Floating-Point Symbolic Execution: A Case Study in N-Version Programming

Daniel Liew*, Daniel Schemmel†, Cristian Cadar*, Alastair F. Donaldson*, Rafael Zähl†‡, Klaus Wehrle†
* Imperial College London, United Kingdom, {daniel.liew,c.cadar,alastair.donaldson}@imperial.ac.uk
† RWTH Aachen University, Germany, {daniel.schemmel,klaus.wehrle}@comsys.rwth-aachen.de
‡ In Memoriam

***Abstract*—Symbolic execution is a well-known program analysis technique for testing software, which makes intensive use of constraint solvers. Recent support for floating-point constraint solving has made it feasible to support floating-point reasoning in symbolic execution tools. In this paper, we present the experience of two research teams that independently added floating-point support to KLEE, a popular symbolic execution engine. Since the two teams independently developed their extensions, this created the rare opportunity to conduct a rigorous comparison between the two implementations, essentially a modern case study on N-version programming. As part of our comparison, we report on the different design and implementation decisions taken by each team, and show their impact on a rigorously assembled and tested set of benchmarks, itself a contribution of the paper.**

I. Introduction

Symbolic execution has become a popular program analysis technique that can be used for test case generation and bug detection in a wide variety of domains [18], [19], [33], [34], [70], [75]. Underpinning any symbolic execution tool is a constraint solver, often a *satisfiability modulo theories* (SMT) solver, which does the heavy lifting associated with determining whether execution paths are feasible at runtime, and whether there exist values of the symbolic inputs that cause correctness conditions to fail.

Due to the challenges associated with constraint solving for floating-point arithmetic, most symbolic execution tools do not directly support symbolic floating-point reasoning, instead either using approximations [7], using structural equivalence of expressions as a proxy for equality [24], or rejecting programs that use floating point as out of scope [18].

A widely-used SMT-based symbolic execution tool is KLEE [18], which reasons about symbolic constraints with bit-level accuracy, and supports the entire C language, with a few exceptions, the most notable of which is symbolic floating-point computation. The original reason for the lack of floating-point support was the absence of a suitable solver. In lieu of this, KLEE handles floating-point programs by concretizing symbolic floating-point expressions, essentially reasoning about a single set of floating-point values on each explored path.

However, recent advances in solver technology have led to several SMT solvers adding support for floating-point reasoning along with an effort to provide a standardized theory of floating-point arithmetic [68]. Thus, it is a natural idea to add floating-point support to KLEE. Coincidentally, we—the two research groups who co-authored this paper—undertook such an extension of KLEE independently and at roughly the same time. When we became aware of each other's activities via communication on the KLEE mailing list, we set up a meeting to understand the status and maturity of each implementation, aiming to avoid duplication of effort. It became clear that we were too late: both teams had already invested significant effort and created mostly complete implementations.

This coincidence gave us a rare and valuable opportunity to empirically compare, in a very direct manner, two distinct and independent implementations of the same functional specification in the same framework, via a case study in N-version programming [5], [21]. We describe in detail our methodology for independently developing floating-point benchmarks, without knowledge of each other's implementations; exchanging these benchmarks and using the combined benchmark suite to independently improve our respective tools in isolation; and finally exchanging tools and conducting a detailed head-to-head comparison with respect to the benchmark suite.

Key contributions. Our major contribution is an experience report on N-version programming (with N=2) in the context of floating-point symbolic execution. This contributes **(1)** a rigorous experimental methodology for controlled N-version programming that can be followed or adapted for future studies; **(2)** two complementary open-source extensions to KLEE that support floating-point symbolic execution [51], [69]; and **(3)** a discussion of lessons learned from this experience.

Summary of lessons learned, and supporting contributions. *Independent preparation of benchmarks pays dividends*. In a domain with such subtle semantic issues as floating-point reasoning, having each team independently prepare a set of benchmarks was useful in providing both a practical specification for floating-point symbolic execution, and a target for tool optimization. The benchmarks from each team provide a relatively unbiased target for evaluating the other team's tool. The benchmark creation process has also led to a supporting contribution: **(4)** an open-source set of floating-point benchmarks tailored for symbolic execution tools [28]–[30].

Dual implementation leads to rich design space exploration. While our tools feature several similar design decisions, their independent development has led to notably different solutions to various floating-point-related issues, e.g. in how the tools support the `long double` data type. Having two complementary tools enables differential testing (cross-checking results

Accepted for publication in Proc. of ASE 2017.

between tools) [56] to find tool bugs; portfolio analysis, where both tools are applied to a program in parallel; and puts us in a good position in the future to combine the strengths of each to yield a high-quality implementation of floating-point support in KLEE.

Floating-point constraint solving remains a challenge. Although our tools complement each other, neither offers a silver bullet for floating-point symbolic execution: surmounting the inherent difficulty of reasoning about floating-point constraints will require advances in constraint solving. To this end, as a supporting contribution: **(5)** we have extracted a set of 35,189 floating-point SMT queries generated by our tools, which have been accepted [3] as benchmarks for the annual SMT competition [72], providing a rich source of challenging examples for solver developers. In this set of benchmarks, 18,033 belong to a new logic added to competition that combines the array, bitvector and floating-point theories.

Rigorous N-version programming can be limiting. The rigor associated with the methodology that we have followed, to enable a controlled study, created a somewhat artificial development environment in which the teams did not ask one another for advice on technical matters despite being ideally placed to do so. We believe our methodology is mainly suitable for future rigorous studies in N-version programming, to gain insights into the software engineering process, rather than for regular software development projects. However, for regular development projects we strongly recommend the independent preparation of benchmark suites by multiple teams.

Structure. After reviewing essential background (§II), we describe the methodology for our experimental comparison (§III). We then detail the benchmarks we developed (§IV), remark on notable features of the independent designs of the two floating-point extensions to KLEE (§V), and compare the tools experimentally (§VI). We finally discuss related work in this area (§VII), and conclude with by recapitulating the main conclusions and lessons learned (§VIII).

II. Background

We provide relevant background on symbolic execution (§II-A) and floating-point arithmetic (§II-B).

A. Symbolic Execution

Symbolic execution is a program analysis technique that provides the ability to automatically explore multiple paths through a program. It has been implemented in many different tools [17], [18], [33], [61], [70] and applied to many different areas, such as software engineering, systems and security [20].

Instead of running the program on concrete input, where a particular input component might e.g. take the value 3, symbolic execution runs the program on *symbolic* input, where each input component is represented by a placeholder for an initially unconstrained value, e.g. `x`. As the program runs, symbolic execution keeps track of how program variables depend on the symbolic input. For instance, after executing the statement `y = x + 1`, symbolic execution will remember that variable `y` holds the symbolic value $\chi+1$, where χ is whatever symbolic value is being held by `x` at the point this instruction is executed. If symbolic execution reaches a branch that depends directly or indirectly on the symbolic input, it first uses a constraint solver to check the feasibility of each branch side. If only one side is feasible, it will follow only that side. If *both* sides are feasible, it forks execution and follows each side separately, adding appropriate constraints on each path. For instance, if in our example the branch `if(y > 0)` is reached, symbolic execution will add the constraint that $\chi+1 > 0$ on the true side and $\neg\,(\chi + 1 > 0)$ on the false side. When a path terminates, the conjunction of all the constraints collected at branch points (called the *path condition*) is sent to a constraint solver, which can provide a concrete solution. This solution represents a test input that follows the same path as the one on which the path condition was collected. Using this mechanism, symbolic execution can systematically explore paths through a program.

TABLE I
X86_64 FLOATING-POINT TYPES.

Name	Size	p	e_{max}	Leading bit m explicit?
fp32	32 bits	24	127	No
fp64	64 bits	53	1023	No
x86_fp80	80 bits	64	16383	Yes

B. Floating-point Arithmetic

In general, floating-point number representations provide finite approximations to the real numbers, trading range and precision for storage space. They utilize scientific notation of the form $(-1)^s \times m \times b^e$ where s is either 1 or 0 and controls the sign, m is a real number called the significand (typically in the range $[0, b)$), b is an integer base (typically 10 or 2), and e is an integer exponent. For example -0.75 can be written as -1.5×2^{-1}. By using a fixed number of digits for exponent and significand, floating-point number representations restrict the range and precision, respectively, of representable numbers.

