefright [20], that exposed roughly 95% of Android devices to remote exploitation. More recently, several major exploit chains from the NSO group for WhatsApp [47], and the recent BLASTPASS [18] on iPhones made critical use of integer overflows.

We can mitigate these bugs with runtime checks. However, these checks are rarely enabled in production because of high and often unpredictable overheads they induce.

For example, when enabling integer overflow checks for C/C++ in Clang/LLVM using Undefined Behavior Sanitizer (UBSan) [12], we see worst-case overheads of 2.04× and a geomean slowdown of 14.9% on SPEC CPU 2017 integer benchmarks (§7.1). Similarly, enabling overflow checks for Rust benchmarks incurs a geomean overhead of 3.5% (§7.3). Unsuprisingly, Rust enables these only in debug builds [45].

We find these overheads stem from the design of current compilers and ISAs. In LLVM, front-ends for languages such as C/C++ and Rust express overflow checks using overflow-checked intrinsics. We find these intrinsics interfere with a wide range of compiler optimizations (§3), among these are auto-vectorization, Reassociation, Dead Code Elimination, and Instruction Combining (§3.3). Our results show this interference often accounts for more than half of cost of overflow checks (§3.3).

In current ISAs, overflow checks are expressed with additional instructions, such as the `jo` or `jc` conditional branch instructions on x86. The cost of these instructions can be deceptive. The cost of each check—a correctly predicted branch–is quite cheap. However, because a branch is required after every checked arithmetic instruction—they effectively double the number of μop's required for checked versus unchecked arithmetic. This in turn doubles number of pipeline resources consumed such as decode, issue, and commit slots. The resulting reduction in effective throughput in a program's critical sections can significantly degrade performance.

We demonstrate how both of these sources of overhead can be greatly reduced and often eliminated with modest changes to existing compilers and processor designs.

To eliminate compiler overheads, we start with the observation that LLVM's overflow-checked intrinsics—that are also used by Rust, Swift, etc.—were designed to support UBSan. UBSan aims to detect undefined behavior at the langauge level. Thus, breaking optimizations that could hide the presence of undefined behavior is a feature. However, for mitigating actual overflow bugs at runtime, we only care about

checking arithmetic the compiler actually produces—after all optimizations have run—irrespective of whether this is undefined behavior at the language level or not.

To avoid the cost of intrinsics, we introduce *late check insertion (LCI)*, a compiler approach that delays inserting overflow checks until after middle-end optimizations. We modify UBSan (and Rust overflow checking) at the stage where they lower their AST to LLVM IR. Instead of immediately replacing arthimetic with overflow-checked intrinsics, we track which arithmetic instruction must be checked with *metadata*. The LLVM IR is otherwise unchanged from normal unchecked arithmetic. After all the middle-end optimization passes have run, we then use this metadata to determine which instructions need to be converted to overflow-checked intrisics, before passing them to the back-end for lowering. Thus, we eliminate the problem of broken compiler optimizations.

To eliminate architecture overheads, we observe that there are multiple ways to express overflow-checked arithmetic that can be expressed in a single micro op (μop). In particular, with trapping semantics—rather than today's branching semantics—there is no need for an immediate (branch target) which is too large to fit into an existing arithmetic μop.

To explore this approach, we develope overflow protection eXtensions (opX), a simple set of ISA extensions—including opX-x86, opX-RISC-V, and opX-ARM—that express trapping overflow semantics in a manner that can be encoded in a single μop.

On ISAs like RISC-V, which have no native support for overflow checks, we explore a single instruction encoding—adding instructions for overflow-checked arithmetic, similar to how architectures of the past such as MIPS and PA-RISC dealt with signed arithmetic. On other ISAs (x86-64 and ARM64), we explore designs based on instruction prefixes and instruction fusion, observing that although these add some amount of CPU front-end pressure, the cost is negligible in practice. We explore extending these checks to the full range of overflowable instructions including normal arithmetic, SIMD, and those used for strength reduction (e.g., `lea`, `shl`).

We evaluate both our compiler approach and ISA extensions using SPEC CPU 2017 and a collection of widely used libraries (including multimedia, compression, and XML parsing workloads) for C/C++, the NAS Parallel Benchmarks [34], and programs derived from Google's SwissTable hash table library for Rust [8].

On SPEC CPU 2017, our approach to eliminating compiler overheads reduces the geomean cost of UBSan on x86 from 14.9% to 11.7%, and worst-case overhead from 2.04× to 1.84×.

On ARM it cuts geomean overhead from 13.9% to 3.6%, and worst-case overhead from 95% to 10%.

On Rust workloads on x86, eliminating compiler overheads lowers the geomean overhead of overflow checking from 3.5% to near zero, and worst-case overhead from 10.7% to 1.3%.

To evaluate eliminating the remaining overheads in hardware level, we modified Clang/LLVM and the Rust compiler to support our new hardware extensions. We evaluate them using compiler-based emulation on real hardware and simulation in gem5. On x86, our implementation reduces the overhead to around 0.3% geomean on SPEC 2017, and worst-case overhead of 3.6% compared to UBSan's 2.04× overhead.

2 Why are integer overflows a problem?

How integer overflows happen. Modern processors use a fixed-width representation for integers (e.g., 32 or 64-bit). Integer overflows occur when the result of an operation exceeds the maximum value this width can represent. This often results in behavior the programmer did not anticipate. For example, with unsigned addition the result ends up smaller than the operands; with signed addition, the result ends up with a different sign from the operands.

Signed overflow is undefined behavior in C/C++, while unsigned overflow is well defined [25, 26]. However, both can lead to memory safety vulnerabilities. For example, overflows in integers used for indexing arrays or in pointer arithmetic often result in buffer overflows. Similarly, overflows in a size passed to `malloc()`, can return a buffer much smaller than the developer expected, again resulting in a buffer overflow [3]. Temporal safety bugs can also occur due to integer overflows, for example, as the result of a reference count overflowing [2]. More generally, integer overflows can give rise to difficult logic bugs [9]. Thus, modern memory safe languages like Rust [44] and Swift [23] offer first class support for discriminating these from intentional overflows.

That said, there are many legitimate applications of unsigned wrap around (overflow)—which is essentially fast addition modulo 2^n. This includes hashing, encryption, protocols sequence numbers, and indices into ring buffers. Often the core problem of integer overflows is not that they can happen, but that code is compiled in a manner that does not match the programmer's expectation.

The Chicken and Egg Problem. At the language level there are ample facilities to express the programmer's intent: For example, Rust offers different types for integers that should wrap around or trap on overflow [44]; Swift offers different operators for trapping versus wrap around [23], and C++ allows intent to be expressed through typed wrappers (e.g., `boost::safe_numerics`) [41].

However, compilers typically default to allowing overflows to occur silently, since runtime checks can impose high and unpredictable costs. As noted earlier, this stems from limitations of current ISAs: overflow detection interferes with common compiler optimizations and expands the instruction stream. As a result, developers rarely enable the overflow checks that compilers do provide in production. For instance,

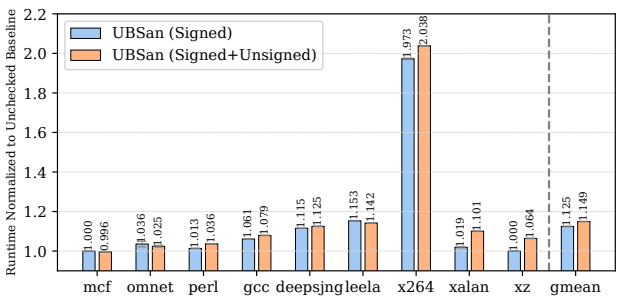

Figure 1: **UBSan Performance Overhead with Signed and Unsigned Overflow on x86.** *Most benchmark programs show significant overhead for signed and/or unsigned overflow checking.*

C/C++ projects seldom use UBSan outside of testing. Similarly, Rust enables overflow checks by default in debug builds but not in release builds.

Thus, we end up with the classical chicken and egg problem of computer architecture. Because overflow checks are slow, they are not deployed in production, and because they are not deployed in production, processor architects do not bother to improve their performance.

3 Isolating the Cost of Overflow Checks

In this section, we analyze how current integer overflow checks limit performance in compilers and hardware.

3.1 Integer Overflow Checking with UBSan

Undefined Behavior Sanitizer (UBSan) [12]—which ships as part of Clang/LLVM [1]—is the dominant tool used to detect integer overflow bugs in C/C++ software today. As discussed, it is primarily used for testing, although it does see occasional production use in security-critical settings. For example, Android started enabling UBSan overflow checks for high-risk code (e.g., certain media codecs, parsers, the bluetooth stack) [6] after Stagefright [20].

UBSan works by adding integer overflow checks at LLVM front-end, the stage where Clang lowers the AST to LLVM IR. UBSan can check for overflows in signed, unsigned, and pointer arithmetic. It also supports a variety of heuristics to detect and avoid checking intentional overflows, to prevent false positives.

To check for overflows, UBSan replaces normal arithmetic instructions with an intrinsic function call and a conditional branch. This is necessary as LLVM IR does not have a built-in notion of an overflow flag. The intrinsic (e.g., `llvm.sadd.with.overflow.i32`) computes both the arithmetic result and an overflow flag. The branch then tests the overflow flag and transfers control to the corresponding UBSan handler (e.g., `__ubsan_handle_add_overflow`) if overflow happens.

UBSan also supports overflow checks for negation and division, which only overflow on `-INT_MIN` and `INT_MIN/(-1)` respectively, or for bitwise operations. Additionally, only 32-bit and 64-bit integers are checked because narrower integers are automatically promoted to 32-bit signed integers according to C/C++ semantics [25, 26].

To understand the causes of overhead in UBSan, we analyze SPEC 2017 integer benchmarks on an x86 server. Our methodology is described in detail in Section 6. To isolate different effects, we first focus on how overflow checks interfere with compiler optimizations then turn our attention to the microarchitectural sources of overhead due to the instructions UBSan adds for overflow checks.

3.2 UBSan's Costs on SPEC 2017

UBSan's overheads for signed and unsigned overflow checks are shown in Figure 1. The cost of signed overflow checks dominate, with a 12.5% geomean slowdown, while unsigned checks adds a further 2.4%. While most benchmarks exhibit noticeable slowdowns, the worst one, *x264*, suffers a dramatic 2.04× increase in runtime.

In the following subsections, we break down the cause of these overheads. This critically informs the design of opX and LCI, which seeks to eliminate these overheads.

