EXCVATE: Spoofing Exceptions and Solving Constraints to Test Exception Handling in Numerical Libraries

Jackson Vanover*, James Demmel†, Xiaoye Sherry Li‡, and Cindy Rubio-González*
*University of California, Davis. Email:{jdvanover,crubio}@ucdavis.edu
†University of California, Berkeley. Email: demmel@berkeley.edu
‡Lawrence Berkeley National Laboratory. Email: xsli@lbl.gov

***Abstract*—Testing a numerical library's exception handling is often left to its regression tests. However, designing floating-point inputs that exercise exceptional behavior is difficult. Furthermore, existing input generation techniques are designed with the view that any exception-causing input is of interest when most are unremarkable from the standpoint of exception handling: most functions should handle exceptions correctly by construction, i.e., by returning exceptional values (`NaN`, $\pm$`Inf`), due to IEEE 754's mandate that most operations propagate such values. To test library exception handling, we propose an approach which – given only a set of regression test executables and a set of function prototypes – cheaply identifies potential failures via exception-spoofing and then reifies these failures using an off-the-shelf SMT solver that generates concrete inputs inducing buggy behavior. We implement this approach in a prototype tool EXCVATE, present an evaluation that targets 26 BLAS functions across three implementations, two compilers, and multiple compiler optimizations, and ultimately identify exception-handling failures in five functions in multiple BLAS versions.**

I. INTRODUCTION

Numerical software is ubiquitous. It is therefore important to ensure its correctness. High-profile failures of mission-critical numerical software emphasize this importance [1], but designing algorithms to be error-free over all possible inputs is difficult and leads to programs that waste time pre-emptively checking for special cases that are uncommon in practice [2, 3]. In light of this, the IEEE 754 Standard defines five floating-point (FP) exceptions [4]. The correctness of numerical software is therefore contingent upon responding to these exceptions in a predictable and consistent way, i.e., **Exception Handling**. Because the correctness of such software is built upon the correctness of its building blocks, it is of the utmost importance to ensure the soundness of the exception-handling strategies implemented by numerical libraries.

Three of the five exceptions generate an **Exceptional Value** (EV): either $\pm$`Inf` or `NaN`. Furthermore, the standard explicitly defines most operations on `Inf` and `NaN` operands to support propagation, thus presenting an attractive definition for a sound exception-handling policy:

Definition 1. *When an Overflow, Divide-by-Zero, or Invalid exception occurs, a* ***sound exception-handling policy*** *notifies users by either (1) the presence of EVs in the output, or by (2) some library-specific reporting mechanism triggered by checking for their presence at some point during the execution.*

This is the definition adopted by the reference implementations for the widely-used LAPACK and BLAS numerical libraries [5] and is the one we consider in this work. Testing exception handling can then be framed as an input generation problem: how can we find inputs that cause exceptions for which the policy described in Definition 1 is violated?

Many works propose approaches to finding inputs that cause FP exceptions [6, 7, 8, 9, 10, 11, 12]. However, such approaches are designed with the view that any exception-causing input is of interest, i.e., all exceptions are bugs. This work offers a complementary perspective: if one instead views exceptions as *inevitabilities*, the focus shifts to ensuring the correctness of the mechanisms for exception handling, i.e., compliance with Definition 1. In such a setting, many exception-causing inputs become unremarkable: *most code should handle exceptions correctly by construction due to IEEE 754's mandate that most operations propagate EVs.* This make existing input generation approaches a poor fit. Our key insight then comes from separating the input generation problem into two challenges:

Challenge 1: *Finding violations of the exception-handling policy.* Because generating exception-causing inputs is expensive and most exceptions are handled correctly, we leverage binary rewriting to cheaply spoof mid-execution FP exceptions in order to check if they are handled according to the policy. For a given function execution, we identify each instruction execution that could generate an EV due to an exception and, for each of them, replay the function execution while overwriting the write register with EVs to "spoof" the exception.

Challenge 2: *Finding concrete inputs that reify the spoofed exception while preserving the control flow that leads to the policy violation.* For the few cases in which the spoofed exception is not handled correctly, we formulate SMT queries that capture the constraints on symbolic FP inputs that reify the exceptions while preserving control flow and feed them to an off-the-shelf solver to yield concrete inputs.

Because exception-spoofing is a dynamic testing approach, we encounter a third challenge:

Challenge 3: *Finding representative inputs to bootstrap the testing.* The power of this exception-spoofing approach is contingent upon quality test inputs which capture the conditions and edge cases that the software might encounter. To this end, we leverage developer-written regression test binaries.

2576-2265/25/$31.00 ©2025 IEEE
DOI 10.1109/ARITH64983.2025.00026

By addressing these challenges, we introduce a novel testing approach which requires no source code. We implement our approach in the tool EXCVATE (EXCeptional VAlue TEster) and perform an experimental evaluation targeting 26 functions from the BLAS, i.e., the Basic Linear Algebra Subprograms which are the de facto standard low-level routines for linear algebra libraries. This evaluation targets three BLAS implementations (reference BLAS [13], BLIS [14], and OpenBLAS [15]) across both GNU and Intel compilers and multiple compiler optimizations, ultimately revealing five exception-handling failures in multiple version of the BLAS.

In summary, the contributions of this paper are:

- A novel binary instrumentation based approach to testing exception handling in numerical libraries using both low-cost exception spoofing and powerful constraint solving applied to existing library regression tests.
- An implementation of our approach in the tool EXCVATE.
- An experimental evaluation of EXCVATE over a cross product of 598 (function, implementation, compiler, optimizations) tuples that reveals five exception-handling failures in multiple versions of the BLAS.