In this work we are interested in analysis of C programs that operate on floating-point data. The IEEE-754 2008 standard [41] (also known as IEC 60559:2011 [43]) formalizes a number of floating-point representations. If an implementation of C respects Annex F of the C language specification [45], then most of its floating-point types and operations are IEEE-754 compliant (some exceptions to this are detailed below).

IEEE floating point on x86_64. We provide details of IEEE floating point representation implemented by the x86_64 family of processors; both tools assume this target.

Three primitive floating-point types are available on this target: 32-bit wide *single precision* (IEEE-754 binary32), 64-bit wide *double precision* (IEEE-754 binary64), and 80-bit wide *double extended precision* (not an IEEE-754 basic format). We refer to these types as fp32, fp64, and x86_fp80 respectively.

Each type has a precision p (number of bits representing the significand), and maximum exponent value e_{max}. A full list can be found in Table I. The last column ("Leading bit m explicit?") states whether the binary encoding of the type contains the leading bit (i.e. the integer portion of) m, or if it is inferred from the remaining bits.

In an Annex F-compliant implementation of C on x86_64, fp32 and fp64 are the `float` and `double` types respectively. The C standard with Annex F only weakly specifies how the `long double` type should be implemented. All C implementations that target x86_64 that we are aware of treat `long double` as x86_fp80.

The IEEE-754 binary format contains several classes of data: normal, denormal, zero, infinity and NaN. Most floating-point numbers belong to the *normal* class, which provides a unique encoding for representable numbers where the leading bit of the significand is always `1` and the exponent is in the range $[-e_{max}+1, e_{max}]$. The *denormal* class represents numbers close to zero and exists to provide a smoother transition from the smallest positive *normal* (largest negative *normal*) number to positive (negative) zero. *Denormal* numbers always have the exponent and leading bit of the significand set to $-e_{max}$ and `0` respectively. The *zero* and *infinity* classes each contain two values: positive and negative zero and infinity, respectively.

The *NaN* class represents "not a number". NaN values arise from invalid computations, such as `0.0/0.0`. There are many different binary encodings for NaN but IEEE-754 only distinguishes between two types: quiet and signaling.

The x86_fp80 type is not an IEEE-754 binary format. It consists of 1-bit sign, 15-bit exponent and 64-bit significand. The binary encoding is similar to that of the IEEE-754 binary format except that the integer portion of the significand is stored explicitly. This additional bit permits extra classes known as *pseudo-NaN*, *pseudo-infinity*, *unnormal* and *pseudo-denormal*. Modern Intel processors consider all these classes apart from pseudo-denormals as invalid operands [42].

Rounding modes and exceptions. IEEE-754 provides several different rounding modes (e.g. round toward positive), a subset of which can be used from within the C language. IEEE-754 also defines several different exceptions (e.g. division by zero) that can be raised when operations are performed. The default handling of these is to set one or more status flags and then continue execution.

Reasoning about floating point. Given the complexity of floating-point arithmetic—with the background information above covering only a small fraction of the standard—reasoning about floating-point code is difficult and error-prone. Common programming errors are often caused by incorrectly assuming that the arithmetic laws for real numbers hold—unfortunately, certain basic laws such as associativity and distributivity do not hold for floating point, resulting in subtle bugs that may be triggered by a subset of input values.

III. Methodology

On first point of contact, both teams had relatively feature-complete floating-point extensions to KLEE that had undergone preliminary correctness testing and performance benchmarking. We structured our controlled N-version programming experiment around three phases: benchmark preparation (§III-A), benchmark and tool improvement (§III-B), and in-depth comparison (§III-C).

A. Phase I: Benchmark Preparation

During a period of approximately one month, each team devoted resources to *independently* preparing benchmark programs to be used for evaluation. Each team prepared 43 benchmarks, divided into 28 synthetic and 15 "real-world" benchmarks. The real-world benchmarks were adapted from existing open-source applications. The synthetic benchmarks were written from scratch, with some designed to test particular aspects of floating-point semantics, and others encoding simple algorithms. The benchmarks are described in §IV.

Both sets of benchmarks were prepared with symbolic analysis in mind: the teams ensured that most benchmarks had at least some inputs marked as symbolic, though a few fully concrete benchmarks were included to thoroughly test concrete interpretation. Due to the known limited scalability of solvers with respect to floating-point reasoning, symbolic data was restricted to aim for a sweet spot where symbolic execution would issue interesting, yet not intractably hard, floating-point queries. Importantly, this fine-tuning was performed by each team in isolation with respect to their own tool.

Each benchmark includes a specification stating how the benchmark should be compiled and whether the benchmark is expected to be correct. An incorrect benchmark's specification states a number of expected property violations, e.g. that an assert should fail, or a division by zero or invalid memory dereference should occur. In each case a set of allowed error locations (source file and line number) are provided.

During this phase, the teams were free to improve their tool with respect to their own benchmarks. At the end of phase I, all benchmarks were pushed to a common repository.

Phase I resulted in a set of floating-point benchmarks suitable for evaluation of symbolic execution tools, with one half known to be somewhat tractable for Aachen's tool, and the other half for Imperial's tool, but importantly with no single benchmark having been prepared knowing the capabilities of both tools.

B. Phase II: Benchmark and Tool Improvement

The full set of benchmarks allowed each team to assess the correctness and performance of their independently-developed tool, through semantic problems and optimization opportunities raised by the other team's benchmarks. Each team spent approximately one month fixing and optimizing their tools. Notable tool changes arising from benchmark exchange are discussed in §V. During this phase the teams communicated any benchmark problems not already identified during phase I, but did not exchange tool implementation details. These benchmark problems are discussed in §VI-A.

At the end of phase II, the teams froze development of their tools and exchanged source code, enabling each team to subsequently (a) understand the design decisions of the other team via source code inspection, and (b) compare experimentally with the other team's tool.

C. Phase III: In-depth Comparison

The teams now set about comparing the tools on the finalized benchmarks. Since both teams' tools leveraged Z3

and LLVM, it was agreed that both should share the same Z3 version (`4c664f1c`) and LLVM version (3.4.2) so as to restrict behavioral differences to design decisions in the KLEE extensions themselves, rather than in their dependencies.

Tool changes based on preliminary experiments. Our intent had been to conduct our in-depth comparison using exactly the tool versions frozen at the end of phase II. However, preliminary experiments revealed a number of remaining tool bugs that were easy to fix, and whose fixes were not influenced by implementation details of the opposite team's tool. We also realized that the tools were forked from different versions of KLEE, leading to potential behavioral differences unrelated to the innovations of each team. Finally, a *dynamic solver timeout* feature (§V-B), implemented by Imperial and orthogonal to floating-point support, allowed KLEE to terminate in a more reliable manner that influenced tool comparison. Based on this experience, we upgraded both tools by rebasing to use a common KLEE revision (`2852ef63`), donating the dynamic solver timeout feature to Aachen, and applying a number of bug-fixes following an inter-team review to confirm that fixes were not inspired by details of the opposite tool.

Running the tools. We ran the finalized versions of both teams' tools on the finalized benchmark suite, on a machine with two Intel® Xeon® E5-2643 v4 CPUs (6 physical cores each) with 256GiB of RAM running Arch Linux. Hyper-threading and turbo boost were disabled. Each tool was run 20 times per benchmark. Each tool was executed in parallel over the set of benchmarks, running on at most 8 benchmarks in parallel. Each KLEE process was pinned to a single CPU core and the CPU's nearest NUMA node. The CPU cores used for pinning were isolated from the kernel scheduler using the `isolcpus` kernel parameter. The pstate CPU governor was set to "performance" requesting the same min/max frequency (3.5GHz) and a `0` performance bias. We enforced a 100 GiB memory limit per KLEE process. Each tool was executed in a Docker [58] container to keep the processes isolated.

KLEE has distinct phases for path exploration and subsequent test case generation. We enforce a time limit of 1 hour for each phase. Each team's tool was configured to use the same path exploration strategy (non-uniform random search prioritizing coverage, with a fixed random seed).

Comparing the tools. In order to compare the tools we extracted the following information from each run.

