A Verified Certificate Checker for Finite-Precision Error Bounds in Coq and HOL4

Heiko Becker*, Nikita Zyuzin*, Raphaël Monat†, Eva Darulova*, Magnus O. Myreen‡ and Anthony Fox§
*MPI-SWS, †ENS Lyon, ‡Chalmers University of Technology, §University of Cambridge
*{hbecker,zyuzin,eva}@mpi-sws.org, †raphael.monat@ens-lyon.org, ‡myreen@chalmers.se, §anthony.fox@arm.com

***Abstract*—Being able to soundly estimate roundoff errors of finite-precision computations is important for many applications in embedded systems and scientific computing. Due to the discrepancy between continuous reals and discrete finite-precision values, automated static analysis tools are highly valuable to estimate roundoff errors. The results, however, are only as correct as the implementations of the static analysis tools. This paper presents a formally verified and modular tool which fully automatically checks the correctness of finite-precision roundoff error bounds encoded in a certificate. We present implementations of certificate generation and checking for both Coq and HOL4 and evaluate it on a number of examples from the literature. The experiments use both in-logic evaluation of Coq and HOL4, and execution of extracted code outside of the logics: we benchmark Coq extracted unverified OCaml code and a CakeML-generated verified binary.**

I. Introduction

Numerical programs, common in scientific computing or embedded systems, are often implemented in finite-precision arithmetic. This approximation of real numbers inevitably introduces roundoff errors, potentially making the computed results unacceptably inaccurate. The discrepancy between discrete finite-precision arithmetic and continuous real arithmetic make accurate and sound error estimation challenging. Automated tool support is thus highly valuable.

This fact was already recognized previously and resulted in a number of static analysis techniques and tools [17, 37, 10, 13] for computing sound worst-case absolute error bounds on numerical errors. The results of such static analysis tools are, however, only as correct as the tools' implementation.

Some of these tools provide independently checkable formal proofs, however we found that none of the current certificate producing tools, FPTaylor [37], PRECiSa [31] and Gappa [13] go far enough. FPTaylor produces a proof certificate in HOL-Light, relying on an in-logic decision procedure [36]. Its analysis is specific to floating-point arithmetic and does not support other finite precisions. PRECiSa and Gappa generate a proof certificate by instantiating library theorems, explicitly encoding verification steps. Any tool that explicitly encodes verification steps, or is to be used interactively [14, 34] requires expert knowledge in IEEE754 floating-point semantics [20] or formal verification; in contrast our goal is to make our tool usable by non-experts. Finally, in-logic verification of certificates can often become unreasonably slow.

This paper describes a new fully automated tool, called *FloVer*, which checks proof certificates of finite-precision roundoff error bounds generated by static analysis tools. Certificates checked by FloVer encode only the minimal static analysis result, and thus using FloVer does not require formal verification expertise. Separately from FloVer, we implement fully automated certificate generation in the static analysis tool Daisy [12], demonstrating our envisioned tool-chain.

FloVer supports straight-line arithmetic kernels, floating-point as well as fixed-point arithmetic, mixed-precision evaluation (including floating-point type inference), and local variable declarations. For floating-point expressions, FloVer proves correctness of each analyzed expression with respect to the concrete bit-level IEEE754 floating-point semantics [20]. Our tool is formally verified in both Coq and HOL4. A succesful run of FloVer shows that the encoded roundoff error is a valid upper bound and that the analyzed function can be run without any errors (e.g. division-by-zero).

In order to handle both floating-point and fixed-point arithmetic, FloVer supports a forward dataflow static analysis. FloVer is furthermore built modularly to allow reusability and easy extensions, and supports dataflow analysis with both interval and affine arithmetic abstract domains.

We have implemented and verified FloVer in two theorem provers to be able to connect to projects in both provers and thereby make FloVer widely applicable. In Coq, we hope to link to the CompCert compiler [26] and CertiCoq [2]; and in HOL4 we already link to CakeML [38].

The connection to CakeML allows us to provide efficient certificate checking: using the CakeML toolchain [38, 33] we produce a verified binary of our certificate checker. At the time of writing, CertiCoq was not capable of extracting our checker functions, thus we extract an unverified binary from Coq and compare its perfomance with the verified CakeML binary.

Our evaluation on standard benchmarks from embedded systems and scientific computing shows that roundoff errors verified by FloVer are competitive with the state of the art, and extracted certificate checking times are significantly faster than in-logic verification.

Contributions

- We explain our modular, fully automated and self-contained approach to certification of absolute finite-precision roundoff error bounds (Section IV and V).
- We implement and prove FloVer correct in both Coq and HOL4. The sources are available at https://gitlab.mpi-sws.org/AVA/FloVer.

- We are the first to provide an efficient and verified way of checking finite-precision error certificates by extracting a verified binary version of FloVer from HOL4 (Section VI).
- We experimentally evaluate (in Section VII) implementations of FloVer on examples from the literature. The results are competitive and show that our approach to certificate checking is feasible. During our experiments, we found a subtle bug in the Daisy static analyzer.

II. Overview

In this section, we give a high-level overview of our certificate generation and checking approach. The next section provides the necessary background on finite precision arithmetic and static dataflow analysis for roundoff errors. Section IV describes the technical details of FloVer.

A certificate (in Coq or HOL4) checked by FloVer encodes the *result* of a forward dataflow static analysis of roundoff errors, but not the analysis or correctness proofs themselves. For each analyzed arithmetic expression (consisting of $+, -, *, /$, `FMA`, and local variables), the certificate contains:

- the expression f, as an abstract syntax tree (AST)
- a precondition P, specifying the domain (interval) of all input variables
- a (possibly mixed-precision) type assignment Γ for all input variables and optionally let-bound variables,
- the analysis result which consists of a range $\Phi_{\mathcal{R}}$ and an error bound $\Phi_{\mathcal{E}}$ for each intermediate subexpression

FloVer then checks the analysis result recursively, by verifying for each AST node that the error bound is a sound upper bound on the worst-case absolute roundoff error:

$$\max_{x \in [a,b]} |f(x) - \tilde{f}(\tilde{x})| \tag{1}$$

where f and x are the real-valued expression and variable, respectively, and $\tilde{f}$ and $\tilde{x}$ their finite-precision counterparts. The interval $[a, b]$ is the domain of x given by precondition P. Ranges for input variables as well as the analysis result are necessary as (absolute) finite-precision roundoff errors depend on the magnitude of the computed values. In the absence of input ranges, roundoff errors are unbounded in general.

FloVer splits the certification into several subtasks and runs separate validator functions (see also Figure 1):

- `validRealRange` validates the range result $\Phi_{\mathcal{R}}$,
- `validTypes` infers and checks types (given in Γ) of all subexpressions
- `validErrors` validates the error results $\Phi_{\mathcal{E}}$,
- `validMachineRanges` validates that no overflow and NaN's (not-a-number special values) occur.

We have implemented the validators in both Coq and HOL4 and proven an overall soundness theorem: when all validators return successfully, then the computed error bounds (for each subexpression) are soundly overapproximating the finite-precision roundoff errors.

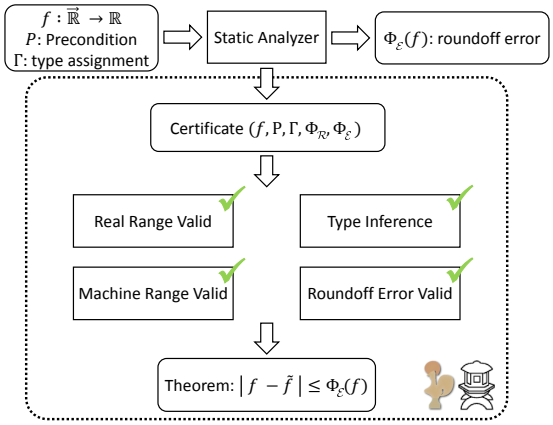

Fig. 1. Overview of the FloVer framework

To verify a certificate, one can run the validator functions in Coq or HOL4 directly. However, while both provers natively support evaluation of functions, this is not particularly efficient. To speed up the certificate checkers, we have used the CakeML in-logic compilation toolchain [33], to extract a *verified* binary from our HOL4 checker definitions. Since the CakeML compiler is fully verified, the binary enjoys the very same correctness guarantees as the certificate checkers implemented in HOL4. Similarly, we have used the extraction mechanism [27] in Coq to extract an, albeit unverified, binary. The binary implementations of the checkers run natively and are thus significantly more efficient, as our experiments in Section VII demonstrate.