II. PRELIMINARIES

A. Floating-Point Exception Handling in Hardware/Software

Three of the five FP exceptions defined by the IEEE 754 standard generate EVs and are supported in compliant hardware. These are *Overflow*, *Divide-by-Zero*, and *Invalid* exceptions. Unlike the *Inexact* and *Underflow* exceptions whose default results incur a graceful degradation that allows computations to continue unhindered, these three exceptions result in an abrupt loss of data by generating a `NaN` or $\pm$`Inf`. Furthermore, the idiosyncracies of operations on EVs as defined by the standard can cause unexpected program behaviors [16]. For instance, trichotomy does not apply in comparisons involving `NaN` which evaluate to *unordered* and many operations on one or more EVs are themselves defined to be *Overflow* or *Invalid* exceptions, thus resulting in the propagation of EVs.

At the software level, one can check for the occurence of an exception by checking for the EVs generated/propagated due to the mandated hardware behavior described above. While checking status flags presents an alternative strategy, previous works note that this is not always possible due to to variability in hardware [5, 10, 11, 17, 18]. It is also the policy adopted by the widely-used LAPACK and BLAS numerical libraries [5].

B. SMT Solvers and Symbolic Execution for Floating-Point Input Generation

In particular, we are interested in FP input generation techniques that leverage SMT (Satisfiability Modulo Theories) solvers. SMT solvers are a generalization of SAT (Boolean Satisfiability) solvers which determine the existence of a satisfying assignment for a set of constraints expressed over a set of boolean variables. For instance, $(A \wedge \neg A)$ is not satisfiable but $(A \wedge B)$ is with an assignment of $A \leftarrow true, B \leftarrow true$. SMT solvers allow for more complex expressions interpreted within some formal theory in first-order logic with equality. For instance, SMT solvers with the theory of floating point can reason about constraints like $(z = x{+}y) \wedge (z \text{ is } \texttt{NaN})$ and yield a satisfying assignment like $x \leftarrow \texttt{NaN}, y \leftarrow 2.0, z \leftarrow \texttt{NaN}$ or $x \leftarrow \texttt{Inf}, y \leftarrow -\texttt{Inf}, z \leftarrow \texttt{NaN}$.

SMT solvers form the basis of symbolic execution which has been used as a coverage-guided input generation technique. By interpreting a program's execution over symbolic inputs, each program path can then be expressed as a set constraints over those inputs. An SMT solver can then find a satisfying assignment of input values for a set of constraints to reify the execution of the corresponding path. While powerful, scalability is hindered by the exponential number of solver invocations corresponding to the exponential number of paths ("path explosion") [8]. In the following motivating example, we show how state-of-the-art FP tools leverage the power of SMT solvers through symbolic execution to generate exception-causing FP inputs and we show the limitations of these tools for the problem of testing exception handling.

C. A Motivating Example

We consider the challenge of testing exception handling in BLAS and LAPACK implementations. In their proposal for a consistent adoption of the policy described in Definition 1 [5], the maintainers of the reference implementations discuss the challenge that input generation presents to thorough testing. While SMT-based approaches are a natural fit for this, symbolic execution tools with floating-point support [6, 7, 8, 9] contain a number of limitations. To demonstrate these limitations, we will use symbolic execution to test the reference implementation of the BLAS function `sgbmv` which performs the operation $\mathbf{y} := \alpha\mathbf{A}\mathbf{x} + \beta\mathbf{y}$ for banded matrices.

First, because the aforementioned tools are all restricted to C code, we must translate all source code involved in `sgbmv` from Fortran to C. Second, for each operation `op` that could trigger an Invalid exception, we must manually add the following instrumentation:

```
if ( (x op y) != (x op y) ) {...}
z  = x op y; // targeted line
```

In this way, generating inputs that cause Invalid exceptions is translated to a code coverage task suitable for symbolic execution – in order to execute the code guarded by the conditional, inputs must be chosen such that `x op y` results in a NaN. Third, because these tools make the simplifying assumption that the only input is a fixed number of floating-point scalars, we must simplify the inputs by constructing a test harness that hardcodes the values of any non-float variables. Figure 4 describes all inputs for `sgbmv`. Because the values of these non-float variables determine the dimensions of `A`, `X`, and `Y` as well as the banded structure of `A`, let us call them *parameterizations*. Hence, more complete testing requires a symbolic execution search for each parameterization. Furthermore, since the values that define a parameterization are dependent on one another and we want to test only well-formed and representative parameterizations, we perform five-

AssemblyLocation	Disassembly	SourceLocation	Event	EVcount
srotmg_::0x00000008b	mulss xmm8, xmm3	bla_rotmg.c:106	G---	1
srotmg_::0x000000097	movaps xmm9, xmm8	bla_rotmg.c:108	-P-r	2
srotmg_::0x00000009b	mulss xmm9, xmm6	bla_rotmg.c:108	-P-r	2
srotmg_::0x0000000a7	andps xmm9, xmm2	bla_rotmg.c:110	-P-r	2
srotmg_::0x0000000b3	ucomiss xmm9, xmm10	bla_rotmg.c:110	---r	2
srotmg_::0x0000000c3	unpcklps xmm6, xmm8	bla_rotmg.c:113	-P-r	3
...				
srotmg_::0x0000000df	shufps xmm1, xmm0, 0x55	bla_rotmg.c:113	-P-r	14
srotmg_::0x0000000e3	mulss xmm1, xmm0	bla_rotmg.c:116	-P-r	14
srotmg_::0x0000000e7	movss xmm5, dword ptr [rip+0x65d71]	bla_rotmg.c:116	--K-	12
srotmg_::0x0000000ef	movaps xmm6, xmm5	bla_rotmg.c:116	--K-	9
srotmg_::0x0000000f2	subss xmm6, xmm1	bla_rotmg.c:116	-P-r	10
srotmg_::0x0000000f6	xorps xmm1, xmm1		--Kr	6
srotmg_::0x0000000f9	ucomiss xmm6, xmm1	bla_rotmg.c:118	---r	6
srotmg_::0x000000029	xorps xmm0, xmm0	bla_rotmg.c:157	--Kr	3