The validity of a reported bug, i.e. whether it is a true or false positive. A test case that detects a particular issue at a certain benchmark source location is considered to detect a true positive if and only if the benchmark's specification indicates that an issue of this type is expected at that location. Checking validity was achieved by replaying KLEE-generated cases natively for reported bugs and verifying that the bug type, source file and line number match what KLEE reported. To check for out-of-bounds memory accesses and division by zero—undefined behaviors in C that are not guaranteed to raise a runtime exception—we instrumented the benchmarks using ASan [71] and UBSan [76] respectively.

Branch coverage achieved on each benchmark. This was measured by replaying all KLEE-generated tests natively on a coverage-instrumented (via gcov [32]) build of the benchmarks. Coverage excludes the C library to avoid bias; e.g. a team's tool might interpret a C library function (may lead to additional test cases) rather than modeling it in Z3 (no additional test cases), possibly leading to greater coverage of the C library which upon replay could inflate the team's coverage artificially.

Execution time and crashes. We recorded execution time for each run of a tool on a benchmark, and noted cases where a tool crashed due to an internal error or running out of memory. Memory and time limits of 10 GiB and 5 minutes were used when replaying test cases.

We merged the repeated runs of the same tool configuration as follows. The true and false positives for a tool with respect to a benchmark were identified by replaying the test cases generated across *all* runs, crediting a tool for finding a true positive during *any* run, but similarly penalizing for any instances of false positives. We computed branch coverage and execution time for a tool with respect to a benchmark as the arithmetic mean across all runs. We counted the number of crashes for a tool with respect to a benchmark as the total number of crashes observed across all runs.

We then ranked the tools on a per-benchmark basis using the following rules, applied in order:

(A) A tool wins if it reports no **false positives** and the other tool reports at least one **false positive**.
(B) Most **true positives** wins.
(C) Highest mean **branch coverage** wins.
(D) If at least one tool **crashed**, smallest crash count wins.
(E) Otherwise, smallest mean **total execution time** wins.

The tools draw if they are not distinguished by these rules. The rationale for ranking is: a symbolic execution tool should never exhibit false positives (A), because its primary goal is to accurately find bugs (B), with a secondary goal of producing a high-coverage test suite (C). Thereafter, we prefer a tool that does not crash (D), and a fast tool when neither crashes (E). It is hard to meaningfully compare false positives (two distinct false positives might not be equally serious), so in (A) we do not rank tools by number or nature of false positives.

When comparing mean branch coverage and execution time we use 95% and 99.9% confidence intervals, respectively, regarding results as indistinguishable if intervals overlap. Mean execution time differences of less than one second are also treated as indistinguishable.

IV. Benchmark Suite

As mentioned in §III-A both teams independently contributed 43 benchmarks (86 in total), written in C99 [44] or C11 [45]. Each team aimed to choose examples that would be challenging yet not intractable for symbolic execution, being free to choose benchmarks that played to the strengths of their tool, with benchmarks prepared by the other team posing an unknown challenge. The suite contains 52 benchmarks expected to be correct and 34 expected to be incorrect. The suite uses KLEE-specific functions (e.g. to introduce symbolic data) for our

convenience, however it would be easy to port the benchmarks to a different interface (e.g. SV-COMP's [1]), so that they are applicable to other analysis tools. Branch counts reported below are the number of static branches in the compiled LLVM IR.

A. Aachen's Benchmarks

Synthetic (17 correct, 11 incorrect): These focus on checking a wide range of floating-point semantic features, split up so that each benchmark tests as few individual features as possible, allowing floating-point symbolic execution errors to be easily pinned to underlying causes. Some check properties that are uncommon in real-world programs, to ensure that unusual and erroneous uses of floating-point numbers are handled accurately. These benchmarks have between 1 and 56 (median 3) branches and request between 0 and 32 (median 8) symbolic bytes.

Real world (13 correct, 2 incorrect): These were drawn from multiple publicly available sources, with care taken to include both large and well-tested software, reflected by benchmarks built upon the GNU Multiple Precision Arithmetic Library [37], and sample programs not intended for production use, e.g. numerical code taken from [64]. These benchmarks have between 6 and 2903 (median 220) branches and request between 1 and 20 (median 8) symbolic bytes.

B. Imperial's Benchmarks

Synthetic (15 correct, 13 incorrect): These comprise of simple algorithms (e.g. binary search), and tests for fundamental properties of floating point (e.g. commutativity and non-associativity of addition). We include a port of William Kahan's *paranoia* benchmark [46], [47]. These benchmarks have between 1 and 301 (median 8) branches and request between 0 and 128 (median 8) symbolic bytes.

Real world (7 correct, 8 incorrect): These were written against the GNU Scientific Library (libGSL) [38], adapted from tutorial examples included with the library source code. Creating these benchmarks involved iteratively increasing the presence of symbolic input, stopping just before the tipping point beyond which Imperial's tool could not make reasonable progress. Some of the libGSL benchmarks used long doubles, a feature that Imperial's early tool did not support; we modified the associated benchmarks to avoid long doubles. These benchmarks have between 6 and 254 (median 67) branches and request between 4 and 48 (median 16) symbolic bytes.

V. Design Details

At the end of phase II (§III-B) we exchanged tool source code, allowing comparison of tool designs that had previously been unknown across teams. We discuss noteworthy similarities and differences between the designs of both tools, as determined by discussion and source-code examination. Both tools build on top of KLEE so they share mostly the same architecture, which is discussed in the original KLEE paper [18].

Exchanging benchmarks at the end of phase I clearly led to improvements in the design of both tools. We consider this a positive outcome, illustrating how a shared set of independently-gathered benchmarks can drive tool development.

A. Notable Similar Design Decisions

Constraint solver. Both teams used Z3 [27] as the constraint solver; this was a natural choice as Z3 supports floating point and was already integrated into open-source KLEE.

Floating-point types, operations and functions. KLEE is primarily designed to execute programs written in C, but actually executes LLVM intermediate representation (IR). Both teams assumed the x86_64 target and thus that the `float`, `double`, and `x86_fp80` LLVM types map to the IEEE-754 fp32, fp64, and x86_fp80 types. Assumption of IEEE-754 semantics was key, as they are used by the SMT-LIB floating-point theory [15] that Z3 implements.

Both teams assumed that the primitive arithmetic LLVM IR instructions (e.g. `fadd`) map to corresponding IEEE-754 operations, except for `frem`, whose semantics are not the same as the IEEE-754 remainder function [57]. Both teams assumed that operations on LLVM types consistently use the same precision and range, so that excess precision/range are not used during computation. This assumption holds if during native code generation the compiler uses SSE instructions (rather than the legacy x87 FPU) to do floating-point computations on fp32 and fp64 types [59].

Vector instruction support. LLVM IR provides vector types derived from the basic floating-point types; enabling compiler optimizations can cause vector instructions to be emitted. To process these, both teams adapted LLVM's `Scalarizer` pass to scalarize as much as possible prior to symbolic execution, so that KLEE can assume that most instructions (e.g. `fadd`) only take scalar operands. A few instructions—`InsertElement`, `ExtractElement`, and `ShuffleVector`—required special handling; both teams added varying levels of support, sufficient to run the benchmarks (§IV).

It is worth noting that at the end of phase I, Aachen and Imperial's support for vector instructions differed greatly. Imperial compiled their benchmarks with optimizations enabled, necessitating vector support; Aachen compiled their benchmarks without optimizations, thus did not need vector support. During phase II it became necessary for Aachen to add vector support in order to handle Imperial's benchmarks.

IEEE-754 rounding modes. Both teams implemented support for all IEEE-754 rounding modes available from the C standard library interface, by having a per-execution state flag that stores the current concrete rounding mode and ensuring that this is used when constructing constraints (e.g. floating-point addition is affected by the rounding mode, making it a ternary operator). KLEE has the ability to call *external functions*—functions missing from the program under analysis that can nevertheless be executed by the running KLEE process on behalf of the program under analysis. Both teams ensured that when calling external functions the rounding mode of the KLEE process is changed to that of the calling execution state and reverted back on return. One slight difference in Imperial's implementation is that the rounding mode is allowed to be symbolic, whereas Aachen's implementation concretizes a symbolic mode. Imperial's implementation achieves this by

forking on symbolic rounding modes (i.e. one path per rounding mode plus an extra path for an invalid rounding mode).