III. Background

A. Finite-Precision Arithmetic

FloVer uses a general abstraction for finite-precision arithmetic relating it to operations on real numbers:

$$x \circ_{fp} y = (x \circ y) + \mathrm{error}(x \circ y, fp) \tag{2}$$

where $\circ \in \{+, -, *, /\}$ and $\circ_{fp}$ denotes the corresponding finite-precision operation at type fp. Function $\mathrm{error}(e, fp)$ computes the error from representing the real value e in the finite-precision type fp. An input x may not be representable in finite-precision arithmetic, and thus FloVer considers an initial error on the input: $|x - \tilde{x}| \leq \mathrm{error}(x, fp)$.

For floating-point arithmetic, we assume IEEE754 [20] semantics with rounding-to-nearest rounding mode and the standard abstraction of arithmetic operations:

$$\mathrm{error}(e, fp) = e * \delta \quad |\delta| \leq \varepsilon_{fp} \tag{3}$$

Constant ε_{fp} is the so-called machine epsilon for precision fp ($fp = 16$, 32 or 64 bits) and represents the maximum relative error for a single arithmetic operation. In addition to binary operations, FloVer also supports unary negation, which does not incur a roundoff error, and fused-multiply-add instructions, where $\mathrm{FMA}(x, y, z)_{fp} = (x * y + z) + \mathrm{error}(x * y + z, fp)$.

Equation 3 holds under IEEE754 floating-point semantics only for normal floating-point values, and thus FloVer reports ranges containing only subnormals, infinity or not a number (NaN) special values as errors. We discuss the proof of correctness wrt. to IEEE754 semantics in Section V-A.

Fixed-point arithmetic is an alternative to floating-points which does not require dedicated hardware and is thus a common choice in embedded systems. No standard exists, but we follow the common representation [3] of fixed-point values as bit vectors with an integer and a fractional part, separated by an implicit radix point, which has to be precomputed at compile-time. We assume truncation as the rounding mode for arithmetic operations. The absolute roundoff error at each operation is determined by the fixed-point format, i.e. the (implicit) number of fractional bits available, which in turn can be computed from the range of possible values at that operation. Since this information must be computed by any static analysis on fixed-point programs, we encode fractional bits as part of our fixed-point type and rely on the certificate containing a full (unverified) map Γ from expressions to types for fixed-point kernels.

B. Static Dataflow Roundoff Error Analysis

FloVer's range and error validators perform dataflow roundoff error analysis and for this follow the same approach for computing absolute error bounds as Rosa [10], Fluctuat [17], Gappa [13] and Daisy [12].

The magnitude of absolute finite-precision roundoff errors depends on the magnitude of values of all intermediate subexpressions (this can be seen e.g. from Equation 3). Thus, in order to accurately bound roundoff errors, the analysis first needs to be able to bound the ranges of all (intermediate) expressions.

At a conceptual level, dataflow analysis computes roundoff error bounds in two steps:

range analysis computes sound range bounds for all intermediate expressions,

error analysis propagates errors from subexpressions and computes the new worst-case roundoffs using the previously computed ranges.

Both steps are performed recursively on the AST of the arithmetic expressions. A side effect of this separation is that it provides us with a modular approach: we can choose different range arithmetics with different accuracy-efficiency tradeoffs for ranges and errors. Common choices for range arithmetics are interval arithmetic (IA) [30] and affine arithmetic(AA) [15].

IV. Certification of Error Analysis Results

Next, we focus on the technical details of our certificate checking. The certificates in Coq, HOL4 and for the extracted binaries are structurally the same and only differ in syntax. Figure 2 shows a sample structure of a certificate in Coq and HOL4, including the types of encoded results. Γ represents a type assignment to all free variables in the analyzed function. Expressions (of type `expr`) are parametric in the type of constants. $\Phi_{\mathcal{R}}$ and $\Phi_{\mathcal{E}}$ map each AST node of the analyzed function to an interval and a positive (absolute) error bound represented by a single fraction, respectively. We discuss the differing types of $\Phi_{\mathcal{R}}$ and $\Phi_{\mathcal{E}}$ in Section V-C.

The validator functions, which check the certificate, also have the same structure in both Coq and HOL4 and we describe them here independently of the particular prover.

A. Checking Range Analysis Results

The range validator is implemented in the function $\texttt{validRealRange}(e, P, \Phi_{\mathcal{R}})$ which takes as input an expression e, the precondition P, which captures the constraints on the input variables, and the real-valued ranges which are to be checked in $\Phi_{\mathcal{R}}$. `validRealRange` verifies by structural recursion on the AST that for each subexpression e' of e, $\Phi_{\mathcal{R}}(e')$ returns a sound enclosure of the true range, which is computed inside the theorem prover with interval or affine arithmetic. That is, we check the ranges in $\Phi_{\mathcal{R}}$ by effectively recomputing them inside the prover.

Since FloVer supports let-bindings in the input program to reuse evaluation results, both at runtime as well as in the certificate validator, we extend `validRealRange` to handle let-bound variables without recomputing results.

B. Mixed-precision Support

Mixed-precision evaluation allows different arithmetic operations to be executed in different precisions. This often allows to speed up computations as evaluation in lower precisions is usually faster. Instead of requiring e.g. uniform 64-bit precision, each subexpression in FloVer can be evaluated in 16, 32 or 64 bit floating-point precisions (each with the corresponding machine epsilon ε_p). FloVer supports the same semantics as C and Scala: for two operands with different precisions, the lower one is implicitly cast to the higher precision, but an explicit cast is required when decreasing precision (e.g. when assigning a 64 bit value to a 32 bit variable).

The typing environment Γ assigns a machine precision to every free variable of the analyzed expression. We further require any constant in the AST, as well as casts to be annotated with its (resulting) precision.

For floating-point precisions, FloVer infers the remaining types automatically, i.e. the user only has to provide this necessary minimal information, and in particular does not need to annotate all intermediate operations.

We can reuse the existing infrastructure to support fixed-point arithmetic. A fixed-point type in FloVer is then represented as a pair of word length w and number of fractional bits f. For fixed-point precisions, FloVer avoids recomputing the fractional bits and thus relies on the information being encoded in Γ.

When checking a certificate, FloVer computes a full type-map $\Phi_{\mathcal{T}}$ from the (partial) type map Γ to avoid recomputing results. To this end we implement the function $\texttt{validTypes}(\Gamma, e)$. The function returns $\Phi_{\mathcal{T}}$ if and only if all types encoded in Γ are valid types for their respective subexpressions. We reuse $\Phi_{\mathcal{T}}$ in both the error validator and the machine range validator.

```
Definition f:cmd Q := <AST f>.
(* Type assignment for free variables *)
Definition Gamma: expr Q → option mType := <Γ>.
(* range constraints on free variables of f *)
Definition Precondition: nat → (Q * Q) := <P>.
(* map from sub-expressions to ranges and errors *)
Definition AbsEnv: expr Q → option ((Q * Q) * Q) :=
 <Φ_R, Φ_E>.

Theorem CertificateCheckingSucceeds =
 CertificateChecker f Gamma Precond AbsEnv = true.
Proof.
 vm_compute; auto.
Qed.
```

```
val f_def = Define `f: real cmd = <AST f>`;
(* Type assignment for free variables *)
val Gamma_def = Define
  `Gamma: real expr → mType option = <Γ>`;
(* range constraints on free variables of f *)
val Precondition_def = Define `
  P: num → (real * real) = <P>`;
(* map from sub-expressions to ranges and errors *)
val AbsEnv_def = Define `
  AbsEnv: real expr → ((real * real) * real) option =
    <Φ_R, Φ_E>`;
val CertificateCheckingSucceeds = prove (
 ``CertificateChecker f Gamma Precond AbsEnv``,
 daisy_eval_tac);
```

Fig. 2. Certificate structure with corresponding types in Coq (left) and HOL4 (right)

C. Checking Error Analysis Results

The error validator $\texttt{validErrors}(e, \Phi_\mathcal{T}, \Phi_\mathcal{R}, \Phi_\mathcal{E})$ takes as input the expression e, a type assignment to subexpressions $\Phi_\mathcal{T}$, the range analysis result $\Phi_\mathcal{R}$ and the error analysis result $\Phi_\mathcal{E}$, which is to be checked.That is, `validErrors` assumes that the ranges and types have been verified independently. As for the range validator, we extend `validErrors` to reuse results of let-bound variables. The validator function checks by structural recursion on the AST of e that for each subexpression e' of e, $\Phi_\mathcal{E}(e')$ is a sound upper bound on the absolute roundoff error.