To start, we look at how UBSan breaks compiler optimizations as result of how ISAs express integer overflow checks—and try to tease this out from optimizations that are broken as an artifact of UBSan's design. One challenge with trying to isolate the cause of different overheads here is that multiple factors can impact the same compiler optimizations. For example, converting normal arithmetic to intrinsics as UBSan does hinders instruction combining, and adding branches—required by UBSan because of how ISAs check for overflows—also hinders instruction combining.

Next, we look at overflow check overheads from an architecture perspective, looking at how current flag-check-and-branch based ISAs (e.g., x86 and ARM), and manual overflow checking (e.g., RISC-V) limit performance.

3.3 Source 1: Compiler Optimizations

The Cost of Intrinsics. As discussed, UBSan adds overflow checks by replacing normal LLVM IR arithmetic instructions with *UBSan Intrinsics*. As a side effect this breaks or constrains many of LLVM's optimization passes, that are designed to operate on normal LLVM arithmetic instructions and do not understand the semantics of intrinsics.

[1] A version of UBSan is also present in GCC but supports only signed integer overflows and fewer advanced features, like ignoring common idioms for intentional wrap around.

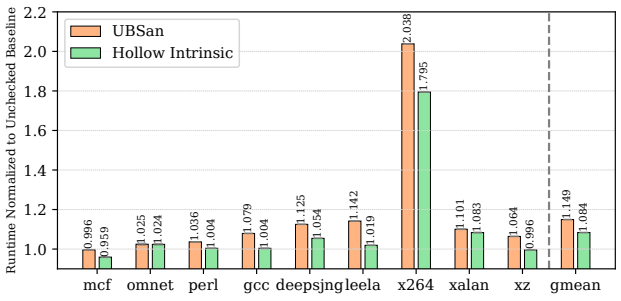

Figure 2: **The impact of UBSan intrinsics on compiler optimizations on x86.** *Hollow Intrinsics do not introduce any conditional branches and only interfere with compiler optimizations. This interference accounts for 8.4% out of the total 14.9% UBSan cost.*

To isolate this effect from other sources of overhead (e.g., added branches that interfere with optimizations, or additional instructions) we created a new type of intrinsics, which we call *Hollow Intrinsics*.

Hollow Intrinsics are simply normal arithmetic instructions in intrinsic form, so in the compiler's back-end these are lowered to normal arithmetic instructions. By modifying UBSan to emit these instead of its own intrinsics and additional branches, we are able to isolate the impact of UBSan's use of intrinsics on optimizations.

Figure 2 shows the performance impact of Hollow Intrinsics compared to standard UBSan intrinsics on x86. The results indicate that, on average, 8.4% out of the total 14.9% overhead of UBSan comes from interference with compiler optimizations. This cost is significant for some benchmarks, especially *x264*, where the impact of lost optimization is very large. In others, this source is smaller than the other major source of overhead (the added instructions). The numbers for ARM are found in Appendix A.3.

The Cost of Branches. In current ISAs overflow checked arithmetic unavoidably introduces a conditional branch—with two different outcomes, either continue execution or call a fault handler—for every checked instruction. This inhibits a variety of optimizations. For example, instruction combining is inhibited, since the optimizer can no longer transform IR in a manner that preserves semantics (consistent regardless of control flow).

Consider the C statement `x = (i * 4) + (s);` which can be lowered to a single instruction like `lea eax, [rsi + 4*rdi]` in x86. But since checking for overflow for the multiplication `(i * 4)` introduces a branch with visible side-effect, the instruction selection can no longer fold the multiplication and addition into a single `lea` instruction.

For similar reasons, this breaks Reassociation (as shown in Figure 3), auto-vectorization (as bundles can no longer be formed), and many other optimizations.

```
int func(int x, int y){
    int z = x + y; return 0;
}
;x86 without UBSan          ;x86 with UBSan
func(int, int):             func(int, int):
  xor  eax, eax               mov  eax, edi
  ret                         add  eax, esi
                              jo   .LBB0_1
                              ...
```

Listing 1: **Missing optimizations due to UBSan's overflow semantics.** *Because UBSan operates at language-level, it must preserve all the code that is potentially overflowing, even if that code is dead.*

Unfortunately, isolating this cost is challenging. While our hollow intrinsics give us the cost of intrinsics without branches, intrinsics inhibit many of the same optimizations. For example, Instruction Combining and Reassociation, as the optimizer cannot reason about the algebraic properties of intrinsics as it can with arithmetic instructions, and it cannot vectorize them for similar reasons.

UBSan Overflow Semantics. The semantics in UBSan are overflow checks at the *language level*: If there is an overflow bug in arithmetic in the program as written, it will signal a fault. While this is ideal for *detection* it is not ideal for *mitigation* in production settings where keeping performance overheads low is paramount, as these semantics break a variety of optimizations.

Optimizations such as Reassocation, Instruction Combining, Constant Folding, and Common Sub-expression Elimination may eliminate instructions which would cause integer overflows at the language level. For bug finding it is important that these are preserved, but for mitigating overflow vulnerabilities at runtime all that matters is that the overflow never occurs. Therefore, we only check arithmetic instructions that are actually executed, ignoring those altered or removed during optimizations.

Consider, for example, the simple C snippet in Listing 1. The addition `x + y` is clearly dead code and can be easily optimized out when UBSan is disabled. However, with UBSan this optimization is blocked: Because UBSan operates at the language level, it must preserve every operation that could potentially overflow. As a result, the compiler is forced to keep both the arithmetic and the associated overflow check, even if the value itself is never used.

Breaking SIMD. Auto-vectorization by the compiler enables automatic use of SIMD instructions across a wide range of applications, significantly improving performance. However, on current ISAs overflow checks fundamentally break auto-vectorization.

First, branches break auto-vectorization. The vectorizer cannot bundle operations together when the code alternates between arithmetic operations and conditional branches.

Source Code

```
int func(int a, int b, int c, int d)
{
  int x = a + b - c + d + 1;
  int y = 2 - (a + b - c + d);
  return x + y;
}
```

Optimized LLVM IR with UBSan

```
define i32 @func(i32 a, i32 b, i32 c, i32 d){
    %1 = @sadd.with.overflow(a, b) ; tail call
    %2 = extractvalue { i32, i1 } %1, 0
    %3 = extractvalue { i32, i1 } %1, 1
    br i1 %3, label %4, label %5 ; a+b overflowed?
 4: tail call void @ubsan_handle_add_overflow(a, b)
    br label %5
 5: ... ; calcuate and check the rest
    ret i32  %33   ;total of 62 IR instructions!
}
```

Optimized LLVM IR **without** UBSan

```
define i32 @func(i32 a, i32 b, i32 c, i32 d){
}  ret i32 3
```

Figure 3: **An example of how UBSan interferes with compiler optimizations.** *Without UBSan, LLVM's Reassociation pass reduces the code to a single IR instruction. With UBSan, overflow intrinsics after every arithmetic operation block the optimization, resulting in 62 IR instructions for the same code.*

Second, auto-vectorization depends on other optimizations such as Instruction Combining and Reassociation which are also disrupted by these branches as discussed above.

The problem runs deeper than just compiler limitations: Current ISAs lack support for overflow checks in SIMD. Existing SIMD instructions do not update the overflow or carry flags (except rare cases like saturating arithmetic in ARM). Thus to check for overflow, the compiler must insert many additional instructions, negating the performance benefits of SIMD and making efficient vectorized overflow checking effectively impossible with today's architectures.

The worst-case slowdown occurs in *x264*. Because *x264* has substantial opportunities for auto-vectorization, we hypothesize that UBSan intrinsics are blocking vectorization and related optimizations, driving its overhead.

To test this, we isolate the effect of the intrinsics on specific compiler passes. We interpose in the LLVM middle-end pipeline at various points and progressively delay the injection of our Hollow Intrinsics (intrinsics that break optimizations similar to UBSan intrinsics) to later IR passes. As the injection point moves later, previously blocked optimizations, such as vectorization, Instruction Combining, and Reassociation begin to apply, and execution time improves. The detailed experimental setup is described in Section 6.

Figure 4 shows the results of this experiment, the two runs are highly consistent. Performance improves sharply once

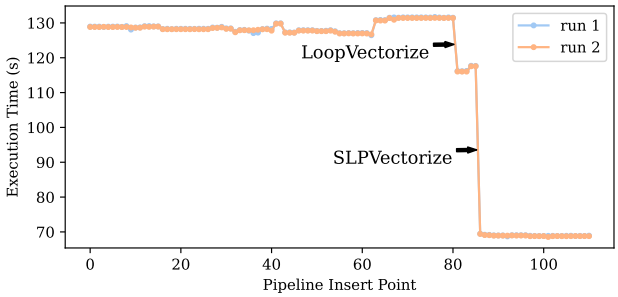

Figure 4: **Isolating the impact of intrinsics on *x264*'s optimization passes on x86.** *Progressively delaying the insertion of Hollow Intrinsics in the pipeline shows that vectorization passes (LoopVectorize and SLPVectorize) account for the largest share of the overhead.*

we prevent the intrinsics from impacting two key passes: LoopVectorize and SLPVectorize. The loop vectorizer operates across iterations of loops, while SLP vectorizer exploits superword-level parallelism within a single loop iteration. These two passes form the core of LLVM's auto-vectorization framework [30]. These results confirm that inhibiting vectorization is the main driver of *x264*'s large overhead.

3.4 Source 2: Degrading µArch Performance

Next, we analyze the second source of overhead, the microarchitectural cost from the extra conditional branches introduced by overflow checks. As observed in Figure 2, by removing the conditional branches introduced by UBSan, Hollow Intrinsics eliminates 6.5% of the total 14.9% overhead. This significant overhead results from the fact that conditional branches affect all stages of a pipeline:

- In the front-end, not only do more instructions have to be fetched and decoded, the conditional branch predictor is also confronted with more branches, which degrades the prediction accuracy given the limited capacity of prediction tables [60].
- In the back-end, more conditional branches lead to more port contention with other instructions [5], as well as more instructions to be retired.