Standard math functions. The teams initially used different approaches for handling C standard library math functions. Aachen represented these functions as operations in KLEE's constraint language in several cases (e.g. `fabs`, `sqrt`), while Imperial simply interpreted the implementation of each function in KLEE's C library. Imperial's approach suffers from path explosion when operands to the functions are symbolic. Upon exchanging benchmarks at the end of phase I, Imperial discovered their tool performed poorly on several of Aachen's benchmarks due to path explosion and so switched to handling `fabs` and `sqrt` as operations in KLEE's expression language.

IEEE-754 exceptions and flags. Neither team implemented this portion of the IEEE-754 specification because it would significantly complicate symbolic execution: one would need to maintain the symbolic value of the flags and check if any exceptions could be raised by every floating-point instruction executed. A consequence of this (and of the SMT-LIB floating-point theory) is that symbolically neither tool can distinguish between quiet and signaling NaNs.

B. Notable Differences

Extending KLEE's expression language. Both teams extended KLEE's expression language to incorporate floating-point expressions in similar ways, but with some notable differences related to how comparison operations and constants are handled. Aachen added operations that correspond directly to the opcodes of the LLVM `FCmp` instruction. The instruction has *ordered* and *unordered* variants, which return false and true, respectively, if either argument is a NaN. Instead, Imperial only added operations that correspond to the *ordered* comparison opcodes because the *unordered* operations can be expressed in terms of the *ordered* comparison operations and the `IsNan` predicate. We consider Imperial's approach to be a better choice because it is a simpler extension to KLEE's expression language. Aachen represented floating-point constants as a separate expression type whereas Imperial represented floating-point constants as implicitly bitcasted integer constants.

Representation of types. KLEE's expression language is based solely on bitvectors, which was problematic when introducing floating-point operations. Imperial handled this by making conversion between floating-point and bitvector types implicit, so that e.g. the bitvector operands to a floating-point `add` instruction are first converted to floating-point types. Aachen instead made this explicit by introducing type conversion operations. We consider Aachen's choice to be superior here because implicit conversion has ambiguities. In particular if an *if-then-else* expression has a mixture of bitvector and floating-point types for its *then* and *else* expressions, the type of the *if-then-else* is ambiguous.

Array ackermannization. KLEE represents all symbolic data, including primitive symbolic variables, as arrays of 8-bit bitvectors. Imperial observed that in a floating-point context, if all arrays of bitvectors are replaced with simple bitvector variables and given to Z3 in such a way that the new query is equisatisfiable with the original query, then performance usually improves. We refer to this transformation as *array ackermannization*. The Z3 developers confirmed this, suggesting that Z3 is not currently well-optimized for queries mixing the theory of quantifier-free bitvector arrays with the theory of quantifier-free floating point [79].

Imperial's tool performs array ackermannization in the case where all array reads are at concrete indices and no writes have been performed to the array. This is a fairly common case because symbolic variables in the original C program being analyzed are typically represented as a concatenation of byte reads at successive concrete indices of a symbolic array.

x86_fp80 support. At the end of phase I, only Aachen added support for x86_fp80 and benchmarks to exercise it. Thus during phase II it became necessary for Imperial to implement support for symbolic reasoning over this type too.

Our designs differed due to several characteristics of the x86_fp80 type. First, it is a padded type: its 80 bits are padded to 128 bits on x86_64, and KLEE does not handle such padding properly. Both teams handled this issue in the same way, making sure KLEE allocates the necessary padding during the allocation of x86_fp80 stack and global variables.

Second, because x86_fp80 is not an IEEE-754 binary type, constant folding expressions of this type required careful consideration, and expressions of this type could not be directly modeled in the SMT-LIB floating-point theory.

KLEE already had support for constant folding the x86_fp80 via LLVM's `APFloat`, which performs arbitrary precision floating-pointing arithmetic in a hardware independent manner. However, one of Aachen's benchmarks caused both teams to discover that `APFloat` has a bug [54] where *unnormal* operands are not handled correctly. Imperial solved this issue by evaluating all x86_fp80 operations natively within KLEE itself. Aachen solved this by storing a flag in every expression which is set to true iff the expression represents a member of one of the IEEE-754 classes. When the value is accessed through a x86_fp80 operation, this flag is examined and if it is false for any operands a NaN is returned, which mirrors how unnormal operands are treated on x86_64. A fix for the bug in `APFloat` would avoid the need for these workarounds.

To handle evaluating symbolic expressions over x86_fp80 with Z3, both teams used the `(_ FloatingPoint 15 64)` type (abbrv. fp15_64) which has a 15-bit exponent, a 64-bit significand, and an implicit integer significand bit. It has exactly the same range and precision as x86_fp80, but a different 79-bit binary encoding due to the SMT-LIB floating-point theory only being able to represent IEEE-754 classes. The different binary encoding requires special handling of conversions between a bitvector that represents x86_fp80 data and fp15_64.

Imperial chose to only allow the IEEE-754 classes of the x86_fp80 type. When converting to fp15_64 from a bitvector the explicit leading significand bit is removed and an additional constraint is added that asserts that the bit has the correct value for the bitvector to represent an IEEE-754 value. When converting a fp15_64 to a bitvector, the explicit leading

significand bit is added back and its value is inferred from the other bits to be an IEEE-754 value.

Aachen chose to allow the non IEEE-754 classes of the x86_fp80 type in addition to the IEEE-754 classes. This was achieved by representing expressions of the x86_fp80 type as a tuple. The first value in the tuple is of type fp15_64. The second value is a boolean flag that is true iff the value represented by the tuple belongs to one of the IEEE-754 classes. These tuples are then handled by each floating-point operation. If one of the tuple operands does not represent a value from one of the IEEE-754 classes it returns the tuple (NaN, `true`).

Imperial's implementation results in simpler constraints being given to Z3 but is incomplete. Aachen's implementation is complete but the constraints are more complicated and also contradict the goals of array ackermannization because the tuple is represented as a two element array.

Dynamic solver timeout. Although KLEE can limit the time allowed for path exploration before switching to test case generation, KLEE does not try to interrupt the solver if the path exploration timeout is reached. For long-running solver queries—especially frequent when using floating-point constraints—this can cause the path exploration timeout to fire much later than intended. This leaves less time for test case generation (see §III-C), and may cause test cases to be lost.

KLEE supports setting a fixed (i.e. static) solver timeout, but this does not interact well with the path exploration timeout. A small solver timeout may leave paths unexplored despite there being available time for further path exploration, while too large a timeout may cause the path exploration timeout to be missed as discussed above.

Imperial implemented a *dynamic* approach to solve this problem. Every time the solver is invoked, the per-query solver timeout is set based on KLEE's current state. If KLEE is doing path exploration, the solver timeout is set to be the remaining path exploration time. When KLEE switches to test case generation, the solver timeout is set to the total allowed time for test case generation divided by the number of test cases to generate. This means that each test case is given an equal share of solver time.

While Aachen did not originally implement such a feature, as noted in §III-C, to ensure compatibility this feature was donated to Aachen's implementation.

NaN representation. Neither tool can distinguish between quiet and signaling NaNs. IEEE-754 does not precisely specify the binary encoding for either of these NaNs, which means there are many different encodings that represent the same type of NaN. Therefore, when converting a floating-point expression to a bitvector expression, if it is feasible for the expression to be NaN, it means the converted bitvector expression can take on many values that all represent NaN. In practice this is rarely a problem, however one of Imperial's synthetic benchmarks deliberately tries to branch on the value of the lower-order bits of a NaN, whose value is not defined by IEEE-754.

During phase II, Imperial discovered that this would sometimes crash their implementation due to inconsistent constraints. This was partly due to some bugs in Z3 [80] and LLVM's APFloat [52], which Imperial worked around by consistently using the same bitvector constant to represent NaN. Aachen did not handle this issue. However the under-specified nature of IEEE-754 NaN bit patterns are a general problem. Even if KLEE's expression language was modified to have identical semantics to Z3, they may still differ from those of the target machine or other SMT solvers.

VI. Experimental Comparison

We now turn to the comparison of the tools. We discuss problems with the benchmarks flagged by tool comparison (§VI-A), and results comparing the finalized tool versions head-to-head during phase III (§VI-B).

A. Benchmark Issues

Non-termination. When replaying tests generated by the tools we found that two benchmarks did not terminate for certain inputs. In one case this was unintentional, due to a bug in the implementation of a binary sort algorithm used by the benchmark. We handled these cases by setting a timeout when replaying test cases (see §III-C). Due to gcov implementation details [32], branch coverage is not recorded for non-terminating tests. This affected only a few benchmarks and gave neither tool an advantage in our ranking scheme.