For constants and variables, the error bounds are straightforwardly derived using Equation 2 and the range analysis result. For arithmetic operations, the error check is more involved. Using Equation 2, Equation 1 and the triangle inequality, we obtain for an addition:

$$\begin{aligned}|(e_1 + e_2) - (\tilde{e}_1 +_{fp} \tilde{e}_2)| \leq \\ |e_1 - \tilde{e}_1| + |e_2 - \tilde{e}_2| + \text{error}((\tilde{e}_1 + \tilde{e}_2), fp)\end{aligned} \tag{4}$$

$|e_1 - \tilde{e}_1|$ and $|e_2 - \tilde{e}_2|$ are the roundoff errors of the operands, which are propagated simply by addition. $\text{error}((\tilde{e}_1 + \tilde{e}_2), fp)$ is the new roundoff error commited by the addition at precision fp. The new roundoff error depends on the magnitude of the operands and thus on the ranges of $\tilde{e}_1$ and $\tilde{e}_2$.

The computation of an upper bound to Equation 4 then uses the range analysis result from $\Phi_\mathcal{R}$, the already verified error bounds on the subexpressions e_1 and e_2 in $\Phi_\mathcal{E}$, and basic properties of range arithmetic.

Similar bounds can be derived for the other arithmetic operations. However, for multiplication and division, the propagation of errors is more involved. For $e_1 * e_2$ we obtain $|(e_1 * e_2) - (\tilde{e_1} *_{fp} \tilde{e_2})| \leq |e_1 * e_2 - \tilde{e_1} * \tilde{e_2}| + \text{error}(\tilde{e_1} * \tilde{e_2}, fp)$ and similarly for division:

$$\begin{aligned}|(e_1/e_2) - (\tilde{e_1}/_{fp}\tilde{e_2})| \leq \\ |e_1 * (1/e_2) - \tilde{e_1} * (1/\tilde{e_2})| + \text{error}(\tilde{e_1} * 1/\tilde{e_2}, fp)\end{aligned}$$

FloVer checks whether a division by zero may occur during the execution of the analyzed function under the real-valued as well as the finite-precision semantics. If it detects that a division by zero can occur in any of the executions, certificate checking fails.

D. Supported Range Arithmetics

FloVer currently supports interval arithmetic (IA) [30] in both provers and affine arithmetic (AA) [15] in Coq to check real-valued ranges. The support for AA in the error validator in Coq as well as the HOL4 development in general is currently work in progress. Arithmetic operations in IA are efficiently computed as: $x \circ^\# y = [\min(x \circ y), \max(x \circ y)], \quad \circ \in \{+, -, *, /\}$. IA cannot track correlations between variables (e.g. it cannot show that $e_1 - e_1 \in [0, 0]$). Affine arithmetic is a simple relational analysis which tracks linear correlations and thus computes ranges for linear operations exactly (like the $e_1 - e_1$); for nonlinear operations it nonetheless has to compute an over-approximation.

V. Soundness

We have proven in both Coq and HOL4 that it suffices to run the validator functions on a certificate to show a) that the static analysis result is correct, and b) that the analyzed function will always evaluate to a finite value. The overall soundness proof relates a succeeding run of the validators `validTypes`, `validRealRange`, `validMachineRanges` and `validErrors` to the semantics of the analyzed function.

We have formalized the semantics of functions according to Equation 2. The rule for binary addition, for instance, is

$$\frac{\begin{array}{c} m_+ = m_1 \sqcup m_2 \\ \Phi_\mathcal{T}(e_1) = m_1 \quad \Phi_\mathcal{T}(e_2) = m_2 \quad \Phi_\mathcal{T}(e_1 + e_2) = m_+ \\ (e_1, E, \Phi_\mathcal{T}) \Downarrow (v_1, m_1) \qquad (e_2, E, \Phi_\mathcal{T}) \Downarrow (v_2, m_2) \end{array}}{(e_1 + e_2, E, \Phi_\mathcal{T}) \Downarrow ((v_1 + v_2) + \text{error}(v_1 + v_2,\, m_+))}$$

E is the environment tracking values of bound variables, and Γ tracks precisions of variables. $(e_1, E, \Phi_\mathcal{T}) \Downarrow (v_1, m_1)$ means that expression e_1 big-step evaluates for the variable environment E and the type assignment $\Phi_\mathcal{T}$ to value v_1 in precision m_1. $m_1 \sqcup m_2$ is an upper bound operator on precisions, returning the most precise of m_1 and m_2.

Real-valued executions map every variable, constant and cast operation to infinite (real-valued) precision, which we denote by $m = \infty$. The rules for subtraction, multiplication, division, casts, and FMA's are defined analogously. Unary negation does not introduce a new roundoff error and keeps the precision of the operand.

Analogously to expressions, we will use E to refer to the idealized real-valued environment and $\tilde{E}$ for the finite-precision environment. The overall soundness theorem is then

Theorem 1. *Let f be a real-valued function, E a real-valued environment, $\tilde{E}$ its finite-precision counterpart, P a precondition constraining the free variables of f, Γ a map from all free variables of f to a precision, $\Phi_{\mathcal{R}}$ a range analysis result, $\Phi_{\mathcal{T}}$ a type-map and $\Phi_{\mathcal{E}}$ an error analysis result. Then*

$$
\begin{gathered}
E \sim_{(\Phi_{\mathcal{E}},\mathcal{V},\mathcal{D},\Phi_{\mathcal{T}})} \tilde{E} \wedge \\
\texttt{validTypes}(\Gamma, f) = \Phi_{\mathcal{T}} \wedge \texttt{validRealRange}(f, P, \Phi_{\mathcal{R}}) \wedge \\
\texttt{validMachineRanges}(f, \Phi_{\mathcal{T}}, \Phi_{\mathcal{R}}, \Phi_{\mathcal{E}}) \wedge \\
\texttt{validErrors}(f, \Phi_{\mathcal{T}}, \Phi_{\mathcal{R}}, \Phi_{\mathcal{E}}) \implies \\
\exists\, v\, \tilde{v}_1\, m_1.\ (f, E, \Phi_{\mathcal{T}}) \Downarrow (v, \infty) \wedge (\tilde{f}, \tilde{E}, \Phi_{\mathcal{T}}) \Downarrow (\tilde{v}_1, m_1) \wedge \\
(\forall \tilde{v}_2\, m_2.\ (\tilde{f}, \tilde{E}, \Phi_{\mathcal{T}}) \Downarrow (v_2, m_2) \Rightarrow |v - \tilde{v}_2| \leq \Phi_{\mathcal{E}}(f))
\end{gathered}
$$

The assumption $E \sim_{(\Phi_{\mathcal{E}},\mathcal{V},\mathcal{D},\Phi_{\mathcal{T}})} \tilde{E}$ states that the real-valued environment E and the finite-precision environment $\tilde{E}$ agree up to a fixed δ on the values of the variables in the sets $\mathcal{V}$ and $\mathcal{D}$. We give the full explanation when explaining soundness of the error validator. To prove the theorem, we have split the proof into separate soundness proofs for each validator function. Each theorem is shown by structural induction on e.

a) Type Validator: Giving the full type map $\Phi_{\mathcal{T}}$ is tedious to do for a user. FloVer thus requires only annotations for casts, constants and (let-bound) variables, and infers the remaining types ($\Phi_{\mathcal{T}}$) fully automatically for floating-point expressions. For fixed-point types only, we require Γ to be a complete map since we rely on the fractional bits to be inferred externally.