We conduct an experiment to demonstrate the impacts of extra conditional branches on all stages of a pipeline on x86. As in Listing 5a, a simple loop is constructed where each loop iteration consists of 128 `INC` instructions with data dependencies. To examine the cost of fetching and decoding extra branch instructions, a 6-byte [2] `nop` instruction is added after each `INC`, as in Listing 5b. Next, to examine the impact of extra branches on the branch predictor, as well as their execution and retirement costs, a `jo` instruction is added after each `INC`, as in Listing 5c. Each loop is executed for a million

[2] Typical size of `jo` in code

```
.Lbody:
.rept 128
inc %rax

.endr
dec %edi
jnz .Lbody
```

(a) Baseline (6.60 ms)

```
.Lbody:
.rept 128
inc %rax
nop
.endr
dec %edi
jnz .Lbody
```

(b) `nop` (11.88 ms)

```
.Lbody:
.rept 128
inc %rax
jo .handler
.endr
dec %edi
jnz .Lbody
```

(c) `jo` (17.72 ms)

Listing 5: **Assembly code snippets testing the µArch impact of extra condition branches on x86.** *Each loop is executed for a million times.* `nop` *version is 80% slower than the baseline, highlighting the cost of fetching and decoding extra instructions.* `jo` *is 88% slower than the* `nop`*, highlighting the cost of branch prediction, executing and retiring extra branches.*

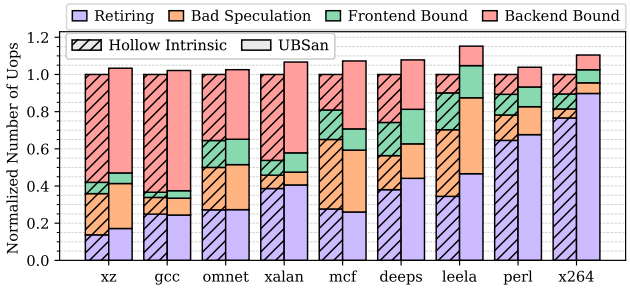

Figure 6: **Top-down analysis of the µArch overhead from branches introduced by UBSan on x86.** *Retiring contributes most to the overhead (50% on average), followed by back-end stalls (38%) and bad speculation (15%). Front-end is not a significant source of overhead.*

times and the latencies are measured.

Results show that the fetch and decode of extra `nop` instructions causes an 80% slowdown. Further, replacing the `nop` with `jo` leads to another 88% slowdown. Varying the length of each loop yields similar results, highlighting that extra branch instructions degrade performance of all stages in a pipeline.

Further, we analyze the impact of extra conditional branches on CPU pipeline in SPEC2017 benchmarks with a top-down analysis [59] on x86. Figure 6 shows the results of the analysis, which classifies all pipeline slots into four categories: front-end bound, back-end bound, bad speculation, and retiring. Retiring contributes most to the overhead, accounting for 50% of the increase in pipeline slots. Port contention in the back-end and branch mispredictions are also responsible for the overhead, accounting for 38% and 15% of all increased pipeline slots respectively. The front-end bandwidth is not a significant source of overhead, as none of the benchmarks are front-end bound. This analysis demonstrates that the primary driver of overhead is the large number of extra micro-ops required for overflow checking. In Section 5, we show how opX eliminates this source of overhead.

4 Eliminating Compiler Overheads with LCI

As discussed in Section 3, UBSan's current approach of using intrinsics to express overflow checked arithmetic in LLVM IR significantly hinders optimizations.

The core issue is that these intrinsics, along with the associated conditional branches, are introduced early in the compilation pipeline and hinder downstream optimization. This design decision is reasonable for a sanitizer whose goal is to detect language-level overflows as specified by the programming model and language semantics. However, our objective is different: We aim to detect overflows that actually occur at runtime and can lead to security vulnerabilities.

To address the limitations of UBSan, we introduce *late check insertion (LCI)*, a compiler-based approach that postpones the insertion of overflow checks until the end of the compiler middle-end, after all other optimizations have completed.

This design, however, introduces several challenges that we need to solve.

First, inserting overflow checks requires information to determine which operations must be checked and how those checks should be performed. In UBSan, this information is available in the front-end, when intrinsics are inserted when lowering to LLVM IR. At that stage, the compiler still retains high-level semantic information, such as whether an operation is signed or unsigned, whether annotations indicate that a check is required or can be omitted, whether the overflow is impossible to happen, and whether an overflow is intentional or follows a known safe pattern that does not require checking. Without UBSan, this information is lost after lowering to LLVM IR. The default IR provides no mechanism to indicate which operations should or should not be checked. More critically, LLVM IR does not explicitly distinguish between signed and unsigned integer types.

To address this challenge, we exploit metadata in LLVM IR to store the information. As illustrated in Figure 7, our approach adds metadata to each arithmetic instruction that needs to be checked in the LLVM front-end. Unlike UBSan's intrinsics, these annotations preserve the original arithmetic operations, allowing existing optimization passes to operate normally on the tagged instructions.

A second challenge is ensuring that the metadata is reliably propagated through the compilation pipeline so that it remains available at the final instrumentation pass. This is nontrivial because certain compiler optimizations can interfere with metadata propagation, for example by merging multiple IR instruction or introducing new arithmetic operations.

To address this issue, we perform a comprehensive analysis of LLVM optimization passes, identifying how metadata should behave under common transformations and designing

propagation rules that preserve the intended semantics across the middle-end. Metadata propagation has been used previously for purposes such as debug location [22], and LLVM already provides supporting infrastructure and interfaces for attaching and forwarding metadata across passes.

Metadata propagation is required when a pass replaces some IR instructions attached with metadata with new instructions, using methods like *replaceAllUsesWith*. The new instructions are often created without metadata, so we need to copy the metadata from the old instructions and attach it to the new ones.

Another important case is when LLVM determines that an arithmetic operation is guaranteed not to incur overflow and annotates the instruction with the no signed wrap (`nsw`) or no unsigned wrap (`nuw`) flags. In such cases, the metadata becomes unnecessary and can be safely removed. We rely on this metadata approach as it is less intrusive to patch passes to propagate or remove metadata correctly, than to update passes to optimize intrinsics. A complete list of our LLVM modifications and metadata propagation rules is presented in Table 2.

Our experiments in Section 7 demonstrate that this approach effectively eliminates the compiler-induced overheads quantified in Section 3.

Eliminating Compiler Overheads for the Rust Compiler. By default, Rust inserts overflow checks only in debug builds, as enabling them in release builds is considered too costly [45]. Even when enabled, checks are inserted only for operations that the Rust compiler cannot statically prove to be non-overflowing.

Rust uses a strategy similar to UBSan to generate overflow checks. Regular arithmetic instructions are replaced with overflow-checking intrinsics, which are lowered to LLVM overflow intrinsics (e.g., `llvm.sadd.with.overflow.i32`). A condtional branch is then inserted to invoke a panic handler generated by the Rust compiler when an overflow is detected. The resulting LLVM IR is the same as with UBSan, except the handler function that gets called.

Consequently, these checks introduce similar overheads and inhibit compiler optimizations in much the same way as UBSan intrinsics. To eliminate compiler overheads for the Rust overflow checks, we rely on the same technique. We reuse the Rust's built-in overflow checking to determine which operations require overflow checks. But instead of generating LLVM IR intrinsics and branches, we attach metadata to the arithmetic instructions. The rest of the compiler pipeline is the same as UBSan's metadata approach described above.

5 overflow protection eXtensions (opX)

Our primary goal is to provide overflow checked arithmetic at the same cost as unchecked arithmetic—enabling it to be on-by-default in production applications. After eliminating

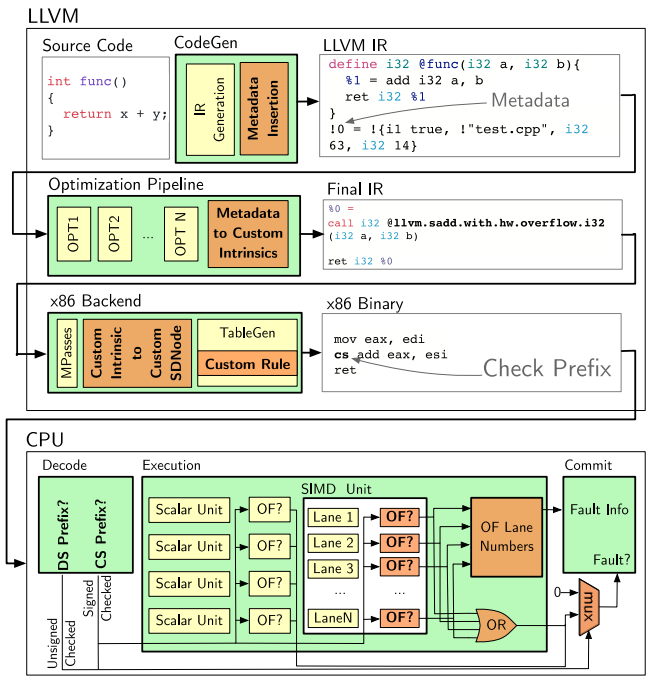

Figure 7: **Architecture Design Overview.** *opX lowers arithmetic operations that need to be checked to custom prefixes (x86). Then modified execution unit in CPU detects the variant and traps if overflow occurs.*

compiler-induced overhead using the approach described in Section 4, the remaining costs stem from architectural sources of overhead, as discussed in Section 3. To achieve our goal of default-on near-zero-cost integer overflow checking, we introduce a set of ISA extensions, called opX, that encode overflow semantics more efficiently. opX does so while introducing only minimal hardware cost and complexity, making the design practical for adoption in existing high-performance processors.

Efficiently Encoding Overflow Semantics As discussed, existing ISAs require a significantly greater number of instructions for checked versus unchecked arithmetic operations, this translates to more μops and lower throughput. This happens for a number of reasons:

1. **Lack of sign information:** ISAs do not always capture sign information in arithmetic instructions. For example, in an x86 `add` instruction, the type is only expressed in the branch instruction (e.g., `jc`), thus at least 2x instructions are required for checked compared to unchecked addition.
2. **Lack of overflow semantics:** Similarly "overflow should be checked" is not expressed in the arithmetic instruction requiring at least 2x more instructions for checked arithmetic (e.g., `jc`). Fine-grained overflow semantics are important as many arithmetic instructions do not need to be checked for overflow: For example, because the compiler has proved overflow cannot happen statically, because the

programmer is intentionally relying on wrap-around behavior for integers (either explicitly annotated, or because the compiler has recognized the pattern), or because there is some other mechanism in place to handle overflows for a particular use case, such as saturation in many SIMD operations.

3. **Branching overflow checks:** Most modern processors primarily support branch-on-overflow, dictating the need for at least two micro-ops, even with optimizations such as macro-op fusion or micro-op fusion, as a jump instruction (e.g., `jo` or `jc`) requires an additional 8-bit or 32-bit immediate, making fusion impractical due to increased storage demands for micro-ops.
4. **Missing overflow checks:** As discussed, most existing ISAs only sets an overflow flag for certain operations (ARM, x86), while others do not provide overflow checking at all (RISC-V). In either case, the result is that multiple instructions are needed for checked arithmetic.