Unnormal values. We found several problems with a benchmark that operated on the x86_fp80 type, involving *unnormal* values (see §II-B). First, Clang would incorrectly optimize the benchmark by performing erroneous constant folding [54]. We thus disabled optimizations for this benchmark. We also discovered it is possible to exploit this problem to crash Clang [53]. Second, we found that several operations on unnormal numbers used in this benchmark behaved *inconsistently* across compilers (e.g. `isnan()` and casting to integers). We concluded that this was due to these operations exhibiting undefined behavior, and removed them from the benchmark.

The remaining issues are cases where our tools found a benchmark to be incorrect, contrary to its specification; for each issue we applied a simple fix:

Failing to account for NaNs. A benchmark that sorted an array of partially symbolic floating-point values was incorrect when infinity values were added to yield NaN values, later triggering an assertion failure when checking correctness of sorting. A benchmark performing matrix multiplication on a partially symbolic matrix was similarly incorrect.

Failing to account for scientific notation. A benchmark that verifies the output of `atof()`[1] intended to constrain the characters of the symbolic input string to represent a small decimal value, asserting that the result of `atof()` was in the expected range. The input constraints accidentally allowed scientific notation (e.g. `1e10`), so that `atof()` could generate a value outside the expected range.

Failing to take poor approximation into account. A benchmark that checks the result of $\sqrt{x^2}$, where x is a symbolic

[1] Converts a string to a floating-point value.

TABLE II
RANKING OF THE TOOLS. EACH COUNT SHOWS THE NUMBER OF WINS FOR A TOOL EXCEPT THE LAST ROW WHICH SHOWS THE NUMBER OF DRAWS.

Reason	Aachen	Imperial
Other tool has false positives	0	0
Finds more bugs	1	1
Highest branch coverage	1	4
Fewest crashes	0	3
Smallest execution time	5	22
Draws	49	

TABLE III
EVALUATION OF THE TOOLS IN TERMS OF BUG-FINDING, EXHAUSTIVE EXPLORATION, NUMBER OF CRASHES AND NUMBER OF TIMEOUTS. THE T^+ AND T^- ROWS SHOW THE NUMBER OF TRUE POSITIVES AND TRUE NEGATIVES RESPECTIVELY.

	Aachen	Imperial	Both	Neither
T^+	33 (67.35%)	33 (67.35%)	32 (65.31%)	15 (30.61%)
T^-	35 (67.31%)	38 (73.08%)	34 (65.38%)	13 (25.00%)
Crashes	9 (10.47%)	2 (2.33%)	2 (2.33%)	77 (89.53%)
Timeouts	21 (24.42%)	24 (27.91%)	21 (24.42%)	62 (72.09%)

floating-point value, did not account for denormal numbers. As the computation may cause a gradual underflow to a denormal number, its precision may be reduced to a single bit in the worst case, causing a very high relative error.

B. Head-to-head Tool Comparison

Tool ranking. Table II summarizes the number of benchmarks for which each team *won*, according to the procedure for ranking tools described in §III-C.

Neither tool reported false positives. For one benchmark Aachen found more bugs than Imperial and for another benchmark Imperial found more bugs than Aachen. Both benchmarks use the `sqrt()` function and come from Aachen's set of synthetic benchmarks and both tools exhibit poor performance on them due to long query solving times.

In cases where the tools found the same number of bugs, Imperial achieved higher branch coverage in four cases, and Aachen in one. Imperial achieved higher coverage when Aachen's tool crashed while trying to generate test cases, thus missing out on coverage that the generated tests would have achieved. In all cases, the crashes were either due to an internal Z3 error (1 case), or there being a semantic mismatch between the expression languages of KLEE and Z3, causing Z3-generated models to be unsatisfiable in KLEE's expression language (3 cases). For the single benchmark where Aachen's tool achieved higher coverage, Imperial's tool reached a path exploration timeout whereas Aachen's tool explored all paths in the benchmark.

For benchmarks where the tools were as-yet indistinguishable, there were three cases where Imperial's tool did better due to Aachen's tool crashing, 27 cases where neither tool crashed but where the tools were distinguished by execution time, with Aachen and Imperial winning 5 and 22 times, respectively.

Of the 49 draws, two are due to the tools both crashing the same number of times, and 47 are cases where neither tool crashes, but where either the confidence intervals associated with mean execution time overlap, or the difference in the mean execution time is less than one second. The crashes here are internal to Z3 which have been reported [78].

Aside from cases where the tools crashed, we attribute the differences in effectiveness of the tools to the performance of the constraint solver which is discussed later in this section.

Coverage. Figure 1 compares the branch coverage for Aachen's and Imperial's tools on a per-benchmark basis. For each benchmark, there are two bars showing the percentage of branches that have been covered, one striped yellow bar for Imperial and one solid green for Aachen. Benchmarks are ordered along the x-axis such that the ones where Imperial's (Aachen's) tool achieved a higher coverage are on the left (right). Benchmarks with identical coverage are sorted alphabetically. The figure shows that for most benchmarks both tools achieve the same coverage. While neither tool strictly dominates the other, it can be seen that Imperial's tool has a slight advantage in this area.

The results are mostly deterministic (have a 95% confidence interval of below 10^{-15}), except for the rightmost benchmark, which has a 95% confidence interval (not shown in the figure for clarity) of ± 11.25 percentage points for the Aachen tool only. We attribute this to Aachen's tool crashing non-deterministically on this benchmark.

Note that while 100% is the theoretical maximum branch coverage, it is not reachable in every case, especially for the real-world benchmarks, many of which use a library but only exercise a small portion of it.

Tool complementarity and limitations. The results so far show that the tools overlap somewhat in their capabilities. We now examine the extent to which the tools are capable of effective analysis of our benchmarks, whether they are hindered by common problems, and cases where they are complementary.

Table III shows the extent to which each tool is capable of correctly finding bugs in the benchmarks. The T^+ row shows, for each tool, the number of total bugs found, out of the number of bugs expected to be present from the benchmark specifications (in the 34 benchmarks with erroneous paths we expect to find a total of 49 bugs). The T^- row shows, for each tool, the number of bug-free benchmarks (52 total) that the tool is able to fully explore. That is, *all* feasible paths are enumerated so that correctness is exhaustively verified. In each row, tool complementarity is indicated by showing the extent to which bugs can be found, or correctness proven, by both tools or by neither tool.

Both tools found the same 32 bugs, with each tool finding one one additional bug that the other did not. This shows that both tools perform reasonably well at bug finding with neither dominating the other. There were 15 bugs that neither tool found indicating that the benchmark suite was challenging.

Both tools determined the same 34 benchmarks to be correct, with Aachen showing one addition benchmark to be correct that Imperial did not and Imperial showing four additional benchmarks to be correct that Aachen did not. This shows that

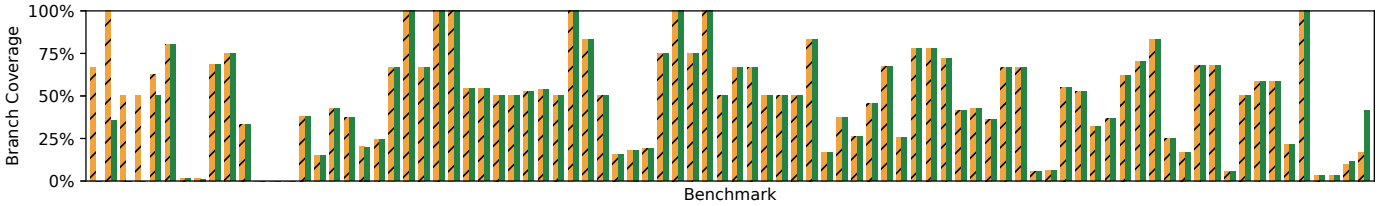

Fig. 1. Branch coverage achieved by the two tools. On the left, Imperial (yellow bars with black stripes) did better, while on the right, Aachen (solid green bars) achieved higher coverage. The long middle section shows benchmarks for which both teams reached the same coverage.

both tools perform reasonably well at exhaustive exploration with neither dominating the other. There were 13 benchmarks expected to be correct that neither tool could show to be so, indicating that benchmark suite was challenging enough that exploration was not exhaustive. Non-exhaustive exploration was due to timeouts, crashes, and memory exhaustion.