Fig. 1: An excerpt of an event trace generated by EXCVATE for one of the buggy `srotmg` cases described in Section IV-B in which a lack of EVs in both the inputs and outputs mask a number of mid-execution exceptions. The four character `Event` code indicates which of four events occurred: EVs were **G**enerated, **P**ropagated, **K**illed, and/or were present in the **r**ead operands. `EVcount` describes how many EVs were present globally after the execution of the instruction.

second searches for each unique parameterization used to test `sgbmv` in the BLAS regression tests.

Our testing of `sgbmv` using symbolic execution covered 1,547 parameterizations and took four hours to generate 45,933 inputs. Figure 2 depicts the results of evaluating `sgbmv` on these inputs. Notably, only 1.5% of them revealed an exception-handling failure in which implicit zeroes in the banded storage format of a "wide" **A** lead to a failure to propagate `NaN`s (See Section IV-B(2) for a full description).

In summary, testing `sgbmv` via symbolic execution required non-trivial engineering for compatibility with existing tools and required four hours of total execution time. Compare this to our prototype tool, EXCVATE, which implements the approach presented in this paper: EXCVATE requires only the binary executables of the library regression tests and simple text files describing the inputs of the target functions (Figure 4). Moreover, EXCVATE reaches the same results for `sgbmv` for the same 1,547 parameterizations in less than one minute of total testing time while also generating a detailed trace for each exception-handling failure that descibes the flow of EVs through individual instructions (Figure 1).

III. APPROACH

We have implemented our approach in the prototype tool EXCVATE (Figure 3). EXCVATE takes as input a set of regression test executables and a set of function prototypes. The *Execution Selector* chooses a representative subset of function executions from the regression tests. Then, for each function execution, the *Exception Spoofer* identifies potential exception sites and, for each of these sites, replays the execution while spoofing an FP exception at the site. Warnings are issued for any spoofed exceptions that are not handled correctly. Finally, the *Input Generator* checks each warning by generating SMT queries that reify the exception while preserving the control flow, feeding these queries to an off-the-shelf SMT solver to determine if the exception failure is feasible, and finally

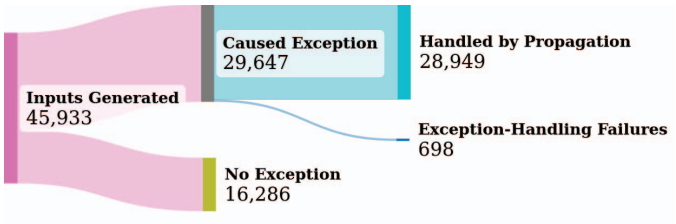

Fig. 2: A Sankey diagram showing the results of using a representative and state-of-the-art FP input generation technique for testing the exception-handling of the `sgbmv` function. Four hours of symbolic execution generated 45,933 inputs; of these, only 698 (1.5%) reveal an exception-handling failure that was found by EXCVATE in less than one minute.

generating event traces for any exception-handling failures encountered from the solver-generated concrete inputs. We now discuss each component in detail.

Component 1: Execution Selector

Input: TestBin, FuncProtos
Output: SelectedExecutions

```
1 processedIDs ← ∅;
2 for FuncExec ∈ TestBin do
3     name = getName(FuncExec);
4     if name ∈ FuncProtos then
5         args = getArgs(FuncExec, FuncProtos[name]);
6         id = getID(FuncExec);
7         if id ∉ processedIDs then
8             processedIDs ← id;
9             SelectedExecutions ← (name, args);
```

The goal of the Execution Selector is to select function executions that are "interesting" for our testing and to ignore those that are "redundant". The inputs are binary

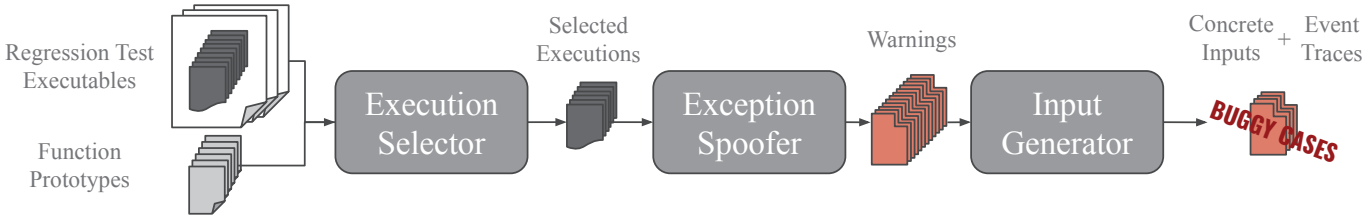

Fig. 3: An overview of the approach to testing exception handling implemented in the prototype tool EXCVATE.

```
TRANS c 8  in    1
M     i 32 in    1
N     i 32 in    1
KL    i 32 in    1
KU    i 32 in    1
ALPHA r 32 in    1
A     r 32 in    ${LDA} * ${N}
LDA   i 32 in    1
X     r 32 in    if [ ${TRANS} == N ]; then 1 + ( ${
    N} - 1 ) * abs(${INCX}); else 1 + ( ${M} - 1 ) *
     abs(${INCX}); fi
INCX  i 32 in    1
BETA  r 32 in    1
Y     r 32 inout if [ ${TRANS} == N ]; then 1 + ( ${
    M} - 1 ) * abs(${INCY}); else 1 + ( ${N} - 1 ) *
     abs(${INCY}); fi
INCY  i 32 in    1
```

