SafeNumbers: Design and Empirical Evaluation of Checked Fixed-Width Numeric Types for C++, from Disclosed CVEs to Heterogeneous Accelerators

Matt Borland
Purdue University, West Lafayette, IN, USA
The C++ Alliance
ORCID: 0009-0005-8183-4254
Email: borlanm@purdue.edu

Jenna DiVincenzo
Purdue University, West Lafayette, IN, USA
ORCID: 0000-0003-3029-2617
Email: jennad@purdue.edu

***Abstract*—Integer overflow, underflow, truncating conversion, and division by zero yield values that look valid but are wrong, and they remain common vulnerabilities in C and C++ software. Sanitizers typically run only in test builds, and trapping compiler flags can respond to a fault only by aborting. Overflow compiler intrinsics are non-portable, and must be written by hand at every call site. The checked-integer library in Boost follows C++ promotion rules by default, so a checked 8-bit addition can evaluate to 256, and no other existing C++ libraries run inside GPU kernels. We present SafeNumbers, a C++20 library of checked fixed-width integer, floating-point, and range-bounded types. It rejects implicit and mixed-width conversions at compile time, and every operation is `constexpr`, so a fault during constant evaluation fails the build. Its central design decision decouples the detection of a fault, implemented once per operation and proven sound in Why3, from the response to it. This decoupling lets the same types run in GPU kernels using CUDA or SYCL, where a device fault is recorded and response occurs on the host. We evaluate the library against 140 Common Vulnerabilities and Exposures (CVE) entries from the National Vulnerability Database (NVD), covering four integer weakness classes and a negative control. It prevents 126 of the 130 in-scope faults (96.9%, 95% Wilson interval 92.4% to 98.8%), including every entry in the four primary classes. Across seven host targets, the median per-operation overhead is 1.00x to 1.18x of native arithmetic, and on a vectorizing SYCL device a branch-free deferred response brings compute-bound floating-point chains from up to 235x performance loss of native to 1.1x to 2.9x while still reporting every error.**

***Index Terms*—integer overflow; software security; C++; safe arithmetic; empirical study; benchmarking; GPU computing; formal verification**

I. Introduction

Arithmetic errors, which include overflow past the largest value of a type, underflow below its smallest value, lossy conversions between types that truncate values, and division by zero, are pervasive even in mature, well-tested C++ code bases [2], [3] because such operations are valid and yield ordinary values rather than errors in the language. Since these values are functionally incorrect, they lead to miscalculations, corrupted program logic, or, at worst, critical software vulnerabilities. For example, a computation on a buffer size may cause the size to wrap around to a small number and result in a heap overflow or an out-of-bounds access when that value sizes an allocation or indexes an array. In practice, C++ software developers attempt to detect arithmetic errors in their code with sanitizers, compiler built-ins/intrinsics, or checked-integer libraries. But, each of these solutions have significant limitations when applied to modern C++ software.

Sanitizers and their trapping flags check only some builds and behaviors, or respond only by aborting. The primary sanitizer for C++, UndefinedBehaviorSanitizer (UBSan) [9] leaves unsigned-overflow checks as opt-in making it easy to miss detection of such issues, such as in the case of the OpenSSH challenge-response handler (see Section II for more details). Furthermore, UBSan works by instrumenting a build to report faults as they occur at run time and is typically only used during testing as a result. Therefore, any gap in test coverage can let arithmetic errors slip into production undetected. Even if a developer configures a sanitizer to trap in production, the only response it supports on detection is to abort—turning a vulnerability into a denial of service with no path to recovery.

Overflow intrinsics are precise but manual. The GCC and Clang overflow intrinsics (`__builtin_*_overflow`, e.g. `__builtin_add_overflow(a, b, &result)`) [10] are built-in functions that perform an arithmetic operation and return true when an overflow occurs (false otherwise). They compile to the processor's carry and overflow flags, so space and performance costs are low and result precision is high. Additionally, the returned boolean result allows developers to respond to overflows however they like in code. Unfortunately, all of these benefits come at the cost of developer effort and portability: the check and error handling must be manually written at every operation that could fault (hopefully none are missed!) and the intrinsics are compiler extensions rather than

Corresponding author: Matt Borland (borlanm@purdue.edu).

portable C++.

No checked-integer library reaches floating point types, 128-bit types, or device code. Checked-integer libraries, such as Boost.SafeNumerics [11], SafeInt [12], and Intel's safe-arithmetic library [13], move the check into a type, so that no operation on a checked value goes unchecked automatically. Unfortunately, no C++ library that we are aware of, offers checked floating-point types, 128-bit types, or device code.

C++ software now runs on GPUs. Modern C++ code increasingly runs on GPUs and other accelerators. Such device code, through say CUDA or SYCL, compiles into kernels that execute across thousands of parallel threads and is consequently constrained in ways host code is not. In particular, exceptions are unavailable because a checking scheme that throws them does not compile in a kernel. Additionally, performance depends on parallel threads executing the same instructions in compiler vectorizing loops. Any scheme that branches out of them early on arithmetic error can significantly degrade performance. Thus, mitigations that wish to cover whole C++ code bases must also work inside device code and cannot adopt the aforementioned traditional checking schemes.

Taken together, these gaps define what a practical mitigation must do. It must (i) protect production builds at every operation for integer, floating-point, and 128-bit types, not only in a tested subset or at the call sites a programmer remembered or for basic integer types; (ii) check each operation at the width the programmer declared; (iii) reject unsafe conversions at compile time; (iv) let the program choose how to respond to a fault; and (v) run in device code. No existing approach meets all five.

This paper presents SafeNumbers [30],[1] a header-only C++20 library of checked numeric types designed to meet all five requirements. It provides fixed-width integers (`u8` through `u128` and `i8` through `i128`), floating-point types (`f32` and `f64`), and range-bounded types, and each of its design decisions answers one requirement. The check belongs to the type rather than to a build configuration or a call site, so it is compiled into every build and applied to every operation on every input (i); adopting the library therefore changes declarations and the conversions at type boundaries, not every arithmetic expression. Every operation returns the type of its operands, so it is checked at the width the programmer declared, and the bounded types check every arithmetic result against their range, not only construction (ii). Implicit and mixed-width conversions are rejected at compile time, and because every operation is `constexpr`, a fault during constant evaluation also fails to compile (iii). SafeNumbers's central design decision separates the *detection* of a fault, implemented once per operation, from the *response* to it, which can throw, saturate, return an optional or a flagged value, terminate, or invoke a user-defined handler, chosen per call site or per type (iv). Because detection does not commit to a response, the same types run unchanged in CUDA and SYCL kernels, where a device fault is recorded and rethrown on the host, and a branch-free response lets a loop vectorize (v).

We evaluate SafeNumbers both formally and empirically. A checked type is only as trustworthy as its detection logic, so we verified the soundness of our detection algorithms in the Why3 deductive verification platform [18]. We also assess SafeNumbers's relevance, cost, and reach through answering the following research questions:

- **RQ1 (Relevance).** To what extent do checked fixed-width and domain-bounded types prevent the integer-arithmetic weaknesses behind real, disclosed Common Vulnerabilities and Exposures entries (CVEs)?
- **RQ2 (Cost).** What runtime overhead does always-on checking impose, relative to native arithmetic and to Boost.SafeNumerics?
- **RQ3 (Reach).** Which error-response mechanisms let the same checked types execute in exception-free, massively parallel device code, and at what cost?