Table III also provides details of the number of benchmarks on which each tool crashed (incorporating cases of memory exhaustion) or timed out. Aachen's tool crashed more than Imperial's due to the semantic inconsistencies discussed in §VI-B. For two of the benchmarks both tools crashed due to hitting an internal Z3 error [78]. Imperial's tool timed out on three benchmarks where Aachen's did not. On those three benchmarks Aachen's tool fully verified one of them and crashed on the other two.

Solver performance. We attribute the performance difference in the tools to the performance of Z3, as constraint solving accounts for the majority of execution time for both tools. We observed that for some paths, the tools would send equisatisfiable but different constraints to Z3—e.g. due to Imperial's *array ackermannization* optimization, as well as differences in the tools' extensions to KLEE's expression language—sometimes resulting in wildly different execution times. SMT solvers rely heavily on heuristics to gain good performance so it is not unexpected that slight differences in constraints could result in different performance. However, we discovered that in KLEE, Z3 was used in a sub-optimal way. The Z3 API used by KLEE bypasses all of Z3's logic-specific strategies and uses the DPLL(T) [60] method (lazy translation to SAT) of solving constraints. This is frequently slower than using Z3's logic-specific strategies. On re-running our experiments after modifying the tools to use the more appropriate Z3 API, the performance of Aachen's tool increased slightly and that of Imperial's dramatically. When ranking the modified tools, Imperial ranked better on 51 benchmarks, with Aachen ranking better on only one and tying on the remaining 34 benchmarks. The better performance of Imperial's tool is due to the *array ackermannization* optimization which causes Z3 to use a logic-specific strategy—eager bit-blasting to SAT [48]—in place of the frequently slower DPLL(T)-based approach [77]. To maintain the integrity of our study, the results we present are of the tools using the sub-optimal Z3 API, but we shall use this insight when fully integrating floating-point support into upstream KLEE.

C. Threats to Validity

Our study has both internal and external threats to validity. Both tools use KLEE and Z3, so errors in these components may lead to bugs that go undetected when comparing the two implementations. However, the manual effort we put in writing specifications for the benchmarks renders this risk minimal.

Since both our tools were built on top of KLEE and Z3, our respective design decisions might be more similar than they would be had different frameworks been used. However, we found that in spite of this common infrastructure, building the extensions required important design decisions that resulted in significant differences. Moreover, having a common infrastructure made it possible to conduct a rigorous comparison that would not have been possible otherwise.

Our benchmarks might not be representative of floating-point code found in large deployed applications. However, our synthetic benchmarks are meant to systematically test challenging floating-point features that real applications would exercise, while our real-world benchmarks are based on existing applications or widely-used libraries.

Finally, due to the computational complexity of floating-point constraint solving, all benchmarks contain comparatively few floating-point operations and symbolic data. This is due to the fact that constraint checking large numeric applications is currently infeasible in the presence of floating-point numbers.

VII. Related Work

Testing of floating-point programs has received significant attention from the research community (e.g. [4], [8], [22], [23], [39]), and various methods have been proposed for verifying floating-point properties through abstract interpretation [35], [36], [65], decision procedures [16], [40], [50], [81], [82], theorem proving [9], [10], and combinations of techniques [13], [14], [62], [63]. Recent work has also focused on estimating bounds on round-off error in floating-point computation [26], [55], [73].

Several works have investigated extending symbolic analysis to floating-point programs [6], [11], [12], [66], [67]. Ours is one of the few that integrates floating-point reasoning with a mature symbolic execution tool, KLEE, and our study is distinguished by our N-version programming approach. Thanks to our combined suite, we believe this is one of the largest-scale studies—in terms of number and diversity of benchmark programs—of symbolic execution in the context of floating point.

Reasoning about floating-point constraints in the context of symbolic execution has often been conducted in a manner that avoids the full complexity of floating-point, e.g. by approximating floating-point numbers with mathematical reals in Ariadne [7] and approximate solving of floating-point constraints using search-based methods in CORAL [74] and FloPSy [49]. KLEE-FP [24] and KLEE-CL [25] are two extensions of KLEE which add support for floating point in the limited context of reasoning about program equivalence. That is, these extensions can only test whether two purportedly equivalent floating-point implementations are indeed equivalent, but cannot perform general symbolic execution of floating-point code, nor can they perform test input generation.

A very recent study provides a large corpus of numerical software bugs, categorized as *accuracy* bugs, *special value* bugs, *convergence* bugs and *correctness* bugs [31]. The corpus includes bugs originating from C source code, including GSL, which could be adapted into a form suitable for analysis using our KLEE-based tools. With further effort, example bugs originating from software written in other languages could be ported to be suitable for analysis using our tools.

VIII. Conclusion and Lessons Learned

We have presented a case study on N-version programming for the independent, benchmark-driven development of two extensions to KLEE to support symbolic reasoning over floating-point arithmetic. We hope that our rigorous procedure for independently developing benchmarks, improving tool versions in response to the benchmarks, and then exchanging tools to evaluate similarities and differences, will provide a useful basis for future researchers in the position where an analogous controlled study is feasible. We conclude by recapitulating the main contributions and key lessons we have learned from the experience.

Independently-developed benchmarks. Our independently-developed benchmarks served both as a practical specification and a target for tool optimization. Each team's benchmarks highlighted non-trivial correctness issues in the other team's tool, leading to a more reliable comparison overall. We strongly recommend the preparation of multiple independent benchmark suites when exploring other problem domains with equally rich design spaces.

Benchmark suite for floating-point symbolic execution and constraint solving. We regard the combined benchmark suite itself as an important contribution of this case study: it provides a source of benchmarks in a domain where existing suitable benchmarks may be hard to find. In our case, where many publicly available programs employ floating point, a set of benchmarks that is interesting enough to warrant symbolic analysis, yet not so large to be intractable, was not readily available; our study has led to such a benchmark suite, which is available publicly. Furthermore, our experiments generated a large number of SMT queries that we have contributed to the annual SMT competition [72]. Our queries helped start a new competition division involving a logic that combines the array, bitvector and floating-point theories; this should have a direct impact on software engineering research.

Dual implementations of floating-point symbolic executors. The very similar settings in which the two teams were innovating enabled a close comparison of design choices. Often this was interesting, allowing us to compare in detail e.g. the manner in which the teams supported long doubles, and approaches to handling floating-point types in KLEE's expression language. On the other hand, the similar setting perhaps reduced the chances of the teams making radically different design choices, which might have been harder to compare, but might have yielded more fundamental insights into how to approach floating-point symbolic execution. As discussed in the introduction, having two complementary tools enables differential testing to find tool bugs and portfolio analysis to speed up and improve analysis results, both advocated in the deployment stage of N-version programming. A downside of having dual implementations is the rigor associated with N-version programming, in which the two teams have to work independently and to a strict schedule, which often felt unnecessarily restrictive.

Combining design choices. Our case study places us in a good position to combine the strengths of each tool to yield a higher quality implementation of floating-point support in KLEE than either would have achieved individually. We briefly discuss four relevant aspects. (1) Support for vector instructions was handled in a very similar way by both teams and has already been incorporated upstream [2]. (2) KLEE's expression language is not typed, so a necessary first step is to make it typed. Once that is complete, Imperial's implementation of floating-point expressions could be used as it has fewer comparison operations, but then combined with Aachen's more robust explicit casting operations. (3) To support the `long double` type, we lean toward Aachen's approach as it is more complete by supporting non-IEEE-754 classes of floating-point numbers, but at the expense of more complicated constraints. (4) Imperial's array ackermannization optimization combined with the appropriate Z3 API resulted in notable performance gains, and should be incorporated.

Breakthroughs still needed. Our experience is that, despite significant recent advances in the field, breakthroughs are still required before floating-point constraint solving is efficient enough for scalable analysis: on a number of benchmarks, both tools missed bugs or achieved low coverage due to the intractability of the constraints to be solved. As well as innovating at the solver level, we envisage opportunities for using higher-level program analyses to simplify constraints.