Fig. 4: A prototype file for the BLAS function `sgbmv` used for the motivating example in Section II-C. Prototype files contain a row for each input/output variable describing the variable name, its type (**c**haracter, **i**nteger, or **r**eal), the number of bits per element, the "intent" for real variables (in, out, inout, return), and the number of elements.

executables for the library's regression tests and a set of prototypes for each of the target functions to be tested (Figure 4).

For each function execution for which there is a provided prototype, the Execution Selector assigns an ID: a hash of the sorted concatenation of unique basic block addresses executed. This is because executions taking the same path are equivalent from the perspective of the testing technique to be described in the remainder of Section III; this will fall out naturally by construction. Hence, by using this proxy for path coverage, we can sort all function executions into equivalence classes from which single representatives are selected (line 9).

Component 2: Exception Spoofer

```
   Input: SelectedExecutions
   Output: Warnings
1  for FuncExec ∈ SelectedExecutions do
2      sites, timelimit = getPotentialExSites(FuncExec);
3      for s ∈ sites do
4          fail = spoofAndCheck(FuncExec, s, timelimit);
5          if fail then
6              Warnings ← (FuncExec, s);
```

The goal of the Exception Spoofer is to identify pairs (FuncExec,s) **for which, given the control flow induced by the inputs of** FuncExec**, an exception at exception site** s **would result in an exception-handling failure violating the policy described in Definition 1.** For each selected function execution, a first baseline execution identifies potential exception sites s and a time limit for the testing to follow (line 2). Potential exception sites are instructions for which there exist FP inputs that could trigger an exception that we can spoof by overwriting the output with an EV. Each site s is a tuple (id, n, o, i, j) where id is the ID calculated by the Execution Selector, n is the symbol name of the containing routine, o is the instruction's offset from the containing routine's base address, i is the instruction's execution count for the current function execution, and j is the index of the instruction's output in the destination SIMD register.

For each s, the execution is replayed with instrumentation to spoof an exception and to check if the policy described in Definition 1 is respected (line 4). The first property (propagation to outputs) is checked by inspecting the values of variables defined as "out", "inout", or "return" in the provided prototype. The second property (a library-specific reporting mechanism) is checked by instrumenting any error routines whose names were given as optional inputs to EXCVATE. Any failures result in warnings (line 6).

Component 3: Input Generator

```
   Input: Warnings
   Output: GeneratedInputs, EventTraces
1  for FuncExec, s ∈ Warnings do
2      {q1, q̂1, q2, q̂2} = getQueries(FuncExec, s);
3      for q ∈ {q1, q̂1, q2, q̂2} do
4          sat = SMTsolver(q);
5          if sat then
6              ins = getFPinputs(sat);
7              fail, trace = getEventTrace(FuncExec, ins);
8              if fail then
9                  GeneratedInputs ← ins;
10                 EventTraces ← trace;
```

The goal of the Input Generator is to check each warning from the Exception Spoofer by attempting to reify the spoofed exception while preserving the control flow that led to the exception-handling failure. For each site for which spoofed exceptions were not handled correctly, the execution+spoofing is replayed with additional instrumentation to initiate symbolic values for all FP inputs, to collect constraints

```
vmulpd xmm1, xmm1, xmmword ptr [rdi+rdx*8]
      ;( assert ( = xmm1_72_b64_0 ( fp.mul rm xmm1_71_b64_0 ( ( _ to_fp 11 53 ) #b1011111111100000001001...
      ;( assert ( = xmm1_72_b64_1 ( fp.mul rm xmm1_71_b64_1 ( ( _ to_fp 11 53 ) #b1011111110011000001011...
vaddpd xmm0, xmm1, xmm0
      ;( assert ( = xmm0_158_b64_0 ( fp.add rm xmm1_72_b64_0 xmm0_157_b64_0 ) ) )
      ;( assert ( = xmm0_158_b64_1 ( fp.add rm xmm1_72_b64_1 xmm0_157_b64_1 ) ) )
vshufpd xmm1, xmm0, xmm0, 0x1
      ;( assert ( = xmm1_73_b64_0 xmm0_158_b64_1 ) )
      ;( assert ( = xmm1_73_b64_1 xmm0_158_b64_0 ) )
vaddsd xmm0, xmm0, xmm1
      ;( assert ( = xmm0_159_b64_0 ( fp.add rm xmm0_158_b64_0 xmm1_73_b64_0 ) ) )
      ;( assert ( = xmm0_159_b64_1 xmm0_158_b64_1 ) )
      ;( assert ( fp.isNaN xmm0_159_b64_0 ) )          <- reifies an "Invalid" exception!
```

Fig. 5: Example translation from executed assembly to SMT constraints (depicted as code comments following each instruction). The SMT variable name `xmm0_159_b64_0` is interpreted as "the 64-bit quadword at index 0 in the 159th write of register `xmm0`". The expressions prefixed by `#b` are binary literals, implying the memory operand in `vmulpd` contains FP data that is not affected by the program's symbolic inputs. Integer instructions will generate no constraints. The constraint in the last line reifies a spoofed exception while the others preserve the control flow.

on those symbolic values that will preserve the control flow, and to formulate these into SMT queries with the additional constraint that the EV from the spoofed exception is actually generated (line 2). See Figure 5 for an example translation from FP instructions to SMT constraints.