To answer RQ1, we sourced 140 CVEs from the National Vulnerability Database (NVD) across four arithmetic weakness classes and a negative control, reduced each CVE to its root-cause computation, and wrote that computation once with built-in types and once with SafeNumbers. The library prevents 126 of the 130 in-scope faults, including every entry in the four primary classes. To answer RQ2, we benchmarked every operation at every width on seven operating-system and architecture targets against native arithmetic, Boost.SafeNumerics, SafeInt, and Intel's safe-arithmetic library. We found a median overhead of at most 1.16x of native arithmetic through 64 bits on 64-bit hosts. To answer RQ3, we timed memory-bound and compute-bound kernels under CUDA on an NVIDIA GPU and under SYCL on a CPU device, and found that most checks add no measurable cost to memory-bound GPU kernels, while on the SYCL device a branch-free response that records faults and reports them afterward keeps compute-bound floating-point chains within 3x of native arithmetic, down from up to 235x.

In summary our contributions, all open source under the Boost Software License [30], are:

1) SafeNumbers, a header-only C++20 library of checked 8- to 128-bit integer, floating-point, and bounded types whose detection of a fault is decoupled from the response to it, so one set of types serves throwing host code, exception-free code, vectorized loops, and CUDA and SYCL kernels, with the detection algorithms proven sound in 195 Why3 lemmas (Section III).
2) A reproducible corpus of 140 NVD-sourced CVEs across four integer weakness classes and a negative control, each reduced to a self-contained test with machine-readable metadata, and a script that regenerates every corpus statistic (Section IV-A).
3) A per-operation cost model for checked arithmetic when no fault occurs, across seven host targets and two heterogeneous back ends, with a semantic probe matrix and

[1]SafeNumbers was developed with support from The C++ Alliance and is intended for proposal to Boost. It is not an official Boost library at the time of writing, although its namespace, `boost::safe_numbers`, anticipates that step.

a cost comparison against three other checked-integer libraries (Sections V-B and V-C).
4) An ablation over the library's own response policies showing that in vectorizable loops the cost is set by the kind of response, one that yields a value or one that exits the loop early, rather than by the detection code, which makes the decoupling the enabling mechanism for vectorized loops and device kernels (Sections V-B and VI).
5) Lessons for practitioners on where checked arithmetic could be used and how to choose a response (Section VI).

The remainder of this paper is organized as follows. Section II describes the integer weakness classes and the C++ rules that make them easy to write. Section III presents the design of SafeNumbers. Section IV describes the study design for each research question, and Section V reports the results. Section VI discusses lessons for practitioners, Section VII the threats to validity, and Section VIII related work. Section IX concludes and outlines future work.

II. Background and Motivation

A. Integer Weaknesses as Root Causes

We consider four weakness classes from the NVD whose root cause is a fixed-width integer computation. *CWE-190, Integer Overflow or Wraparound*, leads to the classic allocation-size fault. In CVE-2002-0639 the OpenSSH challenge-response handler computed `nresp * sizeof(char*)` in 32 bits, an attacker-controlled `nresp` of 2^{30} wrapped the product to zero, and the subsequent writes overflowed the heap. *CWE-191, Integer Underflow*, goes the other way, typically a length minus a header size that wraps from zero to a huge value. *CWE-681, Incorrect Conversion Between Numeric Types*, covers signedness confusion and truncation: a negative signed length compared against an unsigned bound passes the check because the comparison converts it to a large positive value. *CWE-369, Divide by Zero*, occurs where an untrusted divisor reaches a division or modulo. In each, a representable but wrong value propagates to a memory operation, and the CVE is filed against that operation; the arithmetic upstream is the root cause. The negative control class used is *CWE-682, Incorrect Calculation*.

B. Why the Language Does Not Help

Consider the comparison `-1 < 1U`. Intuitively the result is `true`. In C++ the signed operand converts to `unsigned int`, so the expression is `false`. The conversion is well defined, and it is diagnosed only when `-Wsign-compare` is enabled. The hardware already computes carry and overflow flags on every arithmetic instruction, what is missing is a type that consults them.

III. SafeNumbers

This section presents the design of SafeNumbers: each decision, the gap in the existing mitigations it closes (Section I), and what it costs. The full reference is available online [30].

A. Explicit Over Implicit

The library prohibits implicit conversion in every direction: between safe and built-in types, between safe types of different widths, and between signed and unsigned safe types. These are all compile-time errors, as are construction from `bool` and unary minus on unsigned types. This closes the promotion gap of Boost.SafeNumerics. An operation on two `u8` values is checked as an 8-bit operation and yields a `u8`. A signedness confusion (CWE-681) no longer compiles, so it is found even if there is no test input that triggers this issue. The cost is verbosity at type boundaries, where a `static_cast` is required, but every conversion becomes visible in the source. These type requirements also enables simpler verification. Every operation stays within a single ring $\mathbb{Z}/2^n\mathbb{Z}$, which removes widening and narrowing from the proof obligations.

B. Fail as Early as Possible

Every operation is `constexpr`. The default error path contains a `throw` expression, which is not permitted in a constant expression, so an overflow during constant evaluation becomes a compile-time error rather than a run-time exception. The developer writes one expression and receives the strongest guarantee the context allows: a build failure where the operands are known, or a check where they are not.

C. Detection Decoupled from Response

Handling an arithmetic fault has two parts, detecting it and responding to it. The mitigations in Section I either fix the response or leave it to each call site, so none fits both a server that must recover and a kernel that cannot throw. SafeNumbers implements detection once per operation, classifying the outcome into an `error_kind` (`overflow`, `underflow`, `divide_by_zero`, and for floating point `nan_operation` and `invalid_operation`), and routes the classification to a response chosen independently. The detection code, and the proofs that cover it, are therefore shared by every response. Three API surfaces expose that choice to the developer.

Default operators throw a typed standard exception (`std::overflow_error`, `std::underflow_error`, `std::domain_error`). The throw itself is contained in a function annotated as cold, and forced non-inline so that an operator's body is only the arithmetic, the check, and a branch.

Call-site families, named after Rust's primitive-type methods [16], select the response per operation without changing the type: `checked_add` returns `std::optional`, `saturating_add` clamps, `overflowing_add` returns a `std::pair` of the (possibly wrapped) value and a flag, `strict_add` immediately terminates with `std::exit`, and `widening_add` promotes to the next width (e.g. `u32 + u32` yields `u64`).

Type-level policies carry the response in the type for every operator: `sat_u8` saturates, `strict_i32` terminates, and `basic_u32<Handler>` invokes a user-defined stateless handler whose `on_error(error_kind, value, message)` return value becomes the operation's result. Only

Listing 1. A SYCL kernel using the deferred response. Detection is unchanged; the response is one branch per work item.

```
boost::safe_numbers::device_error_context ctx {q};

q.parallel_for(sycl::range<1>(n), [=](sycl::id<1> idx)
{
    const auto i {idx[0]};
    boost::safe_numbers::deferred_errors errs;

    // unwrap() folds each flag in without branching
    auto x {errs.unwrap(overflowing_add(in[i], in[i]))};
    x = errs.unwrap(overflowing_mul(x, in[i]));

    errs.report(__FILE__, __LINE__); // one branch
    out[i] = x;
});

// rethrows on the host if any item reported
ctx.synchronize();
```

responses whose result is the operand type may live in the type, so a value of a safe type is always exactly its numeric value. The `noexcept` specification of every operator follows the handler, so a non-throwing handler yields non-throwing arithmetic.