Single Instruction Encoding. Our approach to addressing these challenge in opX is to provide an alternate set of arithmetic instructions, non-arithmetic instructions (for strength reduction), and SIMD instructions that are "checked". Each of these instructions has a signed and unsigned variant, indicating if a carry or overflow check is needed.

The potential issue this introduces is pressure on the instruction encoding space. In opX-x86, we do not actually create new instructions, instead we provide this with relative simplicity using legacy segment override prefixes (e.g., `CS`/`DS`).

Fusible Prefix Instructions. While a single instruction encoding is optimal, some of the benefits of opX could potentially be achieved by providing extended carry and overflow checks in a flag register and trap-on-overflow instructions such as SPARC's trap on integer condition code instructions [48]. Or x86-32 `INTO` instruction. While such an approach does not offer the front-end benefits of single-instruction encoding, as we showed in the previous section, most of the overhead of overflow checking is in the back-end of the pipeline—and fusing trap-on-overflow instructions with arithmetic instructions could offer similar benefits for reducing μop counts. This is the approach we take for ARM, as ARM already supports fusible prefix instructions (e.g., `MOVPRFX`).

Minimal New Instruction Set. In RISC-V, we do not have the luxury of instruction prefixes or prefix instructions. Therefore, for the reasons discussed in Section 5.3, our design relies on introducing a small number of new instructions. For non-SIMD arithmetic, this is straightforward, as only a limited set of checked instruction variants is required. Looking forward, RISC-V vector extensions are expected to move toward larger instruction encodings (e.g., 48-bit formats), which would provide additional encoding space. Extending the design to SIMD requires additional consideration. Not all SIMD instructions require overflow-checking semantics, since only a subset are typically generated by auto-vectorizers. Moreover, because most code is not vectorized, the highest impact comes from supporting overflow checks in arithmetic instructions and in operations used for strength reduction.

5.1 opX-x86

The x86 ISA has a variety of properties that made it easy to extend with opX—and emulate opX performance on existing hardware. To start, x86 has several legacy segment override prefixes that are no longer used in long mode (x86-64) [24]. Among these, we choose to use `CS` (0x2E) and `DS` (0x3E) segment override prefixes [3] — historically these were also used as hints to the branch predictor (*branch-taken* and *branch-not-taken*).

We use `CS` to denote instructions that must trap on overflow (signed), and `DS` to denote instructions that should trap on carry (unsigned). Using these is ideal, as it allows us evaluate opX's performance without modifying hardware. We initially used `rep` and `repne`, but they did not work with SIMD instructions.

Next, x86 processors already set the carry flag (CF) for unsigned overflow and overflow flag (OF) on signed overflow on most arithmetic operations of interest (e.g., `add`, `sub`, `mul`). Processor vendors could re-use this logic for checking our prefixes. Other instructions— `lea` which silently wraps on carry today, and `shl`, which currently stores that last bit shifted out in CF, instead of saturating—would require some additional logic.

5.2 opX-ARM

Unlike x86, ARM is a largely fixed-length instruction set architecture (ISA) and does not support prefix bytes for instructions. On the other hand, supporting both checked and unchecked variants of each arithmetic instruction would require duplicating a large portion of the instruction set, which is impractical.

However, ARM supports instruction fusion. The `MOVPRFX` instruction introduced in ARMv9 may be combined with the immediately following SVE instruction in program order to form a single operation [7]. One common use case is extending instructions such as `FMLA`, which natively support only three operands due to instruction encoding constraints, to effectively accept a fourth operand. In this scenario, `MOVPRFX` encodes the additional register, and the processor may either execute it as a regular vector move or fuse it with the subsequent instruction into a single four-operand micro-operation via front-end macro-op fusion.

Furthermore, our characterization study in Section 3, while presenting results for x86, shows that the majority of the

[3] These prefixes can be used without a memory operand. For example, `cs shl eax, 1` is a valid x86 instruction.

overhead introduced by additional instructions arises from back-end execution costs, rather than front-end effects such as fetch bandwidth, branch prediction, or instruction cache behavior. We observe similar trends on ARM platforms.

Consequently, encoding overflow-checking information via an additional instruction incurs minimal overhead, if the arithmetic instruction and the check instruction are fused into a single micro-op. This form of fusion is widely supported in modern ARM processors, making it a practical mechanism for implementing checked arithmetic.

Based on this observation, we propose a new instruction that checks for overflow in the preceding arithmetic instruction. This instruction reads the appropriate condition flags (OF for signed overflow and CF for unsigned overflow) and raises an exception if an overflow is detected. Conceptually, this instruction is similar to the legacy x86-32 `INTO` instruction. However, with instruction fusion support, the arithmetic operation and the overflow check are decoded into a single micro-operation, minimizing performance overhead.

5.3 opX-RISC-V

RISC-V does not have instruction prefixes and will not in the foreseeable future due to RISC philosophy. Instead, opX-RISC-V utilizes unused opcode space to create new custom instructions that check for overflow, and track the type (signed/unsigned) of the instruction.

These new instructions include checked versions of `add`, `mul`, and `sub`. The full list of the new instructions and their encoding is listed in the Appendix A.1 (Table 1).

opX is especially beneficial in RISC-V, as it does not have a flags register—a conscious decision to keep the pipeline simpler by avoiding the need to rename another register—and instead relies on a series of instructions to do an explicit overflow or carry check and then branch.

For example, for unsigned addition, a carry is detected by comparing the result of an `add` to one of its operands (using a `bltu`). Signed addition is trickier: overflow happens only when the operands share a sign and the result's sign flips, so software must compute and test those sign relationships before branching (using an extra `slt`).

This explodes very quickly for more complex operations such as multiplication. For example, for a 64-bit multiply, the hardware provides `mul` for the low half and `mulh`/`mulhu` for the high half, and overflow detection reduces to checking whether the high result matches the expected extension of the low result. This requires 4 instructions for signed multiplication and 3 for unsigned. Pleasantly, opX illustrates how RISC-V can have first class integer overflow checking, without the need for a flags register.

Design Rationale and Alternative Designs. Our design introduces checked version of arithmetic operations such as `add`, `sub`, and `mul` to support overflow detection. This puts pressure on instruction encoding space. We explored multiple design alternatives and conclude that this single-instruction encoding provides the most practical balance between deployability, architectural complexity, and compiler support.

One alternative is the use of instruction prefixes to encode overflow-checking behavior. However, RISC-V does not support instruction prefixes, and prior discussions have characterized prefix-based designs as a significant architectural escalation [61]. Introducing prefixes would complicate decoding and place additional burden on the front end, making this approach impractical for current RISC-V implementations.

Another option is a global mode flag, such as trap-on-overflow or trap-on-carry. While conceptually simple, this approach introduces substantial challenges for both hardware and software. Mode flags increase compiler complexity and require careful management across control flow. In hardware, such flags either require serialization, which degrades performance, or register renaming, which incurs nontrivial silicon cost. For these reasons, similar designs have been explicitly avoided in prior RISC-V proposals.

Historically single-instruction trap semantics were supported in RISC architectures for signed arithmetic, including MIPS and PA-RISC as discussed in Section 1, demonstrating the feasibility of this design approach.

In addition, RISC-V may gain additional encoding space as it transitions to longer instructions in the future [43], which would entirely alleviate current instruction encoding constraints. In the near term, to further reduce encoding pressure, we omit checked variants for certain instruction forms, such as immediate operations, and limit support for overflow detection in SIMD instructions.

5.4 Microarchitecture and Hardware Costs

opX requires only minimal changes to the microarchitecture of existing x86, RISC-V and ARM processors. Almost all of the necessary circuitry already exists: functional units already compute carry and overflow signals for arithmetic operations on x86, and on RISC-V and ARM the required logic consists of only a handful of gates.

In brief, opX needs the following changes to a typical pipeline.

- **Overflow Computation:** x86 and ARM ALUs already compute overflows, adding support to RISC-V requires a few gates. For unsigned it is the carry out of a full adder, for signed it is the XOR of carry-in/carry-out of the last bit.
- **SIMD Overflow Checks:** For SIMD execution, opX adds overflow logic per lane. Like normal ALU checks, per-lane SIMD checks are a handful of gates. This is a modest addition. Similar support already exists in ARM SIMD units for saturating arithmetic.
- **Decode Logic:** We add only a small number of additional decode terms and two extra bits in the micro-op/control-signal space to encode signedness and whether overflow

should be checked (on the order of a few hundred basic gates in a PLA-style decoder), for even a small RV32 microcontroller (order of tens of thousands of gates) this is still under 0.1%. In a more substantial core (100K+ to 1M+) it is vanishingly small.

- **Extra State:** opX needs a few bits of state to carry the exception along the pipeline, but other than that it does not require memory storage.

One subtle effect of opX is its interaction with grouped reorder buffers (ROBs) [39]. Grouped ROB is a microarchitectural optimization that improves commit bandwidth by aggregating instructions that are guaranteed not to change control flow or raise exceptions into a single ROB entry. Since opX expands the set of instructions that can trap, it reduces opportunities for such grouping, and thus may diminish the effectiveness of this optimization.

5.5 Compiling for opX

Similar to the approach described in Section 4, we use LCI for opX. In the final pass, instead of generating UBSan checks, we convert metadata-attached instructions into special intrinsics. We then modified the LLVM RISC-V, x86 and ARM backends to lower these intrinsics to opX-RISC-V, opX-x86, and opX-ARM.

6 Experimental Methodology

Platform. We perform our x86 measurements on a server with a 28-core Intel(R) Xeon(R) Gold 5512U CPU on CloudLab [15]. Hyperthreading and ASLR were disabled, and the core frequency fixed to 3.0 GHz. Our RISC-V evaluation use a VisionFive2 board equipped with a 4-core StarFive JH7110 SoC (RV64GC) with 8 GB RAM. For ARM evaluations, an Ampere dual-socket server with two 80-core Altra AC-108021002P @3.0 GHz processors (Neoverse N1) is used.

Benchmark Applications. We use SPEC CPU 2017 benchmarks v1.1.9 [49] for our C/C++ performance measurements. Our modified compiler is based on LLVM 19.1.0. The UBSan options we use for C/C++ are `-fsanitize=signed-integer-overflow,unsigned-integer-overflow`. Enabling more checks (e.g., with `-fsanitize=shift`) incurs additional performance costs for our baseline UBSan. We focus on C/C++ benchmarks in SPECInt, as FP benchmarks see limited performance impact from integer overflow checks.

We also use a selection of real-world applications to enrich the types of benchmarks that we evaluate. These include *coremark* [16], multimedia programs (*aac* [17], *opus* [56], *vpx* [19], *hevc* [27]), compression (*zstd* [58]), and an XML library (*expat* [28]). We also test 3 popular databases: SQLite, RocksDB and Redis, with results in Appendix A.3.