In line 2, these constraints are formulated into four queries:

	Input EVs OK	No Input EVs
Omit Post-Exception Constraints	q_1	$\hat{q_1}$
Include Post-Exception Constraints	q_2	$\hat{q_2}$

Both q_2 queries constrain outputs to be non-EVs; however, in order to do so, they must encode the post-exception control flow that produces the outputs, thereby excluding the possibility of inputs that reify the exception while inducing different but still-failing post-exception control flow. On the other hand, the corresponding q_1 queries omit the post-exception constraints. This allows for such a possibility while introducing a chance of "false positives" where post-exception control flow induces correct exception handling. Both $\hat{q}$ queries constrain the inputs to be non-EVs; if SAT, they yield interesting cases where normal inputs and outputs mask mid-execution exceptions entirely.

In line 4, each of these queries is fed to an SMT solver with the theory of floating point. Note that with respect to the set of constraints they contain, the queries form a complete lattice ordered by inclusion with $\hat{q_1} \wedge q_2 = q_1$ and $\hat{q_1} \vee q_2 = \hat{q_2}$. This allows us to short-circuit the evaluation of queries when starting from bottom. Specifically: if q_1 is UNSAT, then all other queries are UNSAT; if either $\hat{q_1}$ or q_2 is UNSAT, then $\hat{q_2}$ is UNSAT. This is omitted from the above script for brevity.

Whenever the solver returns SAT, the satisfying assignment of FP values is given as input to the function (line 7) to generate an event trace like the one in Figure 1 and to filter out any "false positives" from the q_1 queries.

IV. Evaluation

Our evaluation is designed to answer the following two research questions:

> **RQ1** What kind of exception-handling failures, if any, can EXCVATE reveal in foundational linear algebra libraries?
>
> **RQ2** What effect do compilers and optimizations have on exception-handling failures?

A. Setup

Benchmarks & Compilers. We target the BLAS (Basic Linear Algebra Subprograms). The BLAS are a specification of low-level routines for common linear algebra operations and are the de facto standard for such routines. Here, we consider single-precision versions of the Level 1 and Level 2 BLAS which perform vector and matrix-vector operations, respectively. We omit functions which do not perform any exception-prone operations (`sswap`, `scopy`) and those which do not return any FP output or make any error-handling related calls (`isamax`). This leaves 26 functions. We test the implementations provided by the reference BLAS (v1.12) [13] as well as two high-performance implementations: BLIS (v1.0) [14], and OpenBLAS (v0.3.28) [15]. These versions are the current releases at the time of writing. We test each library across both GNU (`gfortran`/`gcc` v11.4.0) and Intel compilers (`ifx`/`icx` v2024.2.0) as well as multiple optimization levels. This makes for a cross product of 598 (function, library, compiler, optimization level) tuples.

Hardware. Our evaluation was conducted on a workstation with a 3.50 GHz Intel i7-3770K and 24 GB of RAM.

Implementation Details. EXCVATE implements the spoofing of Invalid exceptions and targets serial programs using the SSE2-SSE4 and AVX SIMD extensions. We exclude function executions with more than 32 FP inputs to manage scalability;