D. Bounded Domains as an Additional Precondition

The bounded types express a value's legal domain in the type (`bounded_uint<Min, Max>`, `bounded_int`, and `bounded_float`) and enforce it on construction and on every arithmetic result, unlike the range types of Boost.SafeNumerics, which check only construction. This is modeled on Ada's range subtypes [17], and it reaches faults that would be legal within a type width. A correct addition whose result is one past the end of a 150-element table does not overflow, but `bounded_uint<0, 149>` rejects it when the value is formed, before a bad-memory access could occur. Whenever a bounded value is converted out of its type, the declared domain is still provided to the optimizer as an assumption, so checks on the receiving type that the domain makes impossible are deleted (Section V-B).

E. Device Execution

All integer, floating-point, and bounded types compile with NVCC and with Intel's oneAPI DPC++ compiler for execution inside CUDA and SYCL kernels, so host and device code share one set of checked types. Device code cannot throw, so only the response half of handling a fault changes. A `device_error_context` contains a shared error record (CUDA `__managed__` memory or a SYCL `device_global`) into which the first faulting work item writes the file, line, thread identifier, expression, and exception type. A subsequent call to `synchronize()` on the host waits for the kernel, and throws the same standard exception as if the operation had been executed on the host. For hot kernels a second response exists, the `overflowing_*` families produce a value and a flag with no branch, and a `deferred_errors` accumulator collects the flags across a chain and raises them once per work item (Listing 1), trading the precision of the report for a straight-line kernel body.

F. Soundness of the Detection Logic

Following the verification of GMP's low-level routines in Why3 [19], [20], the overflow, underflow, and edge-case detection for every fundamental operation was re-expressed in WhyML and proven with Why3 [18]. The unsigned theory clones Why3's generic bit-vector module so one proof covers every width from 8 to 128 bits. The signed theory proves the two's-complement sign-bit identities, and the asymmetric cases `INT_MIN / -1` and `-INT_MIN`. Lastly, the floating-point theory proves that the IEEE 754 result classifier assigns the correct category to every input. In total, we verify 195 lemmas. The C++ source is a manual translation of the verified algorithms, so the claim is soundness of the detection logic, not verification of the shipped source.

IV. Study Design

A. RQ1: CVE Corpus

1) Sourcing: CVEs were fetched from the NVD via their REST API 2.0 [25], by CWE identifier, on 19 August 2026. Four primary categories (CWE-190, CWE-191, CWE-681, CWE-369) were sampled with thirty relevant examples each. CWE-682, Incorrect Calculation, serves as a negative control with twenty examined entries. Many of its defects are in-range logic errors that no arithmetic check can catch, so we expect a lower rate.

2) Reproduction: For each CVE we extracted the computation that the disclosure and its patch identify as the root cause and wrote a self-contained test set with three sections: a native part that computes the expression with built-in types and asserts the specific silent wrong value the disclosure describes, a safe part that computes the identical expression with the library types and asserts the fault (a thrown exception of the expected type, or a compile failure), and a bounded part that declares the value's documented domain with a bounded type and asserts rejection at construction. Where the original width was platform dependent we model the width the advisory publishes; 96 of the 130 in-scope entries model 32 bits, and the 30 CWE-681 entries mix signed and unsigned operands.

3) Outcome categories: Each examined CVE receives one outcome: prevented at run time (the safe computation throws at the faulting operation), prevented at compile time (the bug is ill-formed using the library and is committed as a compile-fail test), prevented by a bound (the declared domain rejects the input before the arithmetic), not prevented (a genuine arithmetic CVE whose faulting value is in range, counted in the denominator), or out of scope (not a fixed-width integer fault on reduction, recorded with a reason and reported separately). Each test carries a machine-readable metadata block, and a dependency-free script cross-checks the blocks against the test outcomes and regenerates every corpus table, exiting non-zero on any inconsistency, so the corpus is validated in continuous integration.

4) Statistics: The primary metric per category is the prevention rate, prevented over in scope, with a 95% Wilson score interval [22], reported once with the default checked

types alone and once including the bounded types. The native baseline detects zero faults as expected, so every prevented case is a discordant pair and the exact McNemar test [23] reduces to $p = \min(1, 2 \cdot 0.5^b)$ for b prevented. We reproduce the root-cause computation, not the original program, so the claim is that the fault would have surfaced at the faulting operation, not that the whole program would have run correctly (Section VII).

B. RQ2: Host Benchmark Grid

The benchmarks use Google Benchmark [24]. Each benchmark fills a vector of one million random values once, then for every operation evaluates `op(vec[i], vec[i+1])` across the vector, passing each result through `DoNotOptimize` to ensure the computation is not elided. Code is compiled with `-O2`, or MSVC equivalent. Integer operands are drawn from a uniform distribution over $[1, w-1]$ for a w-bit type and sorted in descending order so that all five operations stay on the non-overflowing path. Floating-point operands are drawn uniformly from $[1, 100]$. The measurement therefore reflects the cost of the checks that production code pays on every operation, not the cost of throwing, which is paid only on a fault. We run five repetitions and report the median time per element.

Three contenders are compared at each width: the built-in type as baseline, the library type, and Boost.SafeNumerics's `safe<T>` with its default template arguments. Boost.SafeNumerics has no floating-point and no 128-bit type, so those tables only have two contenders. The grid ran on seven GitHub Actions runners: Linux x64 and ARM64 (GCC 14), Linux x86 32-bit (GCC 14, `-m32`), macOS ARM64 (Apple Clang), and Windows x64, ARM64, and x86 32-bit (MSVC), all at C++20.

1) Semantics of the comparators: RQ2 is a question of cost-effectiveness rather than cost alone, because Boost.SafeNumerics checks promoted rather than declared widths by default, so we first establish what each comparator detects with ten semantic probes, each a single arithmetic situation with runtime operands (Table III), expressed idiomatically in SafeNumbers, Boost.SafeNumerics, Microsoft SafeInt [12], and Intel safe-arithmetic [13], all in their default configuration. A driver compiles every (library, probe) pair, records a compile failure as a compile-time rejection with its diagnostic, and otherwise runs the binary and records its output line. Intel safe-arithmetic tracks intervals in the type and proves containment at compile time, with a runtime check only where a raw value enters an interval type through `safe::match`, so its probes admit the operands through `match`.

2) Cross-library cost: A second benchmark runs the built-in type, SafeNumbers, Boost.SafeNumerics, SafeInt, and Intel safe-arithmetic on the same operands at 32 and 64 bits. The Intel library admits both operands through `match` and performs the proven operation unchecked (unsigned subtraction on equal intervals is not expressible and is therefor omitted).

3) Ablation over response policies: A third benchmark isolates the response half of the architecture, testing the claim that the choice of response, not the shared detection code, sets the cost. For 8-, 32-, 64-, and 128-bit unsigned and 32- and 128-bit signed widths it runs the five operations on the built-in type and on seven responses: the default throwing operators, the saturating and strict aliases, two user-defined handlers (a `noexcept` handler returning the wrapped fallback, and one throwing a user exception type), the deferred `overflowing_*` family with its flag folded without a branch, and the `checked_*` family unwrapped with a fallback. A further pair of rows measures what a declared domain provides. The default operators on operands validated into a bounded type at ingress, against a built-in loop over the same one-byte operands.

4) Two kernel shapes: The host grid times a *scalar* shape in which every element's result passes through `DoNotOptimize`, which forces a round trip per element and prevents vectorization for every contender alike. The cross-library and ablation benchmarks additionally time a *stream* shape, `out[i] = a[i] op b[i]` into an output array with one memory clobber after the pass, the shape real element-wise loops have, which lets the optimizer vectorize whichever contenders it can. Both ran locally on an Apple M4 Max (macOS 26.6, Apple Clang 21.0, `-O2`, five repetitions, median reported), with inputs on the non-violating path as in the grid.