As it happens, SPEC 2017, contains many instances of signed and unsigned integer overflows — around 1.5% of all dynamic arithmetic instructions. We are interested in the costs of the checked, rather than the costs of handling these checks, which should be rare in production. To this end, we provide UBSan with the locations of all arithmetic operations known to overflow and exclude them from checking, similar to [10], unless otherwise noted.

We use a Rust port of the NAS parallel benchmarks (NPB-Rust) [34]. We also include select benchmarks from Hashbrown [21], a widely-used Rust port of Google's Swisss Tables [8] hash table library, as a real-world application. We select *set_ops*, *grow*, and *lookup* from Hashbrown, and the *EP*, *CG*, *FT*, and *IS* kernels from NPB-Rust. Our modified Rust compiler is based on Rust 1.86.0.

All benchmarks run at least 8 times and we report both mean and standard deviation via error bars. All tests on RISC-V and ARM do not enable auto-vectorization.

Emulation and Simulation. We implement both opX-x86 and opX-RISCV in the gem5 simulator v24.1 [33] using its cycle-accurate out-of-order CPU model. For RISC-V, we match the custom opcodes in the decoder and perform corresponding signed/unsigned overflow checks in the execution stage. If an overflow occurs, an exception will be raised and terminate the program. For x86, we modified the decoder to recognize the prefix. The gem5 simulator already implemented overflow detection in execution, and raising/handling exceptions.

However, since we can evaluate the performance of our extensions with much higher accuracy and speed using compiler-based emulation, we do not report performance results of gem5. We use gem5 for functional validation, and we run microbenchmarks on gem5. The performance results are consistent with the results from Listing 5, confirming that our prefixes show minimal performance overhead.

We evaluate the performance of opX-x86 by compiling to a binary that uses `CS`/`DS` prefixes in arithmetic instructions for overflow semantics. This approach allows us to measure the overhead of the prefixes, including the pressure on the front-end. Since the hardware implementation of a checked arithmetic would execute in the same number of cycles as unchecked ones and existing hardware simply ignores these prefixes, running such binaries on real processors accurately emulates the performance of a opX-x86 implementation.

Similarly, for RISC-V, we simply make the compiler emit the original opcodes instead of our custom ones when evaluating performance, since the mostly fixed-length nature of RISC-V instructions should not incur additional overhead.

For ARM, the added check instruction in our design has the following expected performance characteristics: (1) it depends on the condition flag register, (2) if fused with the preceding arithmetic instruction, it does not consume additional ROB or execution slots, and (3) it may raise an exception on overflow. To estimate the performance impact

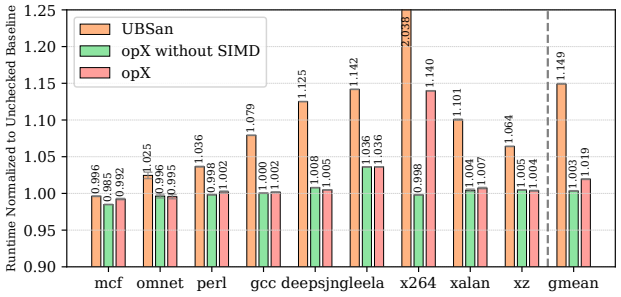

Figure 8: **Performance of opX-x86 vs. UBSan on SPEC 2017.** *opX achieves near-zero-cost overflow checks (0.3% without SIMD and 1.9% with SIMD checks) compared to 14.9% of UBSan.*

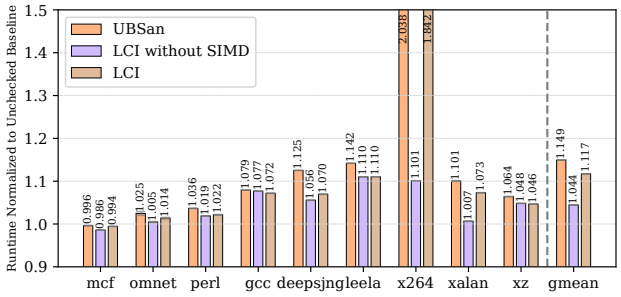

Figure 9: **Performance of LCI vs UBSan on x86.** *Software checks with LCI achieve 4.4% (11.7% with SIMD checks) overhead, much lower than UBSan.*

of such a prefix-style instruction, we evaluate two proxy instructions: `nop` and `csel`. We use `csel x1, x1, x1, eq`, which reads the condition flags and leaves the architectural state unchanged, but still consumes execution resources and ROB entries. We also run a microbenchmark with `csel x1, x1, x1, eq` and `csel x1, x2, x3, eq`, which shows no performance difference between them, suggesting no microarchitectural optimization for same source operands. In contrast, `nop` occupies a ROB entry but does not use an execution slot. As a result, `csel` provides an upper-bound estimate of the performance cost of our proposed check instruction. In all configurations SIMD overflow checks are disabled.

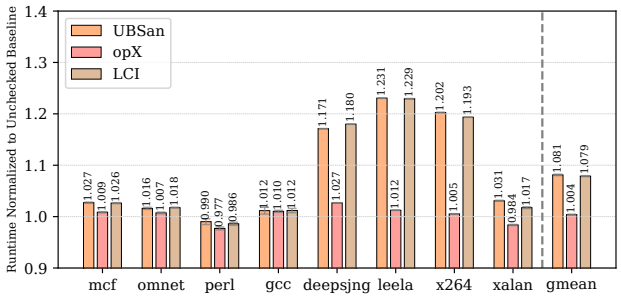

Figure 10: **opX-RISC-V vs. UBSan On SPEC 2017.** *opX-RISC-V achieves overhead of only 0.38% geomean, far lower than 8.1% by UBSan.*

7 Evaluation

7.1 opX's Performance on SPEC 2017

We compare the performance overhead of: our hardware based overflow checks (§4) using **opX-x86**, **opX-RISC-V** and **opX-ARM** (§5); standard software overflow checks (e.g., `jo`, `jc`) using our **LCI** approach; normal **UBSan**; and **Baseline** compiled with no checks.

opX-x86 vs. UBSan. As Figure 8 shows opX-x86 offers near-zero-cost overflow checks with 0.3%-1.9% geomean overhead depending on whether checks are enabled for SIMD instructions. We can break down this performance improvement by looking at the 8.4% overhead with Hollow Intrinsics (intrinsics without branches) from Figure 2. From this we can infer that roughly 60% of the total performance improvement comes from the fact that opX-x86 does not break compiler optimizations, while the other 40% is due to the reduction in branch instructions issued for every arithmetic operation.

Looking into individual benchmarks, *x264* shows the most remarkable improvement, with overhead dropping from 2.04× with UBSan, to near zero (14% with SIMD checks) with opX.

LCI vs. UBSan. Figure 9 shows the performance of our software-only LCI approach, which uses existing instructions for overflow checking (e.g., `jc`, `jo`). Our LCI approach reduces the UBSan's 14.9% overhead to 11.7%. Disabling overflow checks for SIMD instructions further reduces LCI's overhead to 4.4%. Unlike opX, our software-only LCI does not enjoy the benefits of native overflow-checked SIMD instructions, thus it must emit explicit branches to check for overflow, reducing overall benefits of vectorization. However, omitting checks for SIMD instructions may still be reasonable, since much security-critical code is not vectorized (e.g., code manipulating indexs, pointers, etc.). These results suggest that UBSan could benefit from an improved implementation, even with current hardware.

opX-RISC-V vs. UBSan. Compared to x86, UBSan on RISC-V shows significantly lower overhead across the board, likely due to our RISC-V board does not support SIMD. Thus, the fact that UBSan breaks vectorization has no impact. opX-RISC-V imposes 0.38% geomean overhead, notably lower than geomean 8.1% of UBSan. For individual benchmarks, *deepsjeng*, *leela* and *x264* show notable improvement. Here, opX manages near zero overhead compared to 17% to 23% with UBSan. The software-only LCI version performs similar to UBSan.

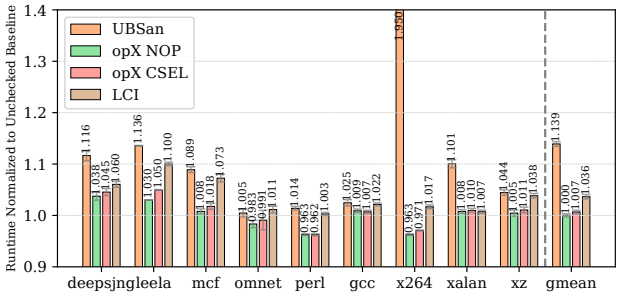

Figure 11: **opX-ARM vs. UBSan On SPEC 2017.** *opX-ARM adhesives overhead of only 0.03% (`nop`) and 0.67% (`csel`) geomean, far lower than 13.9% that of UBSan. Software checks with LCI achieves 3.6% overhead.*

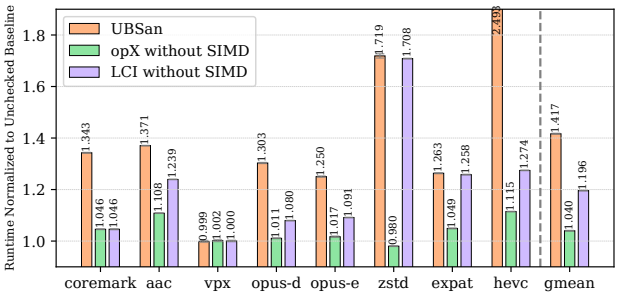

Figure 12: **Popular libraries and coremark on x86.** *opX achieves 4% overall overhead compared to UBSan's 41.7%. The software-only LCI version also achieves 19.6% overhead.*

opX-ARM vs. UBSan. We use `nop` and `csel` to simulate the overhead of instruction fusion. `nop` provides a lower bound while `csel` is a more accurate estimation. They show 0.03% and 0.67% overhead geomean respectively, significantly lower than 13.9% of UBSan. *leela* and *x264* show notable improvement while *deepsjeng* is less so. opX also achieves near-zero overhead on ARM. Our software-only LCI approach results in 3.6% overhead geomean.

7.2 Performance of Additional Benchmarks

Figure 12 compares the performance impact of opX-x86 and UBSan on a selection of popular libraries, most of them Android ships with UBSan overflow checking enabled [6] (*hevc*, *opus*, *expat*, *acc*, *vpx*), and *coremark*.

opX reduces geomean overhead from UBSan's 41.7% to 4%. We see notable overhead reduction with opX in most tests, suggesting the design generalizes well from SPEC2017 to other benchmarks. However, in *hevc* opX still has a rather high (11.5%) overhead. The software only LCI version also performs well in most benchmarks at 19.6% overhead, but still much higher than opX, highlighting the benefits of a hardware-software co-design. With over 3 billion android devices world wide, it's interesting to contemplate the amount of battery life consumed by the UBSan tax on XML parsing and media decoding.