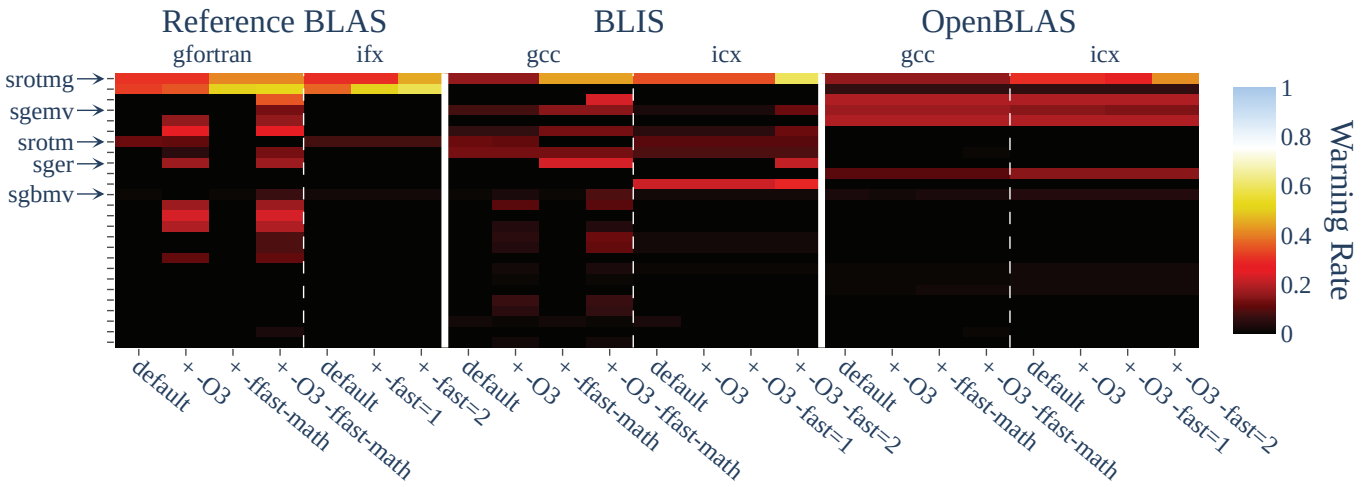

Fig. 6: Heatmaps describing the proportion of spoofed exceptions that resulted in exception failures, i.e., "warnings". Labels on the y-axis indicate the functions for which warnings resulted in true positives. The x-axis label "-fast" corresponds to the Intel compiler option `-fp-model=fast`. The `ifx` column lacks `-O3` sub-columns as that is the default optimization level.

TABLE I: Statistics aggregated by function; true positives are those warnings which were reified by the Input Generator, i.e., at least one of the four corresponding SMT queries was SAT.

Function Name	# Spoofs	# Warnings	# True Positives
`sger`	4,414	292	132
`sgemv`	91,098	11,168	636
`sgbmv`	117,694	4,421	295
`srotmg`	2,325	788	440
`srotm`	4,287	227	208
(21 Others)	330,894	9,617	0
(Total)	**532,491**	**23,693**	**1,711**

note that this is the maximum input size used in the most related works [10, 19]. EXCVATE is implemented using PIN [20] with CVC5 [21] as its SMT solver.

B. Results

Aggregate statistics are summarized in Table I. EXCVATE's Execution Replayer spoofed a total of 532,491 exceptions of which 23,693 (4.4%) resulted in a warning. Figure 6 depicts the warning rate across the 598 (function, library, compiler, optimization level) tuples. This reinforces our assumption that most code handles exceptions correctly. We also see differences in warning rates between compilers and a pattern of increasing warning rates as more aggressive optimizations are applied. Of the 23,693 warnings, EXCVATE's Input Generator component automatically determined 1,711 to be true positives, i.e., for each of these 1,711, it was able to generate a concrete input that reified the spoofed exception and which was then incorrectly handled according to the policy stated in Definition 1. These true positives occur in 5/26 functions. The experiments took 12 wall-clock hours, 11 of which were spent in the Input Generator component in calls to CVC5.

We perform a reduction to equivalence classes based on the assumption that exception-handling failures resulting from exceptions in the same source code line are indicative of the same buggy behavior. This yields 23 classes, a manual inspection of which led us to the three findings summarized in Table II and described in more detail here:

TABLE II: Summary of manual analyses of true positives.

Functions Affected	Cause of Exception-Handling Failure
`srotmg`, `srotm`	poor design/documentation not accounting for EVs
`sgbmv`	implicit zeroes in the input matrix representation
`sger`, `sgemv`	compiler optimizations that changed control flow

(1) In `srotmg` and `srotm`, numerous exception-handling failures indicate that these functions were not designed or documented with careful consideration for EVs.

`srotmg` and `srotm` respectively generate and apply a modified Givens rotation. EXCVATE generated inputs causing various exception-handling failures in all tuples. This included inputs exposing two noteworthy bugs in `srotmg`: one in which inputs containing no EVs yielded normal outputs that were incorrect (relative error $> 2 \times 10^{14}$) in all tested versions of the reference BLAS and another in which inputs containing an `Inf` resulted in infinite loops in all default builds of all three libraries. Evidently, these Level 1 BLAS functions were not designed or documented with careful consideration for EVs. Because, in the reference BLAS and LAPACK implementations, `srotmg` is never called and `srotm` is called only once, our discoveries motivated discussions within the BLAS working group about more careful documentation and even possible deprecation rather than refactoring.

(2) In `sgbmv`, implicit zeros in the representation of banded matrices can cause exception-handling failures, the first such case documented for the dense BLAS.

`sgbmv` performs the operation $\mathbf{y} := \alpha\mathbf{A}\mathbf{x} + \beta\mathbf{y}$ for banded matrices. EXCVATE generated inputs with a "wide" $\mathbf{A}$ and a `NaN` in $\mathbf{x}$ that causes `sgbmv` exception-handling failures in all libraries. This is due to implicit zeros in the banded storage

format of $\mathbf{A}$. In all libraries, `sgbmv` is optimized to skip multiplying implicit zeros with elements of $\mathbf{x}$ and assume the output is zero whereas a naive implementation would propagate any `NaN`s in $\mathbf{x}$. This spurred discussions in the BLAS working group regarding a modification of the exception-handling plan for the dense BLAS [5] to be consistent with the SparseBLAS standard [22] in which multiple error-handling options are provided depending on the desired treatment of implicit zeros.

(3) In `sger` and `sgemv`, compiler optimizations change control flow in the presence of `NaN`s and cause exception-handling failures.

`sger` performs the operation $\mathbf{A} := \alpha\mathbf{x}\mathbf{y}^T + \mathbf{A}$. EXCVATE generated inputs with a `NaN` in $\mathbf{y}$ that cause `sger` exception-handling failures in both BLIS and the reference BLAS when using `-ffast-math` or `-fp-model=fast=2`. Inspecting the source of the reference BLAS reveals the cause. In the main nested loop, the inner loop is only executed if the corresponding element in $\mathbf{y}$ is nonzero, i.e., it is guarded by a conditional of the form `if(Y[i]!=0)`. When `Y[i]` is `NaN`, the control flow of the unoptimized and optimized code diverges at this comparison: the former executes the inner loop and the EV is propagated to the output as expected while the latter skips the computations and the EV is lost.