C. RQ3: Device Benchmarks

Two dedicated benchmarks, one per back end, time identical kernels for each safe type and its basis type, so the ratio isolates the cost of the checks. Inputs are chosen so no check fires, and a `device_error_context` covering the run aborts the benchmark if one does. Two kernel shapes are measured: *streaming*, `out[i] = a[i] op b[i]` over 4 194 304 elements, the shape of real element-wise kernels, memory bound on a discrete GPU; and a *compute-bound chain* in which 65 536 threads each evolve one register-resident value through 256 cycles of eight to nine checked operations. The integer chain mixes in division and modulo so that the optimizer cannot fold the native baseline's wrapping additions; floating-point chains are per operation, since IEEE arithmetic is not associative. Every family also runs the non-throwing forms against the same baseline: a saturating addition streaming row, saturating and overflowing integer chains, and deferred floating-point chains whose flags are accumulated and folded into the result. Every timed region is verified afterwards: safe and native results must match bit for bit.

The CUDA benchmark ran on an NVIDIA GeForce RTX 3060 (sm_86, 28 SMs) with CUDA 13.1 on Ubuntu 24.04 at `-O2`. The SYCL benchmark ran on the OpenCL CPU device of an Intel Core i9-11900K with oneAPI icpx 2026.0 at `-O3`. Because icpx defaults to `-ffp-model=fast`, which grants the optimizer floating-point associativity and would let the native chains fold, the build pins `-ffp-model=precise` so both contenders run IEEE arithmetic exactly as under NVCC.

TABLE I
DISPOSITION AND PREVENTION RATE BY CATEGORY. NOTHING EXAMINED IS DROPPED; OUT-OF-SCOPE ENTRIES ARE REPORTED AND REMOVED FROM THE DENOMINATOR. "DEFAULT" COUNTS THE CHECKED TYPES ALONE; "BND." ADDS CASES PREVENTED ONCE THE DOMAIN IS DECLARED WITH A BOUNDED TYPE. THE RATE, IN PERCENT, IS ON THE LATTER WITH ITS 95% WILSON SCORE INTERVAL.

Category	Exam.	Out	In	Default	Bnd.	Rate (95% CI)
CWE-190	30	0	30	29	30	100.0 (88.6 to 100.0)
CWE-191	30	0	30	30	30	100.0 (88.6 to 100.0)
CWE-681	30	0	30	30	30	100.0 (88.6 to 100.0)
CWE-369	30	0	30	30	30	100.0 (88.6 to 100.0)
CWE-682 (ctrl.)	20	10	10	3	6	60.0 (31.3 to 83.2)
All	140	10	130	122	126	96.9 (92.4 to 98.8)

V. RESULTS

A. RQ1: Prevention of Disclosed CVEs

Table I gives the disposition and prevention rate of every category. Of 140 examined CVEs, 10 were out of scope, all in the CWE-682 control, for reasons recorded in the corpus. The 130 in-scope entries span 2002 to 2022 across products from the Linux kernel to LibTIFF, QEMU, and FreeType. The exact McNemar p is 1.9×10^{-9} for each primary category, 0.031 for the control, and 2.4×10^{-38} pooled.

1) Primary categories: All four primary categories are prevented at 100% including the bounded arm, with a Wilson lower bound of 88.6% at $n = 30$. The mechanism that fires differs by category. CWE-190, CWE-191, and CWE-369 are prevented at runtime by the throwing operators. In CWE-681, 17 of 30 entries are prevented at *compile time*: the root cause is a comparison or arithmetic between a signed and an unsigned quantity, and porting the expression to `i32` and `u32` makes it ill-formed; the remaining 13 are truncating conversions that throw on the explicit narrowing cast. One CWE-190 entry is prevented only through the bounded arm: in CVE-2018-13785 (libpng) an oversized but representable image width overflows a row-factor calculation, and declaring libpng's documented width limit as a `bounded_uint` rejects it at the boundary, before the downstream faults the runtime checks would also have caught.

2) The negative control: CWE-682 behaves as a control should. The default types prevent three of ten in-scope cases (divide-by-zero and shift faults the NVD tagged as calculation errors), and declared domains prevent three more (CVE-2011-3062, CVE-2011-1573, CVE-2018-11790), each a wrong index, length, or offset used against a buffer of independently known size. The remaining four stay unprevented because their wrong result is a valid in-domain value: three are integer divisions that truncate where the formula needed rounding or a ceiling (CVE-2016-7433, CVE-2017-8932, CVE-2018-14439), and one is an index one past the end with no independently known bound to declare (CVE-2018-20999). No bound applies, which is exactly the library's limit and why the control is informative.

3) Answer to RQ1: Among disclosed CVEs whose root cause reduces to a fixed-width integer arithmetic fault, the checked types convert the fault into a typed, catchable failure in every sampled case of the four primary classes, bounded domains extend prevention to in-range faults the operation-level checks cannot see, and the pooled rate is 96.9% (92.4% to 98.8%), bounded below by the honest failures in the control. The explicit-conversion rule catches 17 of the conversion faults with no test input at all.

B. RQ2: Host Overhead

Table II summarizes the grid as the geometric mean, median, and maximum of the per-operation ratio across each type class; the full 58-row tables per platform are in the documentation.

1) Through 64 bits: On every 64-bit host the median per-operation overhead of the checked integer types lies between 1.00x and 1.16x, and the geometric mean between 1.03x and 1.14x. Division and modulo are at parity almost everywhere (on Linux x64 all 16 division and modulo rows through 64 bits are at 1.00x), because the only check is a compare against a divisor already in flight. Addition and subtraction cost about 1.2x on GCC x64 and are at parity on MSVC. Multiplication is the most expensive check, up to 1.7x with GCC and 2.36x for signed 64-bit multiplication on Windows ARM64, because the overflow test needs a wide product or a division; that one cell is the only miss of the design target of less than 2x on 64-bit hosts.

2) 32-bit targets: The two 32-bit targets are where the cost is visible: on Linux x86 32-bit six of the 40 integer rows exceed 2x, all at 8 and 16 bits (3.0x to 4.3x), where GCC's code for the narrow overflow intrinsics is poor; on Windows x86 32-bit, 64-bit multiplication reaches 5.69x because product and check are both emulated in 32-bit registers, and the floating-point classifier costs 1.7x to 4.7x.

3) 128-bit and floating-point: The 128-bit types cost 1.01x to 1.48x by geometric mean on GCC and Clang hosts; their multiplication check, computed from 64-bit limbs because there is no wider arithmetic to lift into, is the one genuinely expensive check in the library, peaking at 3.05x to 7.73x on every host except Linux x64 and ARM64, where GCC keeps it at parity. The floating-point classifiers cost 1.18x on GCC x64, are at parity on Apple Clang and MSVC x64, and reach 1.5x under GCC on Linux ARM64 and MSVC on Windows ARM64.

4) Against the comparators: semantics: Table III gives the outcome of the ten semantic probes; the four libraries embody three models. SafeNumbers and SafeInt check at the declared width at runtime and agree on every runtime fault (P1 to P5 and P7); at the type boundary SafeNumbers rejects mixed signedness, mixed width, and `bool` construction at compile time, while SafeInt compares mixed signedness correctly (P6), accepts `bool`, and rejects mixed widths only through an overload ambiguity. Boost.SafeNumerics under its default promotion widens the 8-bit overflow to 256 (P1), lets a bounded result escape its range (P10), and throws on a mixed-signedness comparison instead of comparing. Intel safe-arithmetic rejects at compile time everything its interval

TABLE II

HOST OVERHEAD RELATIVE TO THE BUILT-IN TYPE OF THE SAME WIDTH, NON-VIOLATING PATH, MEDIAN OF FIVE REPETITIONS. EACH CELL IS GEOMETRIC MEAN / MEDIAN / MAXIMUM OF THE PER-OPERATION RATIO ACROSS THE CLASS (40 OPERATIONS FOR INT 8 TO 64, 10 FOR 128, 8 FOR FLOAT). BOOST.SAFENUMERICS HAS NEITHER 128-BIT NOR FLOATING-POINT TYPES.