7.3 Rust Performance

Figure 13 shows Rust performance. We see that opX-x86 incurs near zero overhead compared to 3.5% with standard overflow checks. Software checks using LCI also incur near zero overhead.

7.4 Metadata Coverage Analysis

To validate that we are not sacrificing completeness, we use UBSan's overflow handlers to print overflow locations with

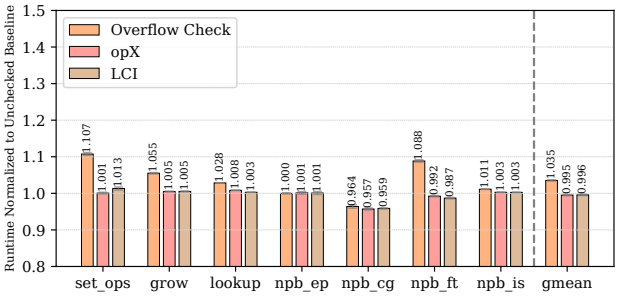

Figure 13: **Hashbrown benchmark and NPB-Rust on x86.** *opX achieves near zero overhead compared to 3.5% of Rust builtin overflow checks. The software-only LCI version also achieves near zero overhead.*

both stock UBSan and our LCI approach, and compare coverage. We achieve 100% coverage on SPECInt benchmarks after ignoring some safe-to-ignore cases:

- Aggressive instruction combining transforms bit manipulation implemented using arithmetic with magic values (which could cause overflow), into specialized LLVM intrinsics (which cannot overflow). Examples include transforming unsigned arithmetic (can overflow) to bit population count (popcnt) and counting trailing zeros (ctz) (which cannot overflow).
- When enabling auto-vectorization, if multiple elements overflow, the UBSan handler only reports one (usually the first element), causing a mismatch between the coverage reported by scalar (UBSan) and SIMD (opX) versions.
- Common subexpression elimination (CSE) causes only the first occurrence to be evaluated; thus, overflows in the same expression are not reported. UBSan usually prevents most CSE optimizations.
- Some loop induction operations are optimized out. A common example is the pattern `while (n--)` where `n` is unsigned. In this example, `0-1` is simply not evaluated after optimization because it is not used, therefore no overflow

occurs at runtime.

- Instruction Combine and Reassociation passes may remove overflows. For example, `-x-1` is transformed into `x^(-1)`.
- Bitwise left shift (shl) checks are not enabled by our UBSan options, so they are not reported in UBSan coverage. However, we do check them in hardware-based opX if they are emitted from strength reduction.
- We report overflows in constant folding during compilation, so they do not show up in runtime coverage.

In all such cases, the overflow is either eliminated by compiler optimizations and therefore does not pose a security risk, or it remains detectable, but with imprecise location reporting which is sufficient for our mitigation purposes.

7.5 Security Evaluation

CWE-190 (Integer Overflow or Wraparound) [3] provides a list of selected observed examples of CVEs. There are 18 CVEs in CWE-190 examples. We reproduced 15 unique cases, with the remaining 3 irrelevant (CVE-2022-21668), close-sourced (CVE-2021-30663) or redundant (CVE-2004-2013). We use Alive2 [32] to automatically verify equivalence of Metadata. The tool can verfy the equivalance of our LCI approach and UBSan on 11 out of 15 cases. The remaining ones fail due to underlying SMT solver errors. For those cases where the verifier failed, we manually verified that UBSan and LCI reach equivalent coverage.

7.6 Memory and Code Size Comparison

We compare the memory and code size of opX-x86 and UBSan. We measure the code size by comparing the size of `.text` segment size. Both have UBSan library statically linked, so around 201 KB is substracted from both. While memory usage of both are similar, opX-x86 has significantly lower code size. opX-x86 increases the code size by only 3% geomean compared to 34% that of UBSan. Interestingly, opX-x86 version of *mcf* is actually 17% smaller than the baseline, which could explain its higher performance than baseline.

8 Related Work

ISA Support for Single-instruction Checked Arithmetic. Historically, several architectures provided efficient single-instruction integer overflow checking. For example, early MIPS ISAs defined arithmetic instructions such as ADD that trap on signed overflow by default, raising an exception when overflow occurs [36]. As C became the dominant systems programming language, however, this default behavior was a mismatch with C's wraparound and undefined-overflow semantics. To accommodate C workloads efficiently, MIPS introduced separate non-trapping variants (e.g., ADDU) that perform the same arithmetic without raising exceptions. Similar trap-on-overflow instruction semantics were also present in other architectures, including Alpha and PA-RISC, and aligned well with languages such as Ada and Pascal that require overflow trapping. The dominance of PC-class architectures and C-family language semantics shifted mainstream ISAs away from default trapping arithmetic, making overflow checking significantly more expensive than unchecked arithmetic in modern systems.

Software-based Overflow Detection. Research on integer overflow detection can be broadly categorized into static and dynamic analysis techniques. Static analysis [29,31,37,38,46, 51,52,54,55,57], which rely on symbolic execution and taint analysis to identify potential integer overflows. While these methods do not introduce runtime overhead, they frequently suffer from a high rate of misclassifications and scalability issues. For example, KUBO [29] strives to achieve both accuracy and scalability, but has a 27.5% false positive rate after 33 hours' analysis on Linux kernel.

To achieve better accuracy, many dynamic approaches [10, 11, 14, 40, 50, 53, 62] are proposed, which collect information during program execution to detect overflow behaviors in real-time. However, they often lead to huge performance degradation. For example, IOC [14] slows down 21 benchmarks in SPEC2006 by 51% on average, while IntFlow [40] reports a 10× slowdown on *gzip* in SPEC CPU2000. While INDIO and IntPatch report smaller overheads (less than 1% on SPEC CINT2000 and real-world applications), they are only applicable to a subset of integer overflows.

Hardware-based Overflow Detection. Prior research has explored hardware-based mechanisms for detecting arithmetic overflows. Mihocka et al. [35] demonstrated that all six IA-32 arithmetic flags (such as OF and CF) can be generated from the result and carry-out vector of addition or subtraction operations, which also aids in overflow detection for SIMD instructions. They further proposed an x86 ISA extension for carry-out vector generation to accelerate software emulation. Regehr [42] similarly advocated for hardware support to trap integer overflows in both x86 and ARM, arguing that the performance cost would be minimal while providing important safety guarantees.

9 Conclusion

Integer overflow is a textbook example of how systems fail due to a disconnection between requirements at different levels of the stack. At the language level, it's clear that integer overflow is a major source of vulnerabilities.

At the compiler level, overflow checks are inefficient to encode, and not compatible with certain critical optimizations due to hardware limitations, leading users to (correctly) perceive that overflow checks are slow, and quite predictably they turn them off in production.

Processor architectures further this situation by making it impossible to compile integer overflow checks efficiently, but the problem receives little attention because users are not compiling with integer overflow checks in production and the cycle continues.

Small compiler and hardware changes can break this cycle, enabling near-zero-cost overflow checks. This moves us towards an integer overflow end game, a future without silent integer overflow bugs, and unmitigated integer overflow vulnerabilities.

Acknowledgments

This work was supported by a gift from Ampere Computing. We thank our reviewers. And finally thanks to our families, without their support this work would not be possible.

Ethical Considerations

Stakeholder Identification and Impacts

We conducted a stakeholder-based ethics analysis to evaluate the broader impacts of developing and disclosing the opX architecture. We identify the following key stakeholders:

Society at Large and End-Users. The primary beneficiaries of this research. During the research procedure, these groups were unaffected as all tests were simulated. Upon publication, they are positively impacted by the potential for systems with default-on integer overflow protection, tangibly reducing the risk of catastrophic failures (e.g., Ariane 5) and remote exploitation (e.g., Stagefright-style attacks).

Software Developers and System Administrators. Publication provides these stakeholders with a pathway to enforce integer safety without the traditional 15% performance penalty, resolving the long-standing tension between security and throughput.

Compiler Maintainers and Hardware Architects. We recognize that proposing architectural and compiler modifications (such as LLVM metadata-related passes and new RISC-V/x86 encodings) places a professional burden on these entities. The publication of this research may require them to review, debate, and potentially maintain these implementations.

The Research Team. Impacted procedurally through the investment of time and resources, and professionally through the execution of this work.

Ethical Principles and Harms

Our analysis is grounded in the principles of the Menlo Report:

Beneficence. The driving ethical principle of this work. By reducing overflow mitigation overhead from 15% to 1%, we aim to maximize systemic security benefits and minimize the risk of unmitigated vulnerabilities in production systems.

Respect for Persons and Human Rights. Our research procedures strictly adhered to ethics regarding human rights. No human subjects were involved, no user data was collected, and no live production systems were targeted.

Potential Harms. We identify one potential tangible harm: the dual-use nature of security research. While this paper focuses on defense, detailing the exact mechanics of how existing and future compilers and hardware handle (or fail to handle) overflow checks could theoretically assist an adversary in identifying unmitigated overflows in software or hardware.

Mitigations

To address the potential harms and burdens identified above, we took the following mitigating steps:

Mitigating Dual-Use Harm. We restricted our evaluation entirely to historical, publicly disclosed, and heavily patched vulnerabilities (e.g., from the CWE-190 database). We did not discover, exploit, or disclose any new zero-day vulnerabilities during this research.

Mitigating Maintainer Burden. To respect the time and resources of the open-source community, we designed opX to be minimally invasive, utilizing existing metadata infrastructure rather than overhauling the LLVM middle-end, and relying on legacy prefixes in x86.

Decision to Proceed and Publish

The decision to undertake this research was driven by the persistent, systemic failure of software-only integer overflow mitigations.

The tangible benefits of establishing a near-zero cost mitigation for a vulnerability class that has caused massive financial and safety damage over the last three decades vastly outweigh the theoretical dual-use risks and the professional burdens placed on system maintainers. Furthermore, by open-sourcing our compiler toolchain and simulator artifacts, we empower defenders to deploy and evaluate these mitigations immediately, ensuring that the defensive benefits scale faster than any potential adversarial advantage.