Acknowledgements

This research is supported by EPSRC through a CASE studentship (sponsored by ARM) and two EPSRC Early Career Fellowships (EP/L002795/1 and EP/N026314/1); and by the European Research Council (ERC) under the EU's Horizon 2020 Research and Innovation Programme (grant agreement n. 647295 (SYMBIOSYS)).

REFERENCES

[1] "Competition on Software Verification (SV-COMP)," http://sv-comp.sosy-lab.org/.

[2] "KLEE pull request 657: Implement basic support for vectorized instructions," https://github.com/klee/klee/pull/657.

[3] "[SMT-COMP] Contribution of QF_FPBV and QF_FPABV benchmarks," http://cs.nyu.edu/pipermail/smt-comp/2017/000436.html.

[4] M. Aharoni, S. Asaf, L. Fournier, A. Koyfman, and R. Nagel, "FPgen - a test generation framework for datapath floating-point verification," in *Eighth IEEE International High-Level Design Validation and Test Workshop 2003, San Francisco, CA, USA, November 12-14, 2003*, 2003, pp. 17–22.

[5] A. Avizienis, "The N-version approach to fault-tolerant software," *IEEE Transactions on Software Engineering (TSE)*, vol. 11, pp. 1491–1501, December 1985.

[6] R. Bagnara, M. Carlier, R. Gori, and A. Gotlieb, "Symbolic path-oriented test data generation for floating-point programs," in *Proc. of the IEEE International Conference on Software Testing, Verification, and Validation (ICST'13)*, Mar. 2013.

[7] E. T. Barr, T. Vo, V. Le, and Z. Su, "Automatic detection of floating-point exceptions," in *Proc. of the 40th ACM Symposium on the Principles of Programming Languages (POPL'13)*, Jan. 2013.

[8] ——, "Automatic detection of floating-point exceptions," in *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, 2013, pp. 549–560.

[9] S. Boldo and J.-C. Filliâtre, "Formal verification of floating-point programs," in *Computer Arithmetic, 2007. ARITH'07. 18th IEEE Symposium on*. IEEE, 2007, pp. 187–194.

[10] S. Boldo, J.-C. Filliâtre, and G. Melquiond, "Combining Coq and Gappa for certifying floating-point programs," in *International Conference on Intelligent Computer Mathematics*. Springer, 2009, pp. 59–74.

[11] M. Borges, M. d'Amorim, S. Anand, D. Bushnell, and C. S. Pasareanu, "Symbolic execution with interval solving and meta-heuristic search," in *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, ser. ICST '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 111–120. [Online]. Available: http://dx.doi.org/10.1109/ICST.2012.91

[12] B. Botella, A. Gotlieb, and C. Michel, "Symbolic Execution of Floating-point Computations," *Softw. Test. Verif. Reliab.*, vol. 16, no. 2, pp. 97–121, Jun. 2006. [Online]. Available: http://dx.doi.org/10.1002/stvr.v16:2

[13] M. Brain, V. D'Silva, A. Griggio, L. Haller, and D. Kroening, "Interpolation-Based Verification of Floating-Point Programs with Abstract CDCL," in *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*, 2013, pp. 412–432. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-38856-9_22

[14] ——, "Deciding floating-point logic with abstract conflict driven clause learning," *Formal Methods in System Design*, vol. 45, no. 2, pp. 213–245, 2014.

[15] M. Brain, C. Tinelli, P. Rümmer, and T. Wahl, "An Automatable Formal Semantics for IEEE-754 Floating-Point Arithmetic," Tech. Rep., (2015). [Online]. Available: http://smtlib.cs.uiowa.edu/papers/BTRW15.pdf

[16] A. Brillout, D. Kroening, and T. Wahl, "Mixed abstractions for floating-point arithmetic," in *2009 Formal Methods in Computer-Aided Design*, Nov 2009, pp. 69–76.

[17] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," in *Proc. of the 23rd IEEE International Conference on Automated Software Engineering (ASE'08)*, Sep. 2008.

[18] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," in *Proc. of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*, Dec. 2008.

[19] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler, "EXE: Automatically Generating Inputs of Death," in *Proc. of the 13th ACM Conference on Computer and Communications Security (CCS'06)*, Oct.-Nov. 2006.

[20] C. Cadar, P. Godefroid, S. Khurshid, C. Pasareanu, K. Sen, N. Tillmann, and W. Visser, "Symbolic Execution for Software Testing in Practice—Preliminary Assessment," in *Proc. of the 33rd International Conference on Software Engineering, Impact Track (ICSE Impact'11)*, May 2011.

[21] L. Chen and A. Avizienis, "N-version programming: A fault-tolerance approach to reliability of software operation," in *Proc. of the 8th IEEE International Symposium on Fault Tolerant Computing (FTCS'78)*, Jun. 1978.

[22] H. Collavizza, C. Michel, O. Ponsini, and M. Rueher, "Generating test cases inside suspicious intervals for floating-point number programs," in *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis, CSTVA 2014, Hyderabad, India, May 31, 2014*, 2014, pp. 7–11.

[23] H. Collavizza, C. Michel, and M. Rueher, "Searching Critical Values for Floating-Point Programs," in *Testing Software and Systems - 28th IFIP WG 6.1 International Conference, ICTSS 2016, Graz, Austria, October 17-19, 2016, Proceedings*, 2016, pp. 209–217.

[24] P. Collingbourne, C. Cadar, and P. H. Kelly, "Symbolic crosschecking of floating-point and SIMD code," in *Proc. of the 6th European Conference on Computer Systems (EuroSys'11)*, Apr. 2011.

[25] ——, "Symbolic testing of OpenCL code," in *Proc. of the Haifa Verification Conference (HVC'11)*, Dec. 2011.

[26] E. Darulova and V. Kuncak, "Sound compilation of reals," in *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*. ACM, 2014, pp. 235–248. [Online]. Available: http://doi.acm.org/10.1145/2535838.2535874

[27] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Proc. of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, Mar.-Apr. 2008.

[28] "Floating Point correctness infrastructure repository," https://github.com/delcypher/symex-fp-bench.

[29] "Aachen floating point correctness benchmark repository," https://github.com/danielschemmel/fp-benchmarks-aachen.

[30] "Imperial floating point correctness benchmark repository," https://github.com/delcypher/fp-benchmarks-imperial.

[31] A. D. Franco, H. Guo, and C. Rubio-González, "A Comprehensive Study of Real-World Numerical Bug Characteristics," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, M. D. Penta and T. N. Nguyen, Eds. Urbana-Champaign, IL: IEEE/ACM, 2017.

[32] "gcov – A Test Coverage Program," https://gcc.gnu.org/onlinedocs/gcc/Gcov.html, 2017.

[33] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *Proc. of the Conference on Programing Language Design and Implementation (PLDI'05)*, Jun. 2005.

[34] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *Proc. of the 15th Network and Distributed System Security Symposium (NDSS'08)*, Feb. 2008.

[35] E. Goubault, "Static Analyses of the Precision of Floating-Point Operations," in *Static Analysis, 8th International Symposium, SAS 2001, Paris, France, July 16-18, 2001, Proceedings*, 2001, pp. 234–259.

[36] E. Goubault, M. Martel, and S. Putot, "Asserting the Precision of Floating-Point Computations: A Simple Abstract Interpreter," in *Programming Languages and Systems, 11th European Symposium on Programming, ESOP 2002, held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, 2002, pp. 209–212.

[37] T. Granlund and the GMP development team, *GNU MP: The GNU Multiple Precision Arithmetic Library*, 6th ed., 2016, http://gmplib.org/.

[38] "GNU Scientific Library," https://www.gnu.org/software/gsl/, 2017.

[39] Y. Gu, T. Wahl, M. Bayati, and M. Leeser, "Behavioral Non-portability in Scientific Numeric Computing," in *Euro-Par 2015: Parallel Processing - 21st International Conference on Parallel and Distributed Computing, Vienna, Austria, August 24-28, 2015, Proceedings*, 2015, pp. 558–569.

[40] L. Haller, A. Griggio, M. Brain, and D. Kroening, "Deciding floating-point logic with systematic abstraction," in *Formal Methods in Computer-Aided Design, FMCAD 2012, Cambridge, UK, October 22-25, 2012*, 2012, pp. 131–140.

[41] "IEEE Standard for Floating-Point Arithmetic," Institute of Electrical and Electronics Engineers, Standard, (2008).