Platform	Compiler	SafeNumbers int 8 to 64	SafeNumerics int 8 to 64	SafeNumbers 128	SafeNumbers float
Linux x64	GCC 14	1.14 / 1.16 / 1.66	1.25 / 1.25 / 4.72	1.10 / 1.10 / 1.81	1.18 / 1.20 / 1.29
Linux ARM64	GCC 14	1.12 / 1.02 / 1.71	1.18 / 1.06 / 10.60	1.01 / 1.03 / 1.04	1.47 / 1.46 / 1.88
Linux x86 32-bit	GCC 14	1.34 / 1.18 / 4.32	1.29 / 1.10 / 4.56	1.48 / 1.24 / 3.46	1.06 / 1.05 / 1.34
macOS ARM64	Apple Clang	1.07 / 1.08 / 1.37	0.95 / 0.96 / 1.04	1.16 / 1.07 / 3.05	1.01 / 1.01 / 1.04
Windows x64	MSVC	1.08 / 1.03 / 1.50	1.04 / 1.01 / 1.50	1.32 / 1.22 / 4.99	1.02 / 1.02 / 1.09
Windows ARM64	MSVC	1.03 / 1.00 / 2.36	1.07 / 1.00 / 2.99	1.61 / 1.19 / 7.73	1.51 / 1.44 / 1.86
Windows x86 32-bit	MSVC	1.26 / 1.16 / 5.69	1.08 / 1.03 / 2.00	1.97 / 2.15 / 4.09	2.66 / 3.83 / 4.73

logic can see and is the one comparator whose bounded type enforces on results (P10), but it accepts `bool`, and its division interval ignores the sign of the quotient, so `INT_MIN / -1` compiles and returns `INT_MIN` (P4). Only SafeNumbers converts every one of the ten situations into a compile-time or runtime fault.

5) Against the comparators: cost: Table IV places the four libraries on the same operands and stream shape; cost follows the response class rather than the library. The three libraries that detect at the declared width and throw, SafeNumbers, Boost.SafeNumerics, and SafeInt, form one class: on unsigned 32-bit addition, subtraction, and multiplication they cost 2.6x to 4.4x, on unsigned 64-bit 1.3x to 1.6x (with one Boost.SafeNumerics addition outlier at 2.5x), and all three are at parity on unsigned division and modulo. Intel safe-arithmetic is the cheapest contender at 0.95x to 1.34x on 32 bits, but its runtime work is not smaller, only differently placed: two range checks on the inputs with a fallback value, rather than one check on the result with an exit. Two controls isolate the effect. Giving `match` a throwing fallback moves Intel to 3.6x to 5.1x, the class of the throwing libraries, and removing its runtime check gives 0.95x to 1.3x, so the checks themselves cost at most 0.3x, about what our value-returning `checked` family costs on addition and subtraction (Table V). Interval reasoning is genuinely cheaper only for multiplication, where a bound check on two inputs vectorizes and a check on the high half of the product does not (1.34x against 4.0x to 5.2x for our unsigned saturating, overflowing, and checked forms). The contracts also differ: Intel enforces a declared input domain and cannot express a full-width addition whose overflow is detected, so the two coincide only when the domain is known, which is what this benchmark constructs.

Signed 32-bit arithmetic is where the comparison improved the library. In our first run SafeNumbers cost 10.1x to 13.3x on signed addition, subtraction, and multiplication, against 4.5x for SafeInt, because Clang declined to inline every signed throwing operator (cost 320 against a threshold of 225) once the exception, its message, and its source location were constructed inline. Outlining that error path into the cold, non-inlined function of Section III-C restored inlining for all twenty signed operators; Table IV reports the library after the fix, 4.0x to 6.1x, the class of SafeInt (3.4x to 4.5x), while Boost.SafeNumerics remains at 10.0x to 13.1x.

6) Ablation over response policies: Table V is the ablation on the stream shape, where the built-in 32-bit loops vectorize four lanes wide and the 64-bit loops two. The responses split into two classes, and the compiler's own diagnostics say why. Every response that must act at the exact faulting element (throw, a throwing handler, or `strict`'s termination) is an early exit and draws the remark "cannot vectorize early exit loop with writes to memory"; these cost 2.8x to 5.1x for 32-bit addition, subtraction, and multiplication, falling to 1.3x to 1.4x at 64 bits where only two lanes are lost. Every response whose result is a value keeps the loop straight-line and vectorizes: the wrapping handler (0.74x to 1.32x), saturating addition and subtraction, which lower to the native saturating vector instructions (0.74x to 1.00x), and the deferred `overflowing` family and `checked` unwrapped with a fallback (0.96x to 1.29x for addition and subtraction). Multiplication is the exception: the unsigned overflow test needs the high half of the product, which has no 32-bit vector form on this target, so unsigned saturating, overflowing, and checked multiplication, and signed saturating multiplication, cost 4.0x to 5.2x, while the signed deferred and checked forms stay at 1.4x to 1.6x. Division and modulo are at parity for every response because the built-in loop does not vectorize them either. At 128 bits, where the basis type is a two-word class, the default operators and value responses stay within 1.25x (unsigned) and 1.6x (signed) except for multiplication, where the throwing and saturating responses cost 1.6x to 1.8x unsigned and 3.2x to 3.3x signed, and the deferred and checked forms 7x to 9x.

The bounded ingress rows show what a declared domain buys. Because the bounded types hand their domain to the optimizer as an assumption whenever a value leaves the type, operands validated into `bounded_uint<1,31>` and widened to `u32` carry the fact that they lie in $[1, 31]$, and Clang deletes every check that fact makes impossible before vectorizing: the default throwing operators cost 1.02x to 1.36x on 32-bit addition and multiplication and 0.81x to 1.07x at 128 bits. Unsigned subtraction keeps its check (5.8x) because $[1, 31] - [1, 31]$ can underflow; the optimizer removes exactly what the declared facts justify.

The result corrects an expectation we held going in: `strict`, modeled on Rust's panic on overflow, does not vectorize, because termination at the faulting element is an early exit exactly as a throw is (Rust's release-mode arithmetic

TABLE III

SEMANTIC PROBES WITH RUNTIME OPERANDS, DEFAULT CONFIGURATION OF EACH LIBRARY. CT: REJECTED AT COMPILE TIME. RT: RUNTIME FAULT (EXCEPTION). OTHER CELLS GIVE THE VALUE THE PROGRAM PRINTED. INTEL S.-A. IS INTEL SAFE-ARITHMETIC.