Open Science

To support the replicability of our results and foster future research into integer overflow mitigation, we have made our complete artifact suite available [4]. It contains the following components:

[4] https://doi.org/10.5281/zenodo.20317245

- The modified LLVM compiler toolchain (including the Metadata and opX passes) based on LLVM 19.1.0.
- The gem5 source code patched with the opX instruction set extensions and configuration files.
- The patched lfi-bench for popular libraries and databases.
- The modified Rust compiler based on Rust 1.86.0.
- Scripts used to build and generate the results and other useful files.

These can reproduce the results in Section 7.

References

[1] Report by the inquiry board on the ariane 5 flight 501 failure. Technical report, European Space Agency, 1996. URL: https://en.wikipedia.org/wiki/Ariane_flight_V88.

[2] CVE-2022-23090: FreeBSD AIO credential reference count overflow leads to use-after-free. https://vuxml.freebsd.org/freebsd/5ddbe47b-1891-11ed-9b22-002590c1f29c.html, 2022. The aio_aqueue path in lio_listio fails to release a credential reference on error, allowing an integer reference-count overflow and resulting in a use-after-free (temporal safety violation).

[3] Cwe-190: Integer overflow or wraparound (4.17), 2025. URL: https://cwe.mitre.org/data/definitions/190.html.

[4] Search results for "integer overflow" on cve.org. https://www.cve.org/CVERecord/SearchResults?query=integer+overflow, 2025. [Accessed: 2025-08-20].

[5] Andreas Abel and Jan Reineke. uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 673–686, April 2019. arXiv:1810.04610 [cs]. URL: http://arxiv.org/abs/1810.04610, doi:10.1145/3297858.3304062.

[6] Android Open Source Project. Android open source project: Android.bp integer-overflow sanitizers (aac, hevc, vpx, expat), 2025. See Android.bp file of libraries in /external for UBSan settings. URL: https://android.googlesource.com.

[7] Arm Ltd. *Arm Cortex-A720 Core Software Optimization Guide*. Arm Ltd., issue 7.0 edition, 2023. URL: https://developer.arm.com/documentation/109720/latest/.

[8] Sam Benzaquen, Alkis Evlogimenos, Matt Kulukundis, and Roman Perepelitsa. Swiss tables and absl::hash. Abseil Blog, 2018. URL: https://abseil.io/blog/20180927-swisstables.

[9] Joshua Bloch. Extra, Extra—Read All About It: Nearly All Binary Searches and Mergesorts Are Broken. Google Research Blog, June 2006. Blog post, June 2, 2006. URL: https://research.google/blog/extra-extra-read-all-about-it-nearly-all-binary-searches-and-mergesorts-are-broken/.

[10] David Brumley, Tzi-cker Chiueh, Robert Johnson, Huijia Lin, and Dawn Song. Rich: Automatically protecting against integer-based vulnerabilities. 2007.

[11] Ping Chen, Hao Han, Yi Wang, Xiaobin Shen, Xinchun Yin, Bing Mao, and Li Xie. IntFinder: Automatically Detecting Integer Bugs in x86 Binary Program. In *Information and Communications Security: 11th International Conference, ICICS 2009, Beijing, China, December 14-17, 2009. Proceedings*, pages 336–345, Berlin, Heidelberg, December 2009. Springer-Verlag. doi:10.1007/978-3-642-11145-7_26.

[12] Clang/LLVM Project. Undefinedbehaviorsanitizer — clang documentation, 2025. URL: https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.

[13] Kees Cook. Mitigating integer overflow in c. YouTube, 2024. Accessed: 2025-08-20. URL: https://www.youtube.com/watch?v=PLcZkgHCk90.

[14] Will Dietz, Peng Li, John Regehr, and Vikram Adve. Understanding integer overflow in c/c++. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 25(1):1–29, 2015.

[15] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019. URL: https://www.flux.utah.edu/paper/duplyakin-atc19.

[16] Embedded Microprocessor Benchmark Consortium (EEMBC). Coremark® cpu benchmark. https://www.eembc.org/coremark/, 2009.

[17] Fraunhofer IIS. Fraunhofer fdk aac codec library. https://github.com/mstorsjo/fdk-aac, 2012.

[18] Jessica Gentles, Mason Fields, Garrett Goodman, and Suman Bhunia. Breaking the vault: A case study of the 2022 lastpass data breach. 2025. URL: http://www.arxiv.org/abs/2502.04287, arXiv:2502.04287.

[19] Google Inc. and Alliance for Open Media. libvpx: Vp8/vp9 codec sdk. https://chromium.googlesource.com/webm/libvpx/, 2010.

[20] Harvard University Research. An in-depth analysis of the stagefright bugs. In *Harvard DASH Repository*, 2017. URL: https://dash.harvard.edu/server/api/core/bitstreams/8efa6308-aac2-43e0-91cf-aed9ef10e6b4/content.

[21] Hashbrown contributors. Hashbrown: Rust port of google's swisstable hash map. https://github.com/rust-lang/hashbrown, 2025.

[22] Shan Huang, Jingjing Liang, Ting Su, and Qirun Zhang. Robustifying debug information updates in llvm via control-flow conformance analysis. *Proceedings of the ACM on Programming Languages*, 9(PLDI):527–549, 2025.

[23] Apple Inc. *The Swift Programming Language: Advanced Operators*. Apple, 2025. URL: https://docs.swift.org/swift-book/documentation/the-swift-programming-language/advancedoperators/.

[24] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture*, june 2021 edition, 2021. URL: https://www.intel.com/content/www/us/en/develop/documentation/64-ia-32-architectures-software-developer-vol-1-manual.html.

[25] ISO/IEC. Iso/iec 9899:2018: Information technology — programming languages — c, 2018. URL: https://www.iso.org/standard/74528.html.

[26] ISO/IEC. Iso/iec 14882:2020: Programming languages — c++, 2020. URL: https://www.iso.org/standard/79358.html.

[27] Ittiam Systems Pvt Ltd. libhevc: Open-source hevc decoder. https://github.com/ittiam-systems/libhevc, 2012.

[28] libexpat developers. Expat xml parser library. https://github.com/libexpat/libexpat, 1997.

[29] Changming Liu, Yaohui Chen, and Long Lu. KUBO: Precise and Scalable Detection of User-triggerable Undefined Behavior Bugs in OS Kernel. In *Proceedings 2021 Network and Distributed System Security Symposium*, Virtual, 2021. Internet Society. URL: https://www.ndss-symposium.org/wp-content/uploads/ndss2021_1B-5_24461_paper.pdf, doi:10.14722/ndss.2021.24461.

[30] LLVM Project. *Auto-Vectorization in LLVM*. LLVM Project, 2025. URL: https://llvm.org/docs/Vectorizers.html.

[31] Fan Long, Stelios Sidiroglou-Douskos, Deokhwan Kim, and Martin Rinard. Sound input filter generation for integer overflow errors. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 439–452, San Diego California USA, January 2014. ACM. URL: https://dl.acm.org/doi/10.1145/2535838.2535888, doi:10.1145/2535838.2535888.

[32] Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. Alive2: bounded translation validation for llvm. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 65–79, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3453483.3454030.

[33] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, Gabe Black, Gedare Bloom, Bobby R. Bruce, Daniel Rodrigues Carvalho, Jeronimo Castrillon, Lizhong Chen, Nicolas Derumigny, Stephan Diestelhorst, Wendy Elsasser, Carlos Escuin, Marjan Fariborz, Amin Farmahini-Farahani, Pouya Fotouhi, Ryan Gambord, Jayneel Gandhi, Dibakar Gope, Thomas Grass, Anthony Gutierrez, Bagus Hanindhito, Andreas Hansson, Swapnil Haria, et al. The gem5 simulator: Version 20.0+, 2020. URL: https://arxiv.org/abs/2007.03152, doi:10.48550/ARXIV.2007.03152.

[34] Eduardo M. Martins, Leonardo G. Faé, Renato B. Hoffmann, Lucas S. Bianchessi, and Dalvan Griebler. Npb-rust: Nas parallel benchmarks in rust, 2025. URL: https://arxiv.org/abs/2502.15536, arXiv:2502.15536.

[35] Darek Mihocka and Jens Troeger. A proposal for hardware-assisted arithmetic overflow detection for array and bitfield operations. In *WISH—Workshop on Infrastructures for Software/Hardware Co-Design*. Citeseer, 2010.

[36] MIPS Technologies, Inc. *MIPS IV Instruction Set*, revision 3.2 edition, September 1995.

[37] Paul Muntean, Jens Grossklags, and Claudia Eckert. Practical Integer Overflow Prevention, November 2017. arXiv:1710.03720 [cs]. URL: http://arxiv.org/abs/1710.03720, doi:10.48550/arXiv.1710.03720.

[38] Paul Muntean, Martin Monperrus, Hao Sun, Jens Grossklags, and Claudia Eckert. IntRepair: Informed Repairing of Integer Overflows. *IEEE Transactions on Software Engineering*, 47(10):2225–2241, October 2021. arXiv:1807.05092 [cs]. URL: http://arxiv.org/abs/1807.05092, doi:10.1109/tse.2019.2946148.

[39] José R. García Ordaz, Marco A. Ramírez Salinas, Luis A. Villa Vargas, Herón Molina Lozano, and Cuauhtémoc Peredo Macías. A reorder buffer design for high performance processors. *Computación y Sistemas*, 16(1):15–25, 2012. URL: https://www.scielo.org.mx/pdf/cys/v16n1/v16n1a3.pdf.

[40] Marios Pomonis, Theofilos Petsios, Kangkook Jee, Michalis Polychronakis, and Angelos D. Keromytis. IntFlow: improving the accuracy of arithmetic error detection using information flow tracking. In *Proceedings of the 30th Annual Computer Security Applications Conference*, ACSAC '14, pages 416–425, New York, NY, USA, December 2014. Association for Computing Machinery. URL: https://dl.acm.org/doi/10.1145/2664243.2664282, doi:10.1145/2664243.2664282.

[41] Robert Ramey. Safe numerics. Boost C++ Libraries Documentation (version 1.89.0), 2012–2018. Online; accessed 20 August 2025. URL: https://www.boost.org/doc/libs/1_89_0/libs/safe_numerics/doc/html/index.html.

[42] John Regehr. We need hardware traps for integer overflow, 2014. Accessed: 2025-07-20. URL: https://blog.regehr.org/archives/1154.

[43] RISC-V International. Long instruction tg proposal of work. Technical Committee documentation for RVG-652. URL: https://riscv.atlassian.net/wiki/external/MzU0MzMyYTJmNTBhNGQwY2IwMThiNzZkZGFkZjkzZGY.

[44] Rust Language Community. *The Rust Programming Language (Data Types)*. Rust Project Developers, 2024. URL: https://doc.rust-lang.org/book/ch03-02-data-types.html.