Soundness of the type inference `validTypes` means that when $\Phi_{\mathcal{T}}(e) = m_t$ and evaluation of e gives value v and precision m_v, then $m_t = m_v$. Thus, we need not recompute type information once the type map has been computed and reuse it in the other validators.

b) Real Range Validator: For `validRealRange`, the soundness theorem proves that if E binds variables in e to values that are within the range given by the precondition P, then e evaluates for environment E to v under a real-valued semantics and v is contained in $\Phi_{\mathcal{R}}(e)$.

c) Machine Range Validator: We prove that whenever `validMachineRanges` succeeds on expression e, valid type-map $\Phi_{\mathcal{T}}$ and valid error map $\Phi_{\mathcal{E}}$, then any evaluation of e results in a finite, representable value for the type of e in $\Phi_{\mathcal{T}}$.

For floating-point precisions this means that v is a finite value according to IEEE754 (i.e. either 0, subnormal or normal). For fixed-point precisions with word size w and f fractional bits, this means that v is within the range of representable values ($|v| \leq \frac{2^{w-1}-1}{2^f}$) and no overflow occurs (i.e. the fractional bits were correctly inferred).

FloVer uses Equation 3 to compute an error for floating-point precisions which is only valid in the presence of IEEE754 *normal* numbers or 0. We note that the roundoff error of the *biggest representable subnormal number* is smaller than the roundoff error of *normal numbers* in general. We add this condition as a check to function `validMachineRanges` by checking that the floating-point range contains at least one normal number.

d) Error Validator: If $\texttt{validErrors}(e, \Phi_{\mathcal{T}}, \Phi_{\mathcal{R}}, \Phi_{\mathcal{E}})$ succeeds, and e evaluates to v, then we want to show that $\tilde{e}$ evaluates to $\tilde{v}$, and that $|v - \tilde{v}| \leq \Phi_{\mathcal{E}}(e)$. The challenge in this proof lies in the fact that we reason about two different executions of similar expressions, e and $\tilde{e}$.

Given a free variable x in the analyzed expression e, the value $E(x)$ may not be representable as a finite-precision value. Thus the values for the related variables x and $\tilde{x}$ will not in general agree. This is the case for every free variable occurring in e. Additionally, the roundoff error of any variable depends on its precision. As a consequence we introduce an inductive approximation relation $\sim_{(\mathcal{V},\Phi_{\mathcal{T}})}$ between values provided by E and $\tilde{E}$ for variables in $\mathcal{V}$ so that we can prove the error bound. Given $E \sim_{(\mathcal{V},\Phi_{\mathcal{T}})} \tilde{E}$, both environments are defined for every variable $v \in \mathcal{V}$. In addition, the difference between $E(v)$ and $\tilde{E}(\tilde{v})$ at precision p is upper bounded by $\mathrm{error}(v, p)$, where p is $\Phi_{\mathcal{T}}(v)$. In the proofs we instantiate $\mathcal{V}$ by the free variables of the analyzed expression. Two empty environments are trivially related under the empty set ($(\_ \mapsto \bot) \sim_{(\emptyset,\,\Phi_{\mathcal{T}})} (\_ \mapsto \bot)$) and for free variables we have:

$$
\text{FreeVar}\ \frac{E \sim_{(\mathcal{V},\Phi_{\mathcal{T}})} \tilde{E} \qquad x \notin \mathcal{V} \qquad \Phi_{\mathcal{T}}(x) = m \qquad |\, v - \tilde{v}\,| \ \leq\ \mathrm{error}(v, m)}{(E\,[x \mapsto v]) \sim_{(\{x\}\cup\mathcal{V},\Phi_{\mathcal{T}})} (\tilde{E}\,[\tilde{x} \mapsto \tilde{v}])}
$$

To prove soundness for let-bindings, we will extend the relation with a rule for defined variables later.

$\Phi_{\mathcal{E}}$ maps expressions to rationals, representing absolute error bounds. FloVer computes error bounds from intervals from $\Phi_{\mathcal{R}}$ and the error bounds on subterms. The propagation errors for multiplication and division depend on both the real-valued and the float-valued ranges. Therefore the soundness proof requires solving 16 and 32 sub-cases for multiplication and division, respectively.

e) Let-Bindings: To extend the soundness proofs to let-bindings, we have to check that the analyzed function f is in SSA form (since $\Phi_{\mathcal{T}}$, $\Phi_{\mathcal{R}}$ and $\Phi_{\mathcal{E}}$ are maps, variables cannot be redefined). For this we use the formalization of SSA defined in the LVC framework [35]. Furthermore, we adapt the approximation relation $\sim$ to include let-bound variables:

$$
\text{DefinedVar}\ \frac{E \sim_{(\Phi_{\mathcal{E}},\mathcal{V},\mathcal{D},\Phi_{\mathcal{T}})} \tilde{E} \qquad x \notin \mathcal{V} \cup \mathcal{D} \qquad \Phi_{\mathcal{T}}(x) = m \qquad |\, v - \tilde{v}\,| \ \leq\ \Phi_{\mathcal{E}}(x)}{(E\,[x \mapsto v]) \sim_{(\Phi_{\mathcal{E}},\mathcal{V},\{x\}\cup\mathcal{D},\Phi_{\mathcal{T}})} (\tilde{E}\,[\tilde{x} \mapsto \tilde{v}])}
$$

Set $\mathcal{D}$, tracks variables added to both environments using let-bindings and $\Phi_{\mathcal{E}}$ is the error analysis result. The sets $\mathcal{D}$ and $\mathcal{V}$ are used to distinguish whether a variable x is free or let-bound.

f) Using Flover: We obtain the overall soundness of FloVer (Theorem 1) as the conjunction of the results of the functions `validTypes`, `validRealRange`, `validMachineRanges` and

`validErrors`. Theorem 1 holds only if checking of the certificate succeeds. If the static analysis result in a certificate is incorrect, e.g. a computed range or roundoff error is incorrect, FloVer fails checking the certificate. Our tool can be used by any other roundoff error analysis tool that computes real-valued ranges, roundoff error bounds and knows about variable types. Using FloVer is then as easy as implementing a pretty-printer for this information.

FloVer performs sound dataflow analysis, which necessarily computes an overapproximation of the true roundoff errors. It is thus possible that FloVer cannot verify a certificate even though the error bounds are indeed correct. Different range arithmetics, which influence the accuracy of FloVer's analysis, commit different overapproximations. Thus we use our implementations of IA and AA in Coq in a portfolio approach and run both when checking range analysis results.

A. Connecting FloVer to IEEE754

We connect our formalization to formalizations of IEEE754 floating-point arithmetic in HOL4 [16] and the Flocq library in Coq [5] by proving that if checking the certificate succeeds, we can evaluate the analyzed function using IEEE754 semantics and the roundoff error bound is valid for this execution.

Theorem 2. *Let $\tilde{f}$ be a function on 64-bit floating-points and f its real-valued counterpart, E a real-valued environment, $\tilde{E}$ its 64-bit floating-point counterpart, P a precondition constraining the free variables of f, Γ a map from all free variables of $\tilde{f}$ to 64-bit precision, $\Phi_{\mathcal{R}}$ a range analysis result, and $\Phi_{\mathcal{E}}$ an error analysis result, Then*

$$\begin{gathered} E \sim_{(\Phi_{\mathcal{E}},\mathcal{V},\mathcal{D},\Gamma)} \tilde{E} \wedge \\ \texttt{CertificateChecker}\,(f, P, \Gamma, \Phi_{\mathcal{R}}, \Phi_{\mathcal{E}}) \wedge \\ \texttt{IEEEevalAvoidsSubnormals}\,\tilde{f} \implies \\ \exists\, v\, \tilde{v}.\ (f, E) \Downarrow v \ \wedge\ (\tilde{f}, \tilde{E}) \Downarrow_{\mathit{IEEE}} \tilde{v} \ \wedge\ |v - \tilde{v}| \leq \Phi_{\mathcal{E}}(f) \end{gathered}$$

The proof of Theorem 2 is an extension of FloVer's soundness theorem (Theorem 1). To show that the roundoff error bounds are valid for the IEEE754 operations, we use the soundness theorem of `validMachineRanges` to establish that all values obtained form an evaluation are finite.