Probe	SafeNumbers	Boost.SafeNumerics	SafeInt	Intel s.-a.
P1 `u8` $255 + 1$	RT	widened (256)	RT	CT
P2 `u32` $0 - 1$	RT	RT	RT	CT
P3 `u32` $2^{31} \cdot 2$	RT	RT	RT	CT
P4 `i32` `INT_MIN` / -1	RT	RT	RT	wrong (`INT_MIN`)
P5 `u32` $x/0$	RT	RT	RT	CT
P6 `i32`$(-1) <$ `u32`(1)	CT	RT	correct (true)	CT
P7 `u32` 300 to `u8`	RT	RT	RT	CT
P8 `u8` + `u32`	CT	widened (2)	CT (ambiguous)	widened (2)
P9 construct from `bool`	CT	accepted (1)	accepted (1)	accepted (1)
P10 $[0, 100]$: $60 + 50$	RT	escaped (110)	no bounded type	CT

TABLE IV

CROSS-LIBRARY COST AT 32 BITS, RATIO TO THE BUILT-IN TYPE, STREAM SHAPE, APPLE M4 MAX WITH APPLE CLANG 21 AT `-O2`, MEDIAN OF FIVE REPETITIONS. INTEL SAFE-ARITHMETIC PERFORMS TWO BOUNDARY RANGE CHECKS AND AN UNCHECKED PROVEN OPERATION.

Type	Library	Add	Sub	Mul	Div	Mod	Gmean
`u32`	SafeNumbers	2.60	3.63	4.20	0.98	1.01	2.09
`u32`	Boost.SafeNumerics	2.91	3.67	4.36	0.97	1.02	2.15
`u32`	Microsoft SafeInt	2.68	3.68	4.07	1.04	1.01	2.11
`u32`	Intel safe-arithmetic	0.97	n/a	1.34	1.05	1.05	1.09
`i32`	SafeNumbers	3.99	5.13	6.05	0.98	1.02	2.62
`i32`	Boost.SafeNumerics	9.99	13.13	10.61	1.81	0.99	4.78
`i32`	Microsoft SafeInt	3.40	3.97	4.49	0.99	0.98	2.26
`i32`	Intel safe-arithmetic	0.95	1.28	1.25	1.04	1.02	1.10

vectorizes because it wraps, not because it panics). What vectorizes is detection whose response is a value, the property that makes the deferred family the enabling mechanism on devices (Section V-C). Replacing a response with another of the same class leaves the cost where the class puts it: the user throwing handler matches the default operators for unsigned 32-bit (3.70x against 3.67x for addition), and the wrapping handler matches the built-in type. The signed user throwing handler (9.2x to 11.5x) is the inlining failure above again: its inline `throw` pushes the operator over Clang's threshold (235 against 225), so a user handler that throws should, like the library, call a cold, non-inlined function. In the scalar shape, where nothing vectorizes, all seven responses fall within 1.0x to 1.2x of one another apart from cells affected by a harness artifact (Section VII).

7) Answer to RQ2: On 64-bit hosts, always-on checking at the declared width costs a median of 1.00x to 1.16x per operation through 64 bits, with division and modulo free and multiplication the most expensive; the exceptions (32-bit targets, 128-bit multiplication, and floating-point classification on some ARM64 compilers) are quantified above. Relative to Boost.SafeNumerics the cost is comparable where the semantics are comparable, and SafeNumbers detects faults that library's default configuration silently absorbs. On a vectorizable loop the class of the response sets the cost regardless of library: a response that is a value vectorizes at 0.7x to 1.3x of native, and an early exit costs 3x to 6x at 32 bits.

TABLE V

ABLATION OVER RESPONSE POLICIES, RATIO TO THE BUILT-IN TYPE, STREAM SHAPE, SAME MACHINE AS TABLE IV. EVERY ROW RUNS THE SAME DETECTION CODE; ONLY THE RESPONSE DIFFERS. †OPERANDS VALIDATED INTO `BOUNDED_UINT<1,31>` OR `BOUNDED_INT<1,31>` AT INGRESS AND WIDENED PER ELEMENT; RATIO TO A BUILT-IN LOOP OVER THE SAME ONE-BYTE OPERANDS.

Type	Response	Add	Sub	Mul	Div	Mod	Gmean
`u32`	throwing (default)	3.67	3.51	3.87	0.99	0.98	2.17
`u32`	saturating	0.98	1.00	3.95	0.97	0.97	1.30
`u32`	strict	3.54	3.31	3.74	0.98	0.99	2.12
`u32`	user handler, wrapping	1.32	1.17	1.12	0.98	1.01	1.11
`u32`	user handler, throwing	3.70	3.31	3.71	0.97	0.97	2.12
`u32`	overflowing (deferred)	1.16	0.96	4.36	0.97	0.98	1.36
`u32`	checked (optional)	1.29	1.21	5.16	0.98	1.00	1.51
`u32`	throwing, bounded ingress†	1.36	5.81	1.09	1.00	1.00	1.54
`i32`	throwing (default)	3.85	3.92	4.69	1.03	1.05	2.38
`i32`	saturating	1.00	0.74	4.19	1.02	1.08	1.28
`i32`	strict	5.13	2.80	3.45	1.02	1.07	2.22
`i32`	user handler, wrapping	0.87	0.74	0.74	1.06	1.04	0.88
`i32`	user handler, throwing	11.53	10.32	9.20	1.02	1.04	4.10
`i32`	overflowing (deferred)	1.00	0.96	1.57	1.05	1.08	1.11
`i32`	checked (optional)	0.99	1.01	1.43	1.03	1.03	1.09
`i32`	throwing, bounded ingress†	1.24	0.90	1.02	0.97	0.98	1.02

C. RQ3: Device Execution

1) Streaming: Table VI gives selected streaming rows for both devices. On the GPU the checks are essentially free: the kernels are bound by memory bandwidth and the checks execute in the shadow of the loads; the rows above parity are the 8-bit types (up to 1.23x), `f64` division (1.44x), and 128-bit multiplication (6.8x). The SYCL CPU device tells the same story for 32- to 128-bit integers (within 11%), but not for `f32` (1.3x to 1.7x) or below 32 bits (up to 1.9x at 16 bits): four million one-byte elements fit in cache, the kernels are compute bound, a CPU device executes a kernel by vectorizing it across work items, and the per-operation reporting branch inhibits exactly that, so the 8-bit rows reach 11.6x. `saturating_add` compiles to the native saturating vector instructions, returning `u8` to parity and `i8` to 1.5x.

2) Compute-bound integer chains: With the arithmetic pinned in registers the worst case becomes visible. On the GPU the throwing integer types cost 1.30x to 2.93x through 64 bits and 7.78x to 10.30x at 128 bits, and the non-throwing forms

TABLE VI
STREAMING KERNELS, RATIO OF SAFE TIME TO BASIS-TYPE TIME, SELECTED ROWS. RTX 3060 UNDER CUDA AND INTEL I9-11900K OPENCL CPU DEVICE UNDER SYCL.

	CUDA (RTX 3060)			SYCL (CPU device)		
Type	Add	Mul	Sat. add	Add	Mul	Sat. add
`u8`	1.05	1.05	1.02	5.47	11.60	0.97
`u32`	1.00	1.00	1.00	1.03	1.04	1.00
`u64`	1.00	1.00	1.00	1.03	1.04	1.00
`u128`	1.00	6.77	1.00	1.00	1.05	0.99
`i8`	1.23	1.22	1.04	9.69	11.32	1.52
`i32`	1.00	1.00	1.00	1.00	1.00	0.96
`i64`	1.00	1.00	1.00	1.04	1.03	1.00
`f32`	1.00	1.00	n/a	1.32	1.35	n/a
`f64`	1.00	1.00	n/a	1.05	1.05	n/a

TABLE VII
COMPUTE-BOUND FLOATING-POINT CHAINS, RATIO TO THE BASIS TYPE, THROWING OPERATORS VERSUS THE DEFERRED (OVERFLOWING) FAMILY.