[45] Rust Language Team. Rfc 560: Integer overflow. https://github.com/rust-lang/rfcs/blob/master/text/0560-integer-overflow.md, 2026. Accessed: 2026-02-05.

[46] Stelios Sidiroglou-Douskos, Eric Lahtinen, Nathan Rittenhouse, Paolo Piselli, Fan Long, Deokhwan Kim, and Martin Rinard. Targeted Automatic Integer Overflow Discovery Using Goal-Directed Conditional Branch Enforcement. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 473–486, New York, NY, USA, March 2015. Association for Computing Machinery. URL: https://dl.acm.org/doi/10.1145/2694344.2694389, doi:10.1145/2694344.2694389.

[47] Manish Singh. Nso group exploited whatsapp to install pegasus spyware even after lawsuit. *The Hacker News*, 2024. URL: https://thehackernews.com/2024/11/nso-group-exploited-whatsapp-to-install.html.

[48] Inc. SPARC International. *The SPARC Architecture Manual: Version 8*, circa 2000. Includes `Tcc` instruction variants such as `TCS` and `TCC`.

[49] Standard Performance Evaluation Corporation (SPEC). Spec cpu®2017 benchmark suite. https://www.spec.org/cpu2017/, 2022.

[50] Hao Sun, Xiangyu Zhang, Chao Su, and Qingkai Zeng. Efficient dynamic tracking technique for detecting integer-overflow-to-buffer-overflow vulnerability. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '15, page 483–494, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2714576.2714605.

[51] Hao Sun, Xiangyu Zhang, Yunhui Zheng, and Qingkai Zeng. IntEQ: Recognizing Benign Integer Overflows via Equivalence Checking across Multiple Precisions. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 1051–1062, May 2016. ISSN: 1558-1225. URL: https://ieeexplore.ieee.org/document/7886979, doi:10.1145/2884781.2884820.

[52] Christof Ferreira Torres, Julian Schütte, and Radu State. Osiris: Hunting for Integer Bugs in Ethereum Smart Contracts. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 664–676, San Juan PR USA, December 2018. ACM. URL: https://dl.acm.org/doi/10.1145/3274694.3274737, doi:10.1145/3274694.3274737.

[53] Tielei Wang, Chengyu Song, and Wenke Lee. Diagnosis and Emergency Patch Generation for Integer Overflow Exploits. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Alfred Kobsa, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Demetri Terzopoulos, Doug Tygar, Gerhard Weikum, and Sven Dietrich, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, volume 8550, pages 255–275. Springer International Publishing, Cham, 2014. Series Title: Lecture Notes in Computer Science. URL: http:

//link.springer.com/10.1007/978-3-319-08509-8_14, doi:10.1007/978-3-319-08509-8_14.

[54] Tielei Wang, Tao Wei, Zhiqiang Lin, and Wei Zou. IntScope: Automatically Detecting Integer Overflow Vulnerability in X86 Binary Using Symbolic Execution.

[55] Xi Wang, Haogang Chen, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. Improving integer security for systems with KINT. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 163–177, Hollywood, CA, October 2012. USENIX Association. URL: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/wang.

[56] Xiph.Org Foundation and contributors. Opus interactive audio codec. https://opus-codec.org, 2012.

[57] Lili Xu, Mingjie Xu, Feng Li, and Wei Huo. ELAID: detecting integer-Overflow-to-Buffer-Overflow vulnerabilities by light-weight and accurate static analysis. *Cybersecurity*, 3(1):18, September 2020. doi:10.1186/s42400-020-00058-2.

[58] Yann Collet and contributors. Zstandard (zstd) real-time compression library. https://github.com/facebook/zstd, 2015.

[59] Ahmad Yasin. A Top-Down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 35–44, March 2014. doi:10.1109/ISPASS.2014.6844459.

[60] Hosein Yavarzadeh, Mohammadkazem Taram, Shravan Narayan, Deian Stefan, and Dean Tullsen. Half&Half: Demystifying Intel's Directional Branch Predictors for Fast, Secure Partitioned Execution.

[61] Adam Zabrocki. Re: [tech-memory-tagging] non-checked memory operations. URL: https://lists.riscv.org/g/sig-j/messages?expanded=1&msgnum=804.

[62] Chao Zhang, Tielei Wang, Tao Wei, Yu Chen, and Wei Zou. IntPatch: Automatically Fix Integer-Overflow-to-Buffer-Overflow Vulnerability at Compile-Time. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Dimitris Gritzalis, Bart Preneel, and Marianthi Theoharidou, editors, *Computer Security – ESORICS 2010*, volume 6345, pages 71–86. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. Series Title: Lecture Notes in Computer Science. URL: http://link.springer.com/10.1007/978-3-642-15497-3_5, doi:10.1007/978-3-642-15497-3_5.

Table 1: List of Custom Instruction Opcode in RISC-V

Mnemonic	funct7	funct3	Opcode
add.u	0000000	000	0101011
sub.u	0100000	000	0101011
mul.u	0000001	000	0101011
add.s	0000010	000	0101011
sub.s	0100010	000	0101011
mul.s	0000011	000	0101011
addi.u	-	000	0001011
addi.s	-	010	0001011
addw.u	0000100	000	0111011
subw.u	0100100	000	0111011
mulw.u	0000101	000	0111011
addw.s	0000010	000	0111011
subw.s	0100010	000	0111011
mulw.s	0000011	000	0111011
addiw.u	-	000	1011011
addiw.s	-	010	1011011

A Appendix

A.1 RISC-V Instructions

Table 1 lists all added instructions and their encoding for opX overflow checking in RISC-V.

A.2 LLVM Passes

Table 2 shows the list of modified passes in our custom LLVM build. Most passes are patched to skip instructions with overflow metadata attached to maintain semantic correctness with UBSan.

A.3 Additional Results

Figure 14 shows that 9.1% out of the total 13.9% UBSan overhead is caused by the interference with compiler optimizations. The cost is also significant for some benchmarks like *x264*, similar to x86 results.

Figure 15 shows the performance comparison of popular libraries on ARM. opX reduces geomean overhead from UBSan's 37% to 6.1% (`csel`) or 3.9% (`nop`). The LCI version also performs well in most benchmarks at 13% overhead, but still much higher than opX, highlighting the necessity of a hardware extension.

Figure 16 shows the performance comparison of popular databases on x86. opX reduces geomean overhead from UBSan's 6% to near zero.

Table 2: List of Modified Optimization Passes in LLVM

Passes	Modifications
Instruction Combine	• Propagate metadata when combining instructions into fewer and simpler instructions. • Disable transformations that alter overflow behaviors to avoid false positives and false negatives. For example, the transformation from `a-b` to `a+(0-b)` is disabled, as it introduces false positives when both operands are unsigned. • Eliminate unnecessary overflow checks on instructions that will never overflow, for example when operands are proven to be sufficiently small.
Reassociate	• Propagate metadata when reassociating commutative expressions. • Disable transformations that completely alter overflow behaviors to avoid false positives.
Aggressive Instruction Combine	This pass is not modified. As discussed in Section 7.4, although this pass may suppress some overflow violations, these overflows are intended by programmers (e.g., bit manipulation) and are benign.
Instruction Simplify	• Detect and report overflow violations during constant folding at compile time (Section 7.4).
CFG Simplify	• Avoid hoisting instructions that carry overflow metadata. Otherwise, the hoisted instructions will execute speculatively before safety checks, leading to potential false positives.
Loop Vectorizer	• Propagate metadata from original instructions to vectorized instructions.
SLP Vectorizer	• Propagate metadata from original instructions to vectorized instructions.
Induction Variable Simplify	• Disable some transformations on loop induction variables that alter overflow behaviors. For example, canonicalizing `i--` to `i+=(-1)` converts an unsigned decrement into a signed addition. This masks wrap-around behavior, introducing false negatives.
Inline Cost Analysis	• Increase the inline cost of instructions carrying metadata if they will be converted to software overflow checks later. This prevents aggressive inlining of instructions with software overflow checks, improving performance in some benchmarks like *opus*.
Correlated Value Propagation	• Eliminate unnecessary overflow checks on instructions that will never overflow.
Misc. Utilities	• Propagate metadata in widely-used utility functions.

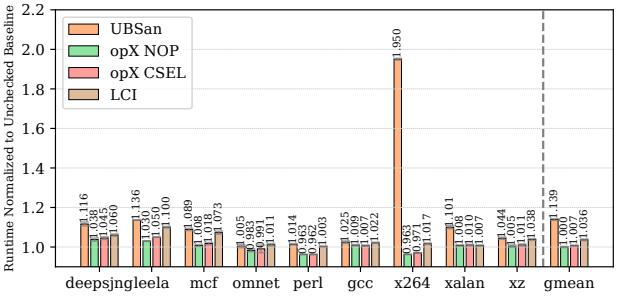

Figure 14: **The impact of UBSan intrinsics on compiler optimizations on ARM.** *Interference with compiler optimizations accounts for 9.1% out of the total 13.9% UBSan cost.*

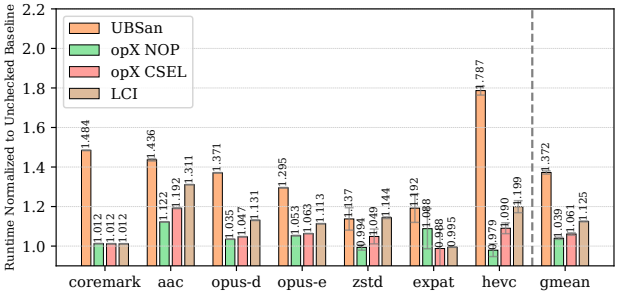

Figure 15: **Popular libraries and coremark on ARM.** *opX estimated with `csel` achieves 6.1% overall overhead compared to UBSan's 37%. The LCI version also achieves 13% overhead.*

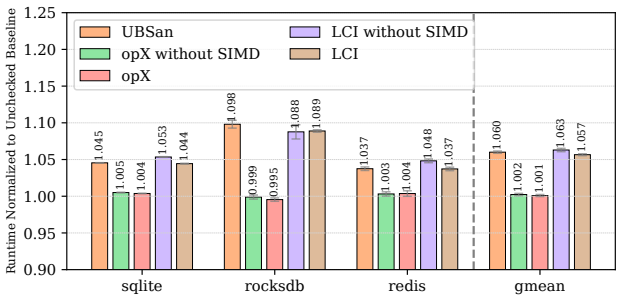

Figure 16: **Database on x86.** *opX achieves near zero overall overhead compared to 6% of UBSan. RocksDB is the most significant improvement.*