`sgemv` performs the operation $\mathbf{y} := \alpha\mathbf{A}\mathbf{x} + \beta\mathbf{y}$ for general matrices. EXCVATE generated inputs with a `NaN` in $\mathbf{x}$ that cause `sgemv` exception-handling failures in BLIS when compiled with `-ffast-math` or `-fp-model=fast=2`. In this case, BLIS source code is not helpful for root-cause analysis because it is obfuscated by the extensive use of functional macros. However, inspecting the divergence of the x86 instructions executed both with and without optimizations suggests the cause is similar to `sger` above. A comparison in the unoptimized/correct version is executed via the instruction sequence `VUCOMISS`→`JNZ`→`JNP` which executes one jump if the operands are equal and a different jump in the presence of `NaN`. In the optimized/incorrect version, this sequence becomes `VUCOMISS`→`JZ` which jumps to the same location in both cases. The bug is not present in the reference BLAS as there are no comparisons to any elements of $\mathbf{y}$.

These two instances concerning `sger` and `sgemv` highlight the interplay of comparison predicates and compiler optimizations in the presence of `NaN`; developers who are not carefully considering this can introduce subtle bugs into numerical code.

Having summarized these three findings, we return now to the two research questions toward which the design of our evaluation was oriented:

RQ1 EXCVATE automatically discovered inputs which caused exception-handling failures in 5/26 functions. Notably, EXCVATE found `srotmg` inputs that yielded relative errors of greater than 2×10^{14} and infinite loops. For 4/5 functions, exception-handling failures were only exercisable via function inputs containing EVs. This highlights the need for more careful exception handling for exceptions caused by EVs in function inputs.

RQ2 While the exception-handling behavior of the BLAS functions was consistent across compilers, EXCVATE discovered inputs for multiple functions that caused control flow divergence across compiler optimizations that resulted in exception-handling failures.

V. RELATED WORK

We divide relevant related work into three categories. The first category is *Input Generation*. Several works present approaches based on symbolic execution [6, 7, 8, 9]. While their optimizations for soundness and scalability vary, they are all alike in that they reduce the problem of generating exception-causing inputs to the problem of code coverage as described in Section II. In particular, Barr et al. [6] use an SMT solver with the theory of reals and searches the neighborhood around the solutions for FP inputs. Wu et al. [7] and Ma et al. [9] trade some soundness for scalabality using interval arithmetic, interval constraints, and manually-modeled elementary functions to facilitate a cheaper range-solver. Other works take a more dynamic approach [10, 12, 23]. FPDiff [23] performs differential testing between numerical libraries using multiple means of input generation, one of which involves EVs in function inputs; while FPDiff triggers discrepancies with such inputs, the correctness of exception handling is not discussed. FPBOXer [10] uses Bayesian optimization targeting GPU binaries while NumFuzz [12] uses grey-box fuzzing with a custom fitness function and mutation strategy; both trigger overflow and underflow exceptions. All of these works focus on simply triggering exceptions, not testing exception handling as is done with EXCVATE. Testing approaches that view correctness as the total absence of exceptions can use the aforementioned tools in an iterative scheme: generate exception-causing inputs, harden code to prevent triggered exceptions, repeat until the program under test is sufficiently "bulletproof". However, this can come at the cost of performance via the development of programs that waste time preemptively checking for special cases that are uncommon in practice [2, 3].

The second category is *Software Designs for Exception Handling At Runtime*. Relevant works include two design proposals for LAPACK [3, 5]. Both require adding checks in software but developers must decide where to check and how often. An answer to this question affects the soundness and performance of the library. Our tool is orthogonal to designing exception handling, and instead, facilitates automatically checking the correctness of said design and its implementation.

The third category is *Runtime Tools for Exception Handling During Execution and/or Debugging Post-Mortem*. Given a known exception-causing input, these tools support debugging the cause. Otherwise, they can run on top of the program in perpetuity at high overhead cost. FPSpy [24] targets x86 binaries using overloaded fault trapping and status bits; like the software design approaches, users must decide where to check and how often. FPChecker [25] (for LLVM-IR), BinFPE [18]/GPU-FPX [17] (for CUDA binaries), and FlowFPX [26] (for Julia) all check operation outputs for EVs while differing

in their supported targets. While these tools facilitate handling during runtime and/or debugging post-mortem, none of them are designed to test exception-handling mechanisms.

VI. Conclusion

We have introduced a novel binary instrumentation based approach to testing exception handling in numerical libraries using both low-cost exception spoofing and powerful constraint solving applied to existing library regression tests. We have implemented our approach in EXCVATE and presented an experimental evaluation targeting the BLAS over a cross product of 598 (function, implementation, compiler, optimizations) tuples that reveals five exception-handling failures in multiple versions of the BLAS. EXCVATE is available at https://github.com/ucd-plse/EXCVATE along with instructions to reproduce the results of our experiments.

Acknowledgments

Vanover and Rubio-González were supported by the U.S. Department of Energy Office of Science, Advanced Scientific Computing Research under award DE-SC0020286, and the National Science Foundation under awards CCF-1750983 and CNS-2346396. Li was supported in part by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) program through the FASTMath Institute under Contract No. DE-AC02-05CH11231 at Lawrence Berkeley National Laboratory. We would also like to thank Mark Gates, Igor Kozachenko, Julien Langou, Jason Riedy, Weslley da Silva Pereira, and the other members of the BLAS working group for discussions surrounding this work.