The formalization in HOL4 (currently) does support neither cast operations nor reasoning about roundoff errors for subnormal values. Until these are supported, we assume Γ to map every variable to 64-bit double precision and disallow subnormal values to occur during evaluation. To this end, we define the function $\texttt{IEEEevalAvoidsSubnormals}(e, E)$, as a temporary workaround. The function returns true only if every subexpression of e evaluates to a normal value or 0.

B. Division Bug Found

We use Daisy [12] to generate certificates for our evaluation. During this, we found a subtle bug in the tool's static analysis of the division operator. The error bounds are only sound in the absence of division-by-zero errors, but only the real-valued range of the denominator was checked for whether it contains zero. It is possible, however, that the real-valued range does *not* contain zero, while the corresponding floating-point range does, essentially due to large enough roundoff errors.

C. Formalization Details

Executions inside FloVer are represented in both Coq and HOL4 as big-step relations using Equation 2. These formalizations do not depend on external libraries. Only the connection to IEEE754 semantics uses external libraries.

Roundoff errors and our theorems relate real-valued executions to finite-precision ones and we thus need a way to represent the numbers and also compute on them. However, the latter is problematic for infinite-precision reals. We use rationals to represent the values in the certificates. To relate these values to the real-valued ($\mathbb{R}$) executions in the theorem statement, we use the fact that rationals are a subset of the `real` type in HOL4, and in Coq we use the translation $\texttt{Q2R}{:}\mathbb{Q} \to \mathbb{R}$ and exploit that our AST is parametric in the constant type by instantiating it with $\mathbb{Q}$ for computations and $\mathbb{R}$ for theorems.

VI. Extracting a Verified Binary with CakeML

Running the range and error checker functions in Coq and HOL4 directly is quite inefficient (see our experiments in Section VII). We have thus extracted a verified binary from our HOL4 checker function definitions, and an unverified binary for Coq. We are aware of the work on certified extraction from Coq in the CertiCoq [2] project, but at the time of writing, the tool could not handle our checker definitions.

We have implemented in HOL4 and Coq an unverified lexer and parser for the encoding of the certificates, which are included in the extracted binaries in both Coq and HOL4.

a) Extracting from HOL4: For extracting a binary from HOL4, we use the CakeML proof-producing synthesis tool [33] which translates ML-like HOL4 functions into deeply embedded CakeML programs that exhibit the same behaviour. In HOL4 we use the `real` type to store the rational bounds in $\Phi_{\mathcal{R}}$ and $\Phi_{\mathcal{E}}$. For each of the arithmetic operations over the `real` type that we used in the HOL4 development, we define a translation into a representation of the arbitrary-precision rationals in CakeML.

CakeML and HOL4 have different notions of equality. Since we perform equality tests in the certificate checkers, we had to prove that our newly defined representation of real numbers respects CakeML's semantics for structural equality. For this purpose, we had to require and prove that our representation of rationals maintains a gcd of one between nominator and denominator.

When translating a HOL4 function into CakeML code, the CakeML toolchain generates preconditions that exclude runtime exceptions, e.g. divisions by zero. We have shown that all generated preconditions are always satisfied, hence the specification theorem for the generated ML code does not have any unproved preconditions left.

Having compiled the CakeML libraries beforehand, we can compile the checking functions into a verified binary in around 90 minutes on the same machine as we used for the experiments in Section VII. Checking the certificate with the

Benchmark	FloVer		FPTaylor
	interval	affine	
ballbeam	2.141e-12	2.141e-12	1.746e-12
bspline1	1.517e-15	1.601e-15	5.149e-16
bspline2	1.406e-15	1.448e-15	5.431e-16
bspline3	1.295e-16	1.295e-16	8.327e-17
doppler (m)	9.766e-05	7.445e-04	3.111e-05
floudas1	1.052e-12	1.074e-12	5.755e-13
floudas26	7.292e-13	7.292e-13	7.740e-13
floudas33	3.109e-15	3.109e-15	6.199e-13
himmilbeau (m)	4.876e-04	4.876e-04	3.641e-04
invertedPendulum	5.369e-14	5.369e-14	3.843e-14
kepler0 (m)	2.948e-05	2.948e-05	1.758e-05
kepler1 (m)	9.948e-05	9.948e-05	5.902e-05
kepler2 (m)	3.732e-04	3.732e-04	1.433e-04
rigidBody1 (m)	4.023e-05	4.023e-05	2.146e-05
rigidBody2 (m)	6.438e-03	6.438e-03	9.871e-03
traincar1-out1	5.406e-12	5.406e-12	4.601e-12
traincar1-state1	5.421e-15	5.421e-15	4.753e-15
traincar1-state2	8.862e-15	8.862e-15	8.099e-15
traincar1-state3	7.784e-15	7.784e-15	7.013e-15
turbine1 (m)	1.356e-05	1.356e-05	3.192e-06
turbine2 (m)	2.034e-05	2.034e-05	4.970e-06
turbine3 (m)	9.038e-06	9.038e-06	1.671e-06

TABLE I
ROUNDOFF ERRORS VERIFIED BY FLOVER AND FPTAYLOR.

Benchmark	# ops	Daisy	Coq		HOL4	CakeML	OCaml
			Interval	Affine			
ballBeam	7	4.62	3.50	3.26	89.04	<0.01	0.02
invertedPendulum	7	3.62	3.59	3.27	112.61	0.01	0.02
bicycle	13	4.31	4.01	4.08	156.76	0.01	0.04
doppler (m)	17	4.86	5.28	12.21	610.67	0.05	0.02
dcMotor	26	5.19	4.97	4.50	316.75	0.02	0.08
himmilbeau (m)	26	3.52	4.11	4.40	65.48	0.02	0.03
bspline	28	4.21	4.61	4.07	298.44	0.03	0.08
rigidbody (m)	33	5.04	7.14	4.52	88.92	0.03	0.06
science	35	5.64	11.69	567.36	1471.96	0.07	0.07
traincar1	36	4.85	10.87	9.84	932.93	0.07	0.11
batchProcessor	56	6.46	8.49	7.43	997.77	0.06	0.16
batchReactor	58	6.84	11.45	9.53	1117.48	0.07	0.17
turbine (m)	82	5.99	18.69	24.90	4095.56	0.25	0.11
traincar2	89	7.90	29.79	28.58	3967.88	0.23	0.27
floudas	99	7.76	13.99	12.76	565.68	0.14	0.27
kepler (m)	158	4.89	21.56	22.70	3848.75	0.21	0.21
traincar3	168	9.14	68.53	68.14	9594.07	0.58	0.49
traincar4	269	10.6	116.94	115.38	17429.3	1.10	0.77

TABLE II
RUNNING TIMES OF DAISY AND FLOVER IN SECONDS.

binary is then extremely fast, since no theorem prover logic is loaded.

b) Extracting from Coq: Coq natively supports unverified extraction into OCaml code [27]. We used the existing libraries for translating Coq numbers into OCaml's `Big_int` type from the base library. The extracted code is compiled using the OCaml native-code compiler ("ocamlopt") in our experiments.

VII. EVALUATION

To evaluate the performance of FloVer, we have extended the static analyzer Daisy to generate certificates of its analysis. As Daisy already computes all the information that needs to be encoded in a certificate, implementing the certificate generation was similar to implementing a pretty-printer for analysis results (we have switched off a few optimizations, which however do not affect the error bounds significantly). Using the certificate generation, we have evaluated Daisy and FloVer on examples taken from the Rosa [9] and real2float [29] projects. Each benchmark consists of one or more separate functions. Daisy analyzes all functions of one benchmark together and produces one certificate containing a call to the certificate checker for each separate function.