		CUDA (RTX 3060)		SYCL (CPU device)	
Type	Op	Throwing	Deferred	Throwing	Deferred
`f32`	Add	18.31	5.39	173.03	1.68
`f32`	Sub	17.66	5.37	171.30	2.11
`f32`	Mul	8.71	2.69	127.59	1.43
`f32`	Div	2.22	1.05	50.01	1.13
`f64`	Add	4.00	4.21	179.70	2.73
`f64`	Sub	4.01	4.21	234.65	2.85
`f64`	Mul	2.77	2.94	102.24	2.05
`f64`	Div	1.49	1.55	17.00	1.05

buy little (1.21x to 2.96x) because a throwing operator already compiles to a well-predicted branch. On the CPU device the dominant cost is lost vectorization: where the baseline cannot vectorize either the throwing types cost 1.05x to 1.88x, and where it can (`i16`, `i32`) the gap is the SIMD width times the check cost, 8.8x to 9.4x, with `u32` between the two at 4.3x.

3) Compute-bound floating-point chains: Table VII is the central result of RQ3. A floating-point chain's baseline operation is a single hardware instruction, so the fixed cost of the classifier multiplies it. On the GPU the throwing `f32` chains cost 2.2x (division) to 18.3x and the `f64` chains 1.5x to 4.0x, and the deferred family cuts `f32` to 2.7x to 5.4x with division at parity. On the CPU device the throwing chains lose vectorization entirely and cost 17x to 235x. The deferred family, whose `overflowing_*` operations are branch-free, vectorizes with the baseline and lands the same chains at 1.05x to 2.85x, two orders of magnitude better, while still detecting every error at the boundary.

4) Answer to RQ3: Two responses built on the same detection code make device execution possible: a shared device-side error record with host-side rethrow, which preserves the throwing model across the kernel boundary, and a branch-free deferred family, which preserves vectorization. In the memory-bound regime typical of real kernels the GPU checks are free above 8 bits apart from 128-bit multiplication and `f64` division. In the compute-bound regime the throwing types cost 1.3x to 2.9x for integers through 64 bits on a GPU, and the deferred family is what makes checked floating point viable on a vectorizing device, at 1.1x to 2.9x instead of up to 235x.

VI. DISCUSSION AND LESSONS FOR PRACTITIONERS

A. Fixed Width Is the Point

The comparison with Boost.SafeNumerics shows that per-operation cost is similar when the two libraries check the same thing. What differs is the semantics: promotion-based checking cannot see an 8-bit or 16-bit fault that the surrounding expression computes in `int`, and 96 of the 130 in-scope CVEs in our corpus are 32-bit faults that promotion would not have widened away on the platforms where they shipped, while the 30 signedness faults are ones promotion actively hides. SafeInt, the other fixed-width comparator, agrees with the library on every runtime probe. Practitioners choosing a checked type should ask at which width the check is applied, not only how fast it is. Within a width, division and modulo are free, addition and subtraction cost one predictable branch, and multiplication, which needs a wide product or a division, is the check to watch.

B. Decoupling Detection from Response Is the Enabling Mechanism

On the host, offering saturating, checked, and overflowing families beside the throwing operators looks like convenience; on a device it is the difference between a library that works and one that does not. Exceptions do not exist in a kernel, so the response must be relocated to the host (`device_error_context`) or deferred to a boundary (`deferred_errors`), with the detection code untouched in both cases. The floating-point chains (Table VII) are the sharpest illustration: the same classifier, with the branch moved from every operation to once per work item, goes from unusable to under 3x on the vectorizing device, and the host ablation shows the same split in the vectorizer's diagnostics. We suggest the lesson applies to any checked-arithmetic design intended for performance-sensitive code: make the response a policy, and make at least one policy a value.

C. Bounded Domains Catch What Checks Cannot

The CWE-682 control doubles its prevented count when domains are declared. Any value later used as an index, offset, or length into a buffer of independently known size has a domain, and declaring it moves the failure from the memory operation to the point the value is formed; the bound must come from that independent quantity, never from the correct answer.

D. Costs of Adoption

The explicit-conversion rule adds casts at every boundary with legacy code, and the mixed-width prohibition forces a decision wherever C++ would have promoted. The corpus suggests that friction is the mechanism: 17 of the 30 CWE-681 entries were prevented because the port would not compile.

VII. Threats to Validity

1) Construct validity: For RQ1, "prevented" means that the arithmetic root cause becomes a typed, catchable failure at the faulting operation. A thrown exception is strictly better than silent corruption, but is a denial of service if unhandled, and we do not claim the original program would have been immune. For RQ2 and RQ3, the ratio measures the non-violating path by construction; the cost of actually throwing is not measured and is deliberately out of scope.

2) Internal validity: Reproduction fidelity is the central threat to RQ1. We do not re-run the original programs but port only the operation the patch cites, at the width the advisory assumes, assert the specific disclosed wrong value, and commit every arm as a test that runs in CI. For RQ2, the host numbers come from shared CI runners and reflect a single run each. Five repetitions with median reporting stabilize the central figure but cannot remove the variance of a shared virtual machine, so differences of a few percent should be read as noise. The grid's scalar shape forces a per-element spill whose cost depends on stack layout. In one build the throwing `u32` and `u64` measured 4.3x to 4.9x because the AArch64 load/store optimizer merged two spill reloads into one `ldp` straddling two prior stores, defeating store-to-load forwarding, disabling that pass returned them to 1.01x. An isolated outlier cell of Table II may therefore reflect the harness rather than a check; the stream-shape comparisons do not share this exposure. The stream-shape ratios on vectorized 32-bit loops rest on baselines near 0.1 ns per element and vary by up to 30% between runs, so their second digit is not meaningful. For RQ3, each device is a single machine and a single run.

3) External validity: The corpus is a purposive sample of CVEs whose root cause reduces to a self-contained snippet, so the prevention rates are conditional on that reducibility and are not rates over all CVEs tagged with a CWE; the out-of-scope funnel, the negative control, and the not-prevented cases carry the credibility, not a claim of random sampling. The device results come from one consumer GPU and one CPU OpenCL device.

4) Conclusion validity: Because n is small and the observed rate is at the ceiling, the rate and its Wilson interval are the result; the McNemar p values are ceilinged by design.

VIII. Related Work

1) Understanding and finding integer faults: Dietz et al. [3] found integer overflow pervasive in mature C/C++, including intentional wrapping that complicates detection; their tool became UBSan's integer checks [9]. KINT [4], RICH [5], and SmartFuzz [6] find integer errors statically, by binary instrumentation, and by test generation; Coker and Hafiz [7] and IntPatch [8] repair them by transformation. These tools find or repair faults in code already written; SafeNumbers keeps its checks on in the shipped program, and we evaluate it against the CVE record those studies characterized.

2) Compiler and language mechanisms: Rust [16] panics on overflow in debug builds and wraps in release, with the `checked_`, `saturating_`, `wrapping_`, and `overflowing_` methods the library's call-site families follow by name; Ada and SPARK [17] provide range subtypes, with GNATprove proving absence of overflow on Why3's back end. C++26 contracts [26] standardize the violation-handling model the library's response architecture anticipates; library hardening [27] and constrained numbers [28] are adjacent WG21 efforts.

3) Safe-integer libraries: Boost.SafeNumerics [11] is the comparator throughout. Microsoft's SafeInt [12] is a widely deployed checked-integer template that throws or calls a handler. `bounded::integer` [14] tracks value bounds at compile time so that overflow is impossible by construction, at the cost of result types that widen. `type_safe` [15] rejects lossy and implicit conversions. Intel safe-arithmetic [13] carries an interval in the type, proves that every operation's result interval fits its destination at compile time, and admits runtime values only through a checked `match`; it is explicitly pre-release, and Section V-B reports both what that model catches and one case it misses. None provides floating-point classification, 128-bit types, or device execution.