[42] Intel, *Intel(R) 64 and IA-32 Architectures Software Devloper's Manual*, 2016, Volume 1, 8.2.2 Unsupported Double Extended-Precision Floating-Point Encodings and Pseudo-Denormals.

[43] "Information technology – Microprocessor Systems – Floating-Point arithmetic," International Organization for Standardization, Geneva, Switzerland, Standard, (2011).

[44] "Programming languages – C," International Organization for Standardization, Geneva, Switzerland, Standard, (1999).

[45] "Information technology – Programming languages – C," International Organization for Standardization, Geneva, Switzerland, Standard, (2011).

[46] W. Kahan, "Paranoia," http://netlib.org/paranoia/paranoia.c, 1987.

[47] R. Karpinski, "Paranoia-a floating-point benchmark," *Byte*, vol. 10, no. 2, p. 223, 1985.

[48] D. Kroening and O. Strichman, *Decision Procedures: An Algorithmic Point of View*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, ch. Bit Vectors, pp. 160–163.

[49] K. Lakhotia, N. Tillmann, M. Harman, and J. de Halleux, "FloPSy - Search-Based Floating Point Constraint Solving for Symbolic Execution," in *Testing Software and Systems: 22nd IFIP WG 6.1 International Conference, ICTSS 2010, Natal, Brazil, November 8-10, 2010. Proceedings*, A. Petrenko, A. Simão, and J. C. Maldonado, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 142–157. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-16573-3_11

[50] M. Leeser, S. Mukherjee, J. Ramachandran, and T. Wahl, "Make it real: Effective floating-point reasoning via exact arithmetic," in *Design, Automation & Test in Europe Conference & Exhibition, DATE 2014, Dresden, Germany, March 24-28, 2014*, 2014, pp. 1–4.

[51] D. Liew, C. Cadar, and A. Donaldson, "KLEE Floating Point Extensions Team Imperial," https://github.com/srg-imperial/klee-float.

[52] "LLVM Bug 30781: APFloat does not correctly handle signaling NaN," https://bugs.llvm.org/show_bug.cgi?id=30781.

[53] "LLVM Bug 31292: llvm::APFloat Assertion 'lost_fraction == lfExactlyZero' failed," https://bugs.llvm.org/show_bug.cgi?id=31292.

[54] "LLVM Bug 31294: llvm::APFloat when using x87DoubleExtended semantics does not handle unsupported operands properly," https://bugs.llvm.org/show_bug.cgi?id=31294.

[55] V. Magron, G. A. Constantinides, and A. F. Donaldson, "Certified Roundoff Error Bounds Using Semidefinite Programming," *ACM Trans. Math. Softw.*, vol. 43, no. 4, pp. 34:1–34:31, 2017. [Online]. Available: http://doi.acm.org/10.1145/3015465

[56] W. M. McKeeman, "Differential testing for software," *Digital Technical Journal*, vol. 10, pp. 100–107, 1998.

[57] D. Menendez, S. Nagarakatte, and A. Gupta, "Alive-FP: Automated Verification of Floating Point Based Peephole Optimizations in LLVM," in *International Static Analysis Symposium*. Springer, 2016, pp. 317–337.

[58] D. Merkel, "Docker: Lightweight Linux Containers for Consistent Development and Deployment," *Linux J.*, vol. 2014, no. 239, Mar. 2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2600239.2600241

[59] D. Monniaux, "The pitfalls of verifying floating-point computations," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 30, no. 3, pp. 12:1–12:41, May 2008.

[60] R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "Solving SAT and SAT Modulo Theories: From an Abstract Davis–Putnam–Logemann–Loveland Procedure to DPLL(T)," *J. ACM*, vol. 53, no. 6, pp. 937–977, Nov. 2006. [Online]. Available: http://doi.acm.org/10.1145/1217856.1217859

[61] C. S. Păsăreanu, W. Visser, D. Bushnell, J. Geldenhuys, P. Mehlitz, and N. Rungta, "Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis," *Automated Software Engineering*, vol. 20, no. 3, pp. 391–425, Sep 2013.

[62] O. Ponsini, C. Michel, and M. Rueher, "Combining Constraint Programming and Abstract Interpretation for Value Analysis of Floating-point Programs," in *Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, April 17-21, 2012*, 2012, pp. 775–776.

[66] M. Quan, "Hotspot Symbolic Execution of Floating-Point Programs," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: ACM, 2016, pp. 1112–1114. [Online]. Available: http://doi.acm.org/10.1145/2950290.2983966

[63] ——, "Verifying floating-point programs with constraint programming and abstract interpretation techniques," *Autom. Softw. Eng.*, vol. 23, no. 2, pp. 191–217, 2016. [Online]. Available: http://dx.doi.org/10.1007/s10515-014-0154-2

[64] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes 3rd Edition: The Art of Scientific Computing*, 3rd ed. New York, NY, USA: Cambridge University Press, 2007.

[65] S. Putot, E. Goubault, and M. Martel, "Static Analysis-Based Validation of Floating-Point Computations," in *Numerical Software with Result Verification, International Dagstuhl Seminar, Dagstuhl Castle, Germany, January 19-24, 2003, Revised Papers*, 2003, pp. 306–313.

[67] J. Ramachandran, C. S. Pasareanu, and T. Wahl, "Symbolic Execution for Checking the Accuracy of Floating-Point Programs," *ACM SIGSOFT Software Engineering Notes*, vol. 40, no. 1, pp. 1–5, 2015.

[68] P. Rümmer and T. Wahl, "An SMT-LIB Theory of Binary Floating-Point Arithmetic," in *International Workshop on Satisfiability Modulo Theories (SMT)*, 2010, p. 151. [Online]. Available: http://www.cprover.org/SMT-LIB-Float/smt-fpa.pdf

[69] D. Schemmel, R. Zähl, and K. Wehrle, "KLEE Floating Point Extensions Team Aachen," https://github.com/comsys/klee-float.

[70] K. Sen, D. Marinov, and G. Agha, "CUTE: A concolic unit testing engine for C," in *Proc. of the joint meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'05)*, Sep. 2005.

[71] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Address-Sanitizer: A fast address sanity checker," in *Proc. of the 2012 USENIX Annual Technical Conference (USENIX ATC'12)*, Jun. 2012.

[72] "SMT-COMP competition 2006," http://www.csl.sri.com/users/demoura/smt-comp, Aug. 2006.

[73] A. Solovyev, C. Jacobsen, Z. Rakamaric, and G. Gopalakrishnan, "Rigorous Estimation of Floating-Point Round-off Errors with Symbolic Taylor Expansions," in *FM 2015: Formal Methods - 20th International Symposium, Oslo, Norway, June 24-26, 2015, Proceedings*, ser. Lecture Notes in Computer Science, vol. 9109. Springer, 2015, pp. 532–550. [Online]. Available: https://doi.org/10.1007/978-3-319-19249-9_33

[74] M. Souza, M. Borges, M. d'Amorim, and C. S. Păsăreanu, *CORAL: Solving Complex Constraints for Symbolic PathFinder*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 359–374. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-20398-5_26

[75] N. Tillmann and J. De Halleux, "Pex: white box test generation for .NET," in *Proc. of the 2nd International Conference on Tests and Proofs (TAP'08)*, Apr. 2008.

[76] "Undefined Behavior Sanitizer," https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html, 2017.

[77] "Z3 issue 1035: Unexpected huge performance difference between solver using "default" tactic, Z3_mk_solver() and Z3_mk_simple_solver()," https://github.com/Z3Prover/z3/issues/1035, 2017.

[78] "Z3 issue 1251: Overflow encountered when expanding vector," https://github.com/Z3Prover/z3/issues/1251, 2017.

[79] "Z3 issue 577: No theory of floating point, bitvectors and arrays and mixed performance?" https://github.com/Z3Prover/z3/issues/577.

[80] "Z3 issue 740: Invalid model generated without warning," https://github.com/Z3Prover/z3/issues/740.

[81] A. Zeljic, C. M. Wintersteiger, and P. Rümmer, "Approximations for Model Construction," in *Automated Reasoning - 7th International Joint Conference, IJCAR 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 19-22, 2014. Proceedings*, 2014, pp. 344–359.

[82] ——, "An Approximation Framework for Solvers and Decision Procedures," *J. Autom. Reasoning*, vol. 58, no. 1, pp. 127–147, 2017.