We compare error bounds verified by FloVer with those verified by FPTaylor, as FPTaylor generally computes the most accurate bounds [37, 11]. Furthermore, Rosa [9], Fluctuat [18] and Gappa [13] use the same technique to compute roundoff errors as Daisy and FloVer. We also compare FloVer's certificate checking times with FPTaylor's, as the tool also provides a proof certificate.We note that FPTaylor can compute less precise error bounds with shorter running times, here we opt for the off-the-shelve solution without additional parameters.

a) Accuracy: Table I gives a subset of the roundoff errors certified by FloVer as well as roundoff errors computed by FPTaylor for comparison (we give the full table in the appendix (Section A).PRECiSa and Gappa compute similar results; we provide them here for two benchmarks for reference. For the ballBeam benchmark, Precisa and Gappa show an error of 1.085e-07 and 1.240e-12 resp., and for the invertedPendulum benchmark, the errors are 3.531e-12 and 3.217e-14. The focus of FloVer is not to compute the most precise bounds possible, but rather to develop the necessary infrastructure for future extensions. Nevertheless, the roundoff errors verified by it are usually close to those proven by FPTaylor. Benchmarks marked with '(m)' are in mixed-precision, otherwise the roundoff errors are evaluated under uniform double (64 bit) floating-point precision (FPTaylor does not support fixed-point precision).

b) Efficiency: In Table II, we compare running times of in-logic evaluation of FloVer in Coq and HOL4, the *verified* binary extracted with the CakeML toolchain and the *unverified* binary extracted from Coq. For our experiments we used a machine with a four core Intel i3 processor with 3.3GHz, 8 GB of RAM, running Debian 9. For the in-logic evaluation in Coq we show range analysis in interval and affine arithmetic, for all other runs we use interval arithmetic. As for the accuracy evaluation, benchmarks marked with '(m)' are in mixed-precision, double precision otherwise.

In Table II, 'OCaml' refers to the Coq binary compiled with the OCaml native compiler. The '# ops' column gives the number of arithmetic operations in the whole benchmark (summed for all functions) and gives an intuition about the complexity of the benchmark. For all columns, the running times are the end-to-end times measured by the UNIX *time*

command in seconds. This time includes parsing and generating the certificate for Daisy, checking the proof that FloVer succeeds for Coq and HOL4 in-logic, and running FloVer in the binaries. The running times for Daisy, Coq and HOL4 are the average running times for a single run over three runs. For the binaries we report the average running time of a single run from 300 executions (due to the small runtime).

We give the running times for FPTaylor's certificate checking in the appendix (Section A)and note that they are larger, but of the same order of magnitude as our Coq in-logic evaluation. Note that FPTaylor's checker requires either a two hour starting time or external checkpointing. FloVer's certificate checking time for fixed-point arithmetic is similar to floating-point checking; we give the detailed running times in the appendix (Section A).

The evaluation of FloVer's Coq checker is faster than the evaluation of the HOL4 checker. This is probably because we benefit from Coq's `vm_compute` tactic in the Coq evaluation. The tactic translates terms to OCaml and evaluates them using a virtual machine. A Coq term is reconstructed from the result. HOL4's `EVAL_TAC` instead uses a simple call-by-value evaluation strategy. We further observe that the evaluation using affine arithmetic sometimes is as fast as the one using intervals. We suspect that the reason for this is that the affine arithmetic checker must memorize polynomials for sub-expressions and thus does not recompute them. The interval validator, however, currently does not memorize sub-expressions, but only let-bound variables.

VIII. Related Work

a) Sound Accuracy Analysis: The tools FPTaylor [37], Gappa [13], PRECiSa [31], real2float [29] and VCFloat [34] are most closely related to our work as they formally verify floating-point roundoff errors. Each tool handles mixed-precision floating-point arithmetic, but other features differ slightly between tools. FloVer is the only tool with the combination of support for both Coq and HOL4, floating-point as well as fixed-point arithmetic and two abstract domains, interval and affine arithmetic. FloVer is fully automated and FloVer and FPTaylor are the only tools that generate certificates using in-logic decision procedures While FPTaylor and PRECiSa handle transcendental functions (which FloVer does not), both tools do not handle fixed-point arithmetic. Gappa has some support for fixed-points, but FloVer is the only tool with formalized affine arithmetic. Finally, FloVer is the first tool to provide *efficient* certificate checking with a verified binary. Fluctuat [17], Gappa++ [28] and Rosa [11] statically bound finite-precision roundoff errors using affine arithmetic [15], but do not provide formal guarantees.

FloVer currently does not handle conditionals and loops. These are—to some extent—supported by Fluctuat [18] and Rosa [11], however not formally verified. PRECiSa [31] provides an initial formalization of these approaches, but scalability is unclear [9, 11]. FloVer furthermore focuses, like most tools, on certifying absolute error bounds. Bounding relative errors is challenging due to the increased complexity as well as due to the issue that often the error is not even well-defined due to an inherent division by zero [22]. Gappa does provide verified relative error support by optimizing a constraint based on Equation 3. This approach has been shown to not provide tight bounds once input ranges and expressions become larger [22]. Finally, note that input ranges are also necessary for computing *concrete* relative error bounds.

b) Sound Verification of Floating-point Computations: Absence of runtime errors in floating-point computations can be shown with abstract interpretation, where different abstract domains have been developed for this purpose [4, 8, 24], which are sound w.r.t. floating-point arithmetic. Jourdan et al. [25] have also formalized some of these abstract domains in Coq. Note, however, that these domains do not quantify the difference between a real-valued and the finite-precision semantics and can only show the absence of runtime errors.

Moscato et al. [32] have built a formalization and implementation of AA for computation of real-valued ranges in PVS. This development does not handle division, which we do. Immler [21] has formalized AA in Isabelle/HOL; our own formalization shares a similar structure.

Coq has also been used to prove entire programs correct w.r.t. numerical uncertainties such as roundoff errors [6]. However, in these efforts much of the work is still manual. Our current development can be seen as complementary as it could potentially provide automation for the verification of roundoff error bounds. The CompCert compiler also supports floating-point computations [7], but only shows semantics preservation and not roundoff error bounds. Harrison [19] has formally verified a floating point implementation of the exponential function inside HOL-Light. The analysis is detailed and specific to this particular function. In contrast, our work aims to provide a fully automated verified analysis for arbitrary real-valued expressions, but at a higher level of abstraction.

c) Real Arithmetic and Finite-precision Formalizations: Formalizations of floating-point arithmetic exist in HOL-Light [23], in Coq in the Flocq library [5] as well as in Isabelle [39] and HOL4 [16]. We found using these formalizations in Coq and HOL4 more complex than was necessary for reasoning inside FloVer, thus we use them only to show a connection to IEEE754. Fixed-point arithmetic has been formalized in HOL4 [1], focusing on its hardware implementation, whereas our focus is on relating their execution to real-valued semantics.

IX. Conclusion

We have presented our modular, reusable and easily extendable approach to certificate checking for error bound analysis in FloVer. Our checker is fully-automated and requires neither user interaction, nor expert knowledge. All of the theorems about FloVer have been proven in both Coq and HOL4. We are the first to extract a verified binary for checking finite-precision roundoff errors using the CakeML toolchain and have shown that we achieve significant performance improvements when using the binary.

References

[1] B. Akbarpour, S. Tahar, and A. Dekdouk. Formalization of Fixed-Point Arithmetic in HOL. *Formal Methods in System Design*, 27(1-2):173–200, 2005.

[2] A. Anand, A. Appel, G. Morrisett, Z. Paraskevopoulou, R. Pollack, O. S. Belanger, M. Sozeau, and M. Weaver. CertiCoq: A Verified Compiler for Coq. In *Coq for Programming Languages (CoqPL)*, 2017.

[3] A. Anta, R. Majumdar, I. Saha, and P. Tabuada. Automatic Verification of Control System Implementations. In *EMSOFT*, 2010.

[4] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A Static Analyzer for Large Safety-Critical Software. In *Programming Language Design and Implementation (PLDI)*, 2003.

[5] S. Boldo and G. Melquiond. Flocq: A Unified Library for Proving Floating-Point Algorithms in Coq. In *IEEE Symposium on Computer Arithmetic (ARITH)*, 2011.

[6] S. Boldo, F. Clément, J.-C. Filliâtre, M. Mayero, G. Melquiond, and P. Weis. Wave Equation Numerical Resolution: A Comprehensive Mechanized Proof of a C Program. *Journal of Automated Reasoning*, 50(4):423–456, 2013.