4) Verified arithmetic: The GMP verification in Why3 [19], [20] is the model for our proofs, Glesner [21] supplies the lifting argument for multiplication, and refinement types [29] inspired the bounded types' role as a precondition layer, though our bounds are checked dynamically.

IX. Conclusion and Future Work

We designed and implemented SafeNumbers, a C++20 library of checked fixed-width numeric types that keeps its checks in the shipped program, checks at the declared width, and decouples the detection of a fault from the response to it, and we evaluated each decision against a measurable claim. It prevents 126 of 130 in-scope CVEs; its median per-operation cost on 64-bit hosts is 1.00x to 1.16x through 64 bits; and on CUDA and SYCL devices the same types run unchanged, with a branch-free deferred response bringing compute-bound floating-point chains on a vectorizing SYCL device from two orders of magnitude slower than native to under 3x. Building and evaluating the library together improved both: the evaluation exposed an inlining failure in the signed throwing operators, now fixed, and led the bounded types to hand their declared domain to the optimizer. Future work is a random corpus draw, a controlled-machine replication of the host grid with dispersion statistics, and a discrete-GPU SYCL run.

Acknowledgment

The first author is an employee of The C++ Alliance, which funded the development of SafeNumbers and the evaluation reported here. The authors thank the C++ Alliance for its continued support of open-source C++ libraries and tools. Anthropic's Claude (Opus 5, Opus 5.5, Fable 5.1, and Sonnet 5.5 models) were used to draft and edit prose throughout this paper. The authors verified every claim and number against the artifacts cited in Section IV.

References

[1] Cybersecurity and Infrastructure Security Agency and Federal Bureau of Investigation, "Product security bad practices," Oct. 2024. [Online]. Available: https://www.cisa.gov/resources-tools/resources/product-security-bad-practices

[2] MITRE Corporation, "CWE Top 25 most dangerous software weaknesses," 2024. [Online]. Available: https://cwe.mitre.org/top25/

[3] W. Dietz, P. Li, J. Regehr, and V. Adve, "Understanding integer overflow in C/C++," in *Proc. 34th Int. Conf. Software Engineering (ICSE)*, Zurich, Switzerland, 2012, pp. 760–770.

[4] X. Wang, H. Chen, Z. Jia, N. Zeldovich, and M. F. Kaashoek, "Improving integer security for systems with KINT," in *Proc. 10th USENIX Symp. Operating Systems Design and Implementation (OSDI)*, Hollywood, CA, USA, 2012, pp. 163–177.

[5] D. Brumley, T. Chiueh, R. Johnson, H. Lin, and D. Song, "RICH: Automatically protecting against integer-based vulnerabilities," in *Proc. Network and Distributed System Security Symp. (NDSS)*, San Diego, CA, USA, 2007.

[6] D. Molnar, X. C. Li, and D. A. Wagner, "Dynamic test generation to find integer bugs in x86 binary Linux programs," in *Proc. 18th USENIX Security Symp.*, Montreal, Canada, 2009, pp. 67–82.

[7] Z. Coker and M. Hafiz, "Program transformations to fix C integers," in *Proc. 35th Int. Conf. Software Engineering (ICSE)*, San Francisco, CA, USA, 2013, pp. 792–801.

[8] C. Zhang, T. Wang, T. Wei, Y. Chen, and W. Zou, "IntPatch: Automatically fix integer-overflow-to-buffer-overflow vulnerability at compile-time," in *Proc. 15th European Symp. Research in Computer Security (ESORICS)*, Athens, Greece, 2010, pp. 71–86.

[9] LLVM Project, "UndefinedBehaviorSanitizer," Clang documentation. [Online]. Available: https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html

[10] Free Software Foundation, *Using the GNU Compiler Collection (GCC)*, sec. "Built-in functions to perform arithmetic with overflow checking" and option `-ftrapv`. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/

[11] R. Ramey, "Boost.SafeNumerics," Boost C++ Libraries, 2019. [Online]. Available: https://www.boost.org/doc/libs/release/libs/safe_numerics/

[12] D. LeBlanc, "SafeInt," Microsoft, commit 72f0745. [Online]. Available: https://github.com/dcleblanc/SafeInt

[13] Intel Corporation, "safe arithmetic: A general purpose C++20 library for writing safe and bug-free code," pre-release, commit c3b6e94. [Online]. Available: https://github.com/intel/safe-arithmetic

[14] D. Stone, "bounded::integer." [Online]. Available: https://github.com/davidstone/bounded-integer

[15] J. Müller, "type_safe." [Online]. Available: https://github.com/foonathan/type_safe

[16] The Rust Project, "Module std::num and the primitive integer types," *The Rust Standard Library*. [Online]. Available: https://doc.rust-lang.org/std/num/

[17] AdaCore, *SPARK 2014 User's Guide*. [Online]. Available: https://docs.adacore.com/spark2014-docs/html/ug/

[18] J.-C. Filliâtre and A. Paskevich, "Why3: Where programs meet provers," in *Proc. 22nd European Symp. Programming (ESOP)*, ser. LNCS, vol. 7792. Rome, Italy: Springer, 2013, pp. 125–128.

[19] R. Rieu-Helft, C. Marché, and G. Melquiond, "How to get an efficient yet verified arbitrary-precision integer library," in *Verified Software: Theories, Tools, and Experiments (VSTTE)*, ser. LNCS, vol. 10712. Springer, 2017, pp. 84–101.

[20] G. Melquiond and R. Rieu-Helft, "A Why3 framework for reflection proofs and its application to GMP's algorithms," in *Proc. 9th Int. Joint Conf. Automated Reasoning (IJCAR)*, ser. LNCS, vol. 10900. Oxford, UK: Springer, 2018, pp. 178–193.

[21] S. Glesner, "Finite integer computations: An algebraic foundation for their correctness," *Formal Aspects of Computing*, vol. 18, no. 2, pp. 244–262, 2006.

[22] E. B. Wilson, "Probable inference, the law of succession, and statistical inference," *J. Amer. Statist. Assoc.*, vol. 22, no. 158, pp. 209–212, 1927.

[23] Q. McNemar, "Note on the sampling error of the difference between correlated proportions or percentages," *Psychometrika*, vol. 12, no. 2, pp. 153–157, 1947.

[24] Google, "Benchmark: A microbenchmark support library," v1.9.1. [Online]. Available: https://github.com/google/benchmark

[25] National Institute of Standards and Technology, "National Vulnerability Database: Vulnerabilities API 2.0." [Online]. Available: https://nvd.nist.gov/developers/vulnerabilities

[26] J. Berne, T. Doumler, A. Krzemieński *et al.*, "Contracts for C++," ISO/IEC JTC1/SC22/WG21, P2900R14, 2025. [Online]. Available: https://wg21.link/p2900r14

[27] K. Varlamov and L. Dionne, "Standard library hardening," ISO/IEC JTC1/SC22/WG21, P3471R1, 2024. [Online]. Available: https://wg21.link/p3471r1

[28] L. Valenty, "Constrained numbers," ISO/IEC JTC1/SC22/WG21, P2993. [Online]. Available: https://wg21.link/p2993

[29] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton Jones, "Refinement types for Haskell," in *Proc. 19th ACM SIGPLAN Int. Conf. Functional Programming (ICFP)*, Gothenburg, Sweden, 2014, pp. 269–282.

[30] M. Borland, "SafeNumbers: Safer fundamental numeric types in C++20," The C++ Alliance, 2026. [Online]. Available: https://github.com/cppalliance/safe_numbers, documentation at https://develop.safe-numbers.cpp.al/