References

[1] T. Huckle and T. Neckel, Eds., *Bits and Bugs: A Scientific and Historical Review on Software Failures in Computational Science*. SIAM, 2019.

[2] J. R. Hauser, "Handling floating-point exceptions in numeric programs," *ACM Trans. Program. Lang. Syst.*, vol. 18, no. 2, pp. 139–174, 1996.

[3] J. Demmel and X. S. Li, "Faster numerical algorithms via exception handling," *IEEE Trans. Computers*, vol. 43, no. 8, pp. 983–992, 1994.

[4] "IEEE Standard for Floating-Point Arithmetic," *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pp. 1–84, July 2019.

[5] J. Demmel, J. J. Dongarra, M. Gates, G. Henry, J. Langou, X. S. Li, P. Luszczek, W. S. Pereira, E. J. Riedy, and C. Rubio-González, "Proposed consistent exception handling for the BLAS and LAPACK," in *Correctness@SC*. IEEE, 2022, pp. 1–9.

[6] E. T. Barr, T. Vo, V. Le, and Z. Su, "Automatic detection of floating-point exceptions," in *POPL*. ACM, 2013, pp. 549–560.

[7] X. Wu, L. Li, and J. Zhang, "Symbolic execution with value-range analysis for floating-point exception detection," in *APSEC*. IEEE Computer Society, 2017, pp. 1–10.

[8] X. Yang, G. Zhang, Z. Shuai, Z. Chen, and J. Wang, "Symbolic execution of floating-point programs: How far are we?" *J. Syst. Softw.*, vol. 220, p. 112242, 2025.

[9] D. Ma, Z. Liang, L. Yin, and H. Liang, "Symbolic testing of floating-point bugs and exceptions," *J. Syst. Softw.*, vol. 219, p. 112226, 2025.

[10] A. Tran, I. Laguna, and G. Gopalakrishnan, "Fpboxer: Efficient input-generation for targeting floating-point exceptions in GPU programs," in *HPDC*. ACM, 2024, pp. 83–93.

[11] I. Laguna and G. Gopalakrishnan, "Finding inputs that trigger floating-point exceptions in gpus via bayesian optimization," in *SC*. IEEE, 2022, pp. 33:1–33:14.

[12] C. Ma, L. Chen, X. Yi, G. Fan, and J. Wang, "Numfuzz: A floating-point format aware fuzzer for numerical programs," in *APSEC*. IEEE, 2022, pp. 338–347.

[13] L. S. Blackford, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry *et al.*, "An updated set of basic linear algebra subprograms (blas)," *ACM Transactions on Mathematical Software*, vol. 28, no. 2, pp. 135–151, 2002.

[14] F. G. V. Zee and R. A. van de Geijn, "BLIS: A framework for rapidly instantiating BLAS functionality," *ACM Trans. Math. Softw.*, vol. 41, no. 3, pp. 14:1–14:33, 2015.

[15] "OpenBLAS: An optimized BLAS library," http://www.openmathlib.org/OpenBLAS/, [Accessed 11-12-2024].

[16] A. D. Franco, H. Guo, and C. Rubio-González, "A comprehensive study of real-world numerical bug characteristics," in *ASE*. IEEE Computer Society, 2017, pp. 509–519.

[17] X. Li, I. Laguna, B. Fang, K. Swirydowicz, A. Li, and G. Gopalakrishnan, "Design and evaluation of GPU-FPX: A low-overhead tool for floating-point exception detection in NVIDIA gpus," in *HPDC*. ACM, 2023, pp. 59–71.

[18] I. Laguna, X. Li, and G. Gopalakrishnan, "Binfpe: accurate floating-point exception detection for GPU applications," in *SOAP@PLDI*. ACM, 2022, pp. 1–8.

[19] H. Guo and C. Rubio-González, "Efficient generation of error-inducing floating-point inputs via symbolic execution," in *ICSE*. ACM, 2020, pp. 1261–1272.

[20] C. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi, and K. M. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *PLDI*. ACM, 2005, pp. 190–200.

[21] H. Barbosa, C. W. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar, "cvc5: A versatile and industrial-strength SMT solver," in *TACAS (1)*, ser. Lecture Notes in Computer Science, vol. 13243. Springer, 2022, pp. 415–442.

[22] A. Abdelfattah, W. Ahrens, H. Anzt, C. Armstrong, B. Brock, A. Buluc, F. Busato, T. Cojean, T. Davis, J. Demmel *et al.*, "Interface for sparse linear algebra operations," *arXiv preprint arXiv:2411.13259*, 2024.

[23] J. Vanover, X. Deng, and C. Rubio-González, "Discovering discrepancies in numerical libraries," in *ISSTA*. ACM, 2020, pp. 488–501.

[24] P. A. Dinda, A. Bernat, and C. Hetland, "Spying on the floating point behavior of existing, unmodified scientific applications," in *HPDC*. ACM, 2020, pp. 5–16.

[25] I. Laguna, T. Tirpankar, X. Li, and G. Gopalakrishnan, "Fpchecker: Floating-point exception detection tool and benchmark for parallel and distributed HPC," in *IISWC*. IEEE, 2022, pp. 39–50.

[26] T. Allred, X. Li, A. Wiersdorf, B. Greenman, and G. Gopalakrishnan, "Flowfpx: Nimble tools for debugging floating-point exceptions," *CoRR*, vol. abs/2403.15632, 2024.