[7] S. Boldo, J.-H. Jourdan, X. Leroy, and G. Melquiond. Verified Compilation of Floating-Point Computations. *Journal of Automated Reasoning*, 54(2):135–163, 2015.

[8] L. Chen, A. Miné, and P. Cousot. A Sound Floating-Point Polyhedra Abstract Domain. In *Asian Symposium on Programming Languages and Systems (APLAS)*, 2008.

[9] E. Darulova. Rosa - The real compiler. https://github.com/malyzajko/rosa, 2015.

[10] E. Darulova and V. Kuncak. Sound Compilation of Reals. In *Symposium on Principles of Programming Languages (POPL)*, 2014.

[11] E. Darulova and V. Kuncak. Towards a Compiler for Reals. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 39(2), 2017.

[12] E. Darulova, A. Izycheva, F. Nasir, F. Ritter, H. Becker, and R. Bastian. Daisy-Framework for Analysis and Optimization of Numerical Programs (Tool Paper). In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 270–287. Springer, 2018.

[13] M. Daumas and G. Melquiond. Certification of Bounds on Expressions Involving Rounded Operators. *ACM Trans. Math. Softw.*, 37(1):2:1–2:20, 2010.

[14] F. De Dinechin, C. Q. Lauter, and G. Melquiond. Assisted Verification of Elementary Functions using Gappa. In *ACM Symposium on Applied Computing (SAC)*, 2006.

[15] L. H. de Figueiredo and J. Stolfi. Affine Arithmetic: Concepts and Applications. *Numerical Algorithms*, 37 (1-4), 2004.

[16] A. Fox, J. Harrison, and B. Akbarpour. A Formal Model of IEEE Floating Point Arithmetic. *HOL4 Theorem Prover Library*, Apr. 2017. https://github.com/HOL-Theorem-Prover/HOL/tree/master/src/floating-point.

[17] E. Goubault and S. Putot. Static Analysis of Finite Precision Computations. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2011.

[18] E. Goubault and S. Putot. Robustness Analysis of Finite Precision Implementations. In *Asian Symposium on Programming Languages and Systems (APLAS)*, 2013.

[19] J. Harrison. Floating Point Verification in HOL Light: The Exponential Function. *Form. Methods Syst. Des.*, 16 (3), 2000.

[20] C. S. IEEE. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, 2008.

[21] F. Immler. A Verified Algorithm for Geometric Zonotope/Hyperplane Intersection. In *Proceedings of the 2015 Conference on Certified Programs and Proofs*, pages 129–136. ACM, 2015.

[22] A. Izycheva and E. Darulova. On Sound Relative Error Bounds for Floating-Point Arithmetic. In *Formal Methods in Computer-Aided Design (FMCAD)*, 2017.

[23] C. Jacobsen, A. Solovyev, and G. Gopalakrishnan. A Parameterized Floating-Point Formalizaton in HOL Light. *Electronic Notes in Theoretical Computer Science*, 317: 101–107, 2015.

[24] B. Jeannet and A. Miné. Apron: A Library of Numerical Abstract Domains for Static Analysis. In *Computer Aided Verification (CAV)*, 2009.

[25] J.-H. Jourdan, V. Laporte, S. Blazy, X. Leroy, and D. Pichardie. A Formally-Verified C Static Analyzer. In *Symposium on Principles of Programming Languages (POPL)*, 2015.

[26] X. Leroy. Formal Verification of a Realistic Compiler. *Communications of the ACM*, 52(7), 2009.

[27] P. Letouzey. A New Extraction for Coq. In *International Workshop on Types for Proofs and Programs (TYPES)*, 2002.

[28] M. D. Linderman, M. Ho, D. L. Dill, T. H. Meng, and G. P. Nolan. Towards Program Optimization through Automated Analysis of Numerical Precision. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2010.

[29] V. Magron, G. A. Constantinides, and A. F. Donaldson. Certified Roundoff Error Bounds Using Semidefinite Programming. *ACM Transactions on Mathematical Software (TOMS)*, 43(4):34, 2017.

[30] R. Moore. *Interval Analysis*. Prentice-Hall, 1966.

[31] M. Moscato, L. Titolo, A. Dutle, and C. A. Munoz. Automatic Estimation of Verified Floating-Point Round-Off Errors via Static Analysis. In *International Conference on Computer Safety, Reliability, and Security*, pages 213–229. Springer, 2017.

[32] M. M. Moscato, C. A. Muñoz, and A. P. Smith. Affine Arithmetic and Applications to Real-Number Proving. In *International Conference on Interactive Theorem Proving (ITP)*, 2015.

[33] M. O. Myreen and S. Owens. Proof-Producing Synthesis of ML from Higher-Order Logic. In *International Conference on Functional Programming (ICFP)*, 2012.

[34] T. Ramananandro, P. Mountcastle, B. Meister, and R. Lethin. A Unified Coq Framework for Verifying C Programs with Floating-Point Computations. In *Certified Programs and Proofs (CPP)*, 2016.

[35] S. Schneider, G. Smolka, and S. Hack. A Linear First-Order Functional Intermediate Language for Verified Compilers. In *International Conference on Interactive Theorem Proving (ITP)*, 2015.

[36] A. Solovyev and T. C. Hales. Formal Verification of Nonlinear Inequalities with Taylor Interval Approximations. In *NFM*, 2013.

[37] A. Solovyev, C. Jacobsen, Z. Rakamaric, and G. Gopalakrishnan. Rigorous Estimation of Floating-Point Round-off Errors with Symbolic Taylor Expansions. In *International Symposium on Formal Methods (FM)*, 2015.

[38] Y. K. Tan, M. O. Myreen, R. Kumar, A. Fox, S. Owens, and M. Norrish. A New Verified Compiler Backend for CakeML. In *International Conference on Functional Programming (ICFP)*, 2016.

[39] L. Yu. A Formal Model of IEEE Floating Point Arithmetic. *Archive of Formal Proofs*, July 2013. ISSN 2150-914x. http://isa-afp.org/entries/IEEE_Floating_Point.shtml.

APPENDIX

TABLE III. Full table with all the roundoff errors verified by FloVer and computed by FPTaylor from our evaluation in Section VII.

Benchmark	FloVer		FPTaylor
	interval	affine	
ballbeam	2.141e-12	2.141e-12	1.746e-12
batchProcessor-out1	1.187e-14	1.187e-14	1.048e-14
batchProcessor-out2	3.187e-14	3.187e-14	2.911e-14
batchProcessor-state1	7.902e-15	7.902e-15	7.132e-15
batchProcessor-state2	8.521e-15	8.521e-15	7.840e-15
batchProcessor-state3	8.405e-15	8.405e-15	7.689e-15
batchProcessor-state4	7.008e-15	7.008e-15	6.217e-15
batchReactor-out1	1.212e-15	1.212e-15	9.938e-16
batchReactor-out2	3.141e-15	3.141e-15	2.692e-15
batchReactor-state1	9.118e-16	9.118e-16	8.006e-16
batchReactor-state2	8.451e-16	8.451e-16	7.455e-16
batchReactor-state3	8.591e-16	8.591e-16	7.462e-16
batchReactor-state4	7.693e-16	7.693e-16	7.019e-16
bicycle-out1	6.943e-15	6.943e-15	5.450e-15
bicycle-state1	5.927e-16	5.927e-16	4.778e-16
bicycle-state2	4.815e-16	4.815e-16	3.995e-16
bspline0	2.405e-16	2.405e-16	1.388e-16
bspline1	1.517e-15	1.601e-15	5.149e-16
bspline2	1.406e-15	1.448e-15	5.431e-16
bspline3	1.295e-16	1.295e-16	8.327e-17
dcMotor-out1	7.851e-17	7.851e-17	6.419e-17
dcMotor-state1	6.58e-16	6.58e-16	5.504e-16
dcMotor-state2	1.172e-15	1.172e-15	9.978e-16
dcMotor-state3	2.271e-17	2.271e-17	1.969e-17
doppler (m)	9.766e-05	7.445e-04	3.111e-05
floudas1	7.292e-13	7.292e-13	5.755e-13
floudas26	1.052e-12	1.074e-12	7.740e-13
floudas33	7.292e-13	7.292e-13	6.199e-13
floudas34	3.109e-15	3.109e-15	2.220e-15
floudas46	1.554e-15	1.554e-15	1.554e-15
floudas47	2.798e-14	2.848e-14	1.665e-14
himmilbeau (m)	4.876e-04	4.876e-04	3.641e-04
invertedPendulum	5.369e-14	5.369e-14	3.843e-14
kepler0 (m)	2.948e-05	2.948e-05	1.758e-05
kepler1 (m)	9.948e-05	9.948e-05	5.902e-05
kepler2 (m)	3.732e-04	3.732e-04	1.433e-04
rigidBody1 (m)	4.023e-05	4.023e-05	2.146e-05
rigidBody2 (m)	1.288e-02	1.288e-02	9.871e-03
verhulst	8.343e-16	8.343e-16	3.235e-16
predatorPrey	3.395e-16	3.468e-16	1.836e-16
carbonGas	5.688e-08	5.666e-08	9.129e-09
traincar1-out1	5.406e-12	5.406e-12	4.601e-12
traincar1-state1	5.421e-15	5.421e-15	4.753e-15
traincar1-state2	8.862e-15	8.862e-15	8.099e-15
traincar1-state3	7.784e-15	7.784e-15	7.013e-15
traincar2-out1	3.96e-12	3.96e-12	2.734e-12
traincar2-state1	1.104e-14	1.104e-14	1.042e-14
traincar2-state2	1.103e-14	1.103e-14	1.041e-14
traincar2-state3	1.11e-14	1.11e-14	1.033e-14
traincar2-state4	9.992e-15	9.992e-15	9.215e-15
traincar2-state5	1.788e-14	1.788e-14	1.699e-14
traincar3-out1	5.44e-11	5.44e-11	4.396e-11
traincar3-state1	8.033e-15	8.033e-15	7.610e-15
traincar3-state2	7.91e-15	7.91e-15	7.790e-15
traincar3-state3	7.595e-15	7.595e-15	7.156e-15
traincar3-state4	1.469e-14	1.469e-14	1.375e-14
traincar3-state5	1.344e-14	1.344e-14	1.249e-14
traincar3-state6	1.222e-14	1.222e-14	1.127e-14
traincar3-state7	1.099e-14	1.099e-14	1.005e-14
traincar4-out1	6.269e-10	6.269e-10	4.374e-10
traincar4-state1	1.083e-14	1.083e-14	1.055e-14
traincar4-state2	1.227e-14	1.227e-14	1.055e-14
traincar4-state3	1.155e-14	1.155e-14	9.828e-15
traincar4-state4	1.083e-14	1.083e-14	9.106e-15
traincar4-state5	1.916e-14	1.916e-14	1.797e-14
traincar4-state6	1.732e-14	1.732e-14	1.621e-14
traincar4-state7	1.599e-14	1.599e-14	1.488e-14
traincar4-state8	1.466e-14	1.466e-14	1.355e-14
traincar4-state9	1.332e-14	1.332e-14	1.221e-14
turbine1 (m)	1.356e-05	1.356e-05	3.192e-06
turbine2 (m)	2.034e-05	2.034e-05	4.970e-06
turbine3 (m)	9.038e-06	9.038e-06	1.671e-06

TABLE IV. This table gives FPTaylor's running times measured by the ocaml 'time' command in seconds. The numbers are measured *after* checkpointing a HOL-Light state with the FPTaylor solver loaded and thus include only certificate checking times. Each number is the average running time of a single run measured over three runs. Since FPTaylor produces separate certificates for each function in the input file, we report them separately. 'Error' refers to FPTaylor's checker failing with the exception `Invalid_argument "index out of bounds"`. For comparison, we give the Coq in-logic checking times for all functions in the benchmark file, as they are closest to those of FPTaylor.

Benchmark	FPTaylor	Coq In-logic
ballbeam	12.37	3.50
batchProcessor-out1	19.54	7.28
batchProcessor-out2	19.51	
batchProcessor-state1	27.33	
batchProcessor-state2	37.84	
batchProcessor-state3	37.33	
batchProcessor-state4	37.74	
batchReactor-out1	19.11	12.03
batchReactor-out2	18.82	
batchReactor-state1	37.94	
batchReactor-state2	38.69	
batchReactor-state3	38.73	
batchReactor-state4	38.33	
bicycle-out1	6.37	4.03
bicycle-state1	11	
bicycle-state2	10.92	
bspline0	10.77	4.46
bspline1	15.67	
bspline2	19.26	
bspline3	6.38	
dcMotor-out1	11.56	5.13
dcMotor-state1	17.82	
dcMotor-state2	17.57	
dcMotor-state3	17.44	
doppler (m)	231.37	6.10
floudas1	63.2	13.16
floudas26	Error	
floudas33	71.15	
floudas34	4.85	
floudas46	2.68	
floudas47	7.9	
himmilbeau (m)	49.95	4.92
invertedPendulum	15.91	3.64
kepler0 (m)	70.42	20.51
kepler1 (m)	147.18	
kepler2 (m)	412.96	
rigidBody1 (m)	14.18	4.90
rigidBody2 (m)	47.33	
verhulst	12.21	9.70
predatorPrey	28.99	
carbonGas	28.21	
traincar1-out1	15.31	10.25
traincar1-state1	46.64	
traincar1-state2	45.59	
traincar1-state3	47.57	
traincar2-out1	31.94	28.92
traincar2-state1	81.43	
traincar2-state2	84.77	
traincar2-state3	86.59	
traincar2-state4	81.17	
traincar2-state5	74.61	
traincar3-out1	Error	66.90
traincar3-state1	Error	
traincar3-state2	Error	
traincar3-state3	Error	
traincar3-state4	Error	
traincar3-state5	Error	
traincar3-state6	Error	
traincar3-state7	Error	
traincar4-out1	Error	112.95
traincar4-state1	Error	
traincar4-state2	Error	
traincar4-state3	Error	
traincar4-state4	Error	
traincar4-state5	Error	
traincar4-state6	Error	
traincar4-state7	Error	
traincar4-state8	Error	
traincar4-state9	Error	
turbine1 (m)	80.96	18.38
turbine2 (m)	56.41	
turbine3 (m)	82.86	

TABLE V

THIS TABLE SHOWS OUR EVALUATION OF FLOVER ON FIXED-POINT ARITHMETIC. FOR THIS WE USED DAISY'S FIXED-POINT ANALYSIS, TO GENERATE A CERTIFICATE FOR A WORD LENGTH OF 32 BITS. '# OPS' IS THE NUMBERS OF ARITHMETIC OPERATIONS IN THE FILE WHICH WE USE AS OUR COMPLEXITY MEASURE. AS FOR OUR EVALUATION IN SECTION VII ALL MEASUREMENTS ARE ELAPSED TIME FROM THE UNIX *time* COMMAND, IN SECONDS.

Benchmark	# ops	Daisy	Coq (IA)	HOL4	CakeML	OCaml
ballBeam	7	3.25	3.07	51.03	<0.01	<0.01
invertedPendulum	7	3.17	3.03	57.55	<0.01	<0.01
bicycle	13	3.48	3.27	73.98	<0.01	<0.01
doppler	17	3.14	3.21	73.14	<0.01	<0.01
dcMotor	26	3.81	3.86	94.69	<0.01	0.01
himmilbeau	26	2.99	3.11	50.30	<0.01	<0.01
bspline	28	3.59	3.50	74.84	<0.01	0.01
rigidbody	33	3.32	3.36	58.80	<0.01	0.01
science	35	3.54	4.11	180.2	0.01	<0.01
traincar1	36	4.11	6.48	252.19	0.01	0.02
batchProcessor	56	4.55	4.67	153.17	0.01	0.02
batchReactor	58	4.61	5.39	189.14	0.01	0.02
turbine	82	3.88	5.20	325.77	0.03	0.01
traincar2	89	5.36	14.37	622.37	0.03	0.04
floudas	99	5.54	6.27	125.58	0.01	0.04
kepler	158	3.92	6.24	216.79	0.01	0.01
traincar3	168	6.47	31.01	1244.96	0.07	0.10
traincar4	269	7.72	45.08	1479.48	0.09	0.18