Max Planck Institute for Security and Privacy, Bochum, Germany

Formal Verification of Emulated Floating-Point Arithmetic in Falcon

Vincent Hwang

September 19, 2024

Goals

- Incorrect zeroization.
- Range checking in FFT (formal verification, for the absence of incorrect zeroizations).
- Equivalences between fadd./fsub./fmul. implementations (formal verification).

Falcon

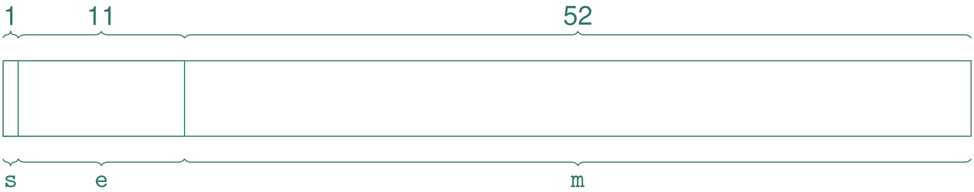
Falcon

- Lattice-based digital signature selected by NIST.
- Compact signature and public key sizes (compared to another lattice-based winner Dilithium).
- Floating-point arithmetic in signing.
  - FFT
  - Falcon tree.
  - Fast Fourier sampling.

Incorrect Zeroization in the Emulated Floating-Point Multiplication

Floating-Point Arithmetic

Double-precision in this talk.



- $0 < \mathtt{e} < 2047$ (normal values):

$$(-1)^{\mathtt{s}}\, 2^{\mathtt{e}-1075} \left(2^{52}+\mathtt{m}\right).$$

- Zeros: $\mathtt{e} = \mathtt{m} = 0$.
- Other values (irrelevant in this work):
  - $\mathtt{e} = 0$ and $\mathtt{m} \neq 0$: subnormals.
  - $\mathtt{e} = 2047$ and $\mathtt{m} = 0$: infinites.
  - $\mathtt{e} = 2047$ and $\mathtt{m} \neq 0$: NaNs.

Concerns of Floating-Point Arithmetic

- Not always constant-time.
- FPU does not even exist on some microcontrollers!

Emulating with Integer and Bit-Wise Arithmetic

- Compute with only
  - Integer: add./sub./mul.
  - Bit-wise: $|, \&, \hat{}, \sim$.
- No secret-dependent branches.
- Conditional computations are implemented as
  1. computations for all the branches;
  2. computations for the selection criteria; and
  3. selections of the desired results with bit-wise arithmetic.

Floating-Point Multiplication

- Assume no exceptions: users' responsibilities.
- Compute the results
    - as if an input is a zero;
    - as if both are non-zero, and apply rounding.
- Compute selection criterion from the inputs.
- Select the desired one.

Incorrect Zeroization (Simplified View)

For simplicity, assume inputs are non-zero floats: `s0|e0|m0` and `s1|e1|m1`.

1. Compute the integer product $\left(2^{52} + \texttt{m0}\right)\left(2^{52} + \texttt{m1}\right)$.
2. Normalize to a 55-bit integer $\texttt{zu}$ (round-mode-dependent).
3. Compute the sum $\texttt{e}$ of exponents.
4. Round.
5. Zeroize if too small.

Incorrect zeroization.

- Falcon submission package:
  1. Zeroize if $(\texttt{e}, \texttt{zu})$ is too small ($< 2^{-1023}$).
  2. Round.
- Issue:
  - Rounding could increase $(\texttt{e}, \texttt{zu})$.
  - If both the following hold, the result is incorrectly zeroized.
    - $(\texttt{e}, \texttt{zu})$ is too small $\implies 0$.
    - $(\texttt{e}, \texttt{zu})$ is sufficiently large after rounding $\implies$ non-zero.

Does it matter?

Falcon FFT

Twiddle factors are stored as floats. For example:

- $\frac{1}{\sqrt{2}}$ is stored as `0|1022|1865452045155277` $\implies$ $\exists$ a float whose product is incorrectly zeroized.
- $692/2048\ (34\%)$ float constants in FFT admit such floats!

Range Checking

Falcon FFT in Signature Generation

- Question: Are non-zero floats even close to ± 0?
- The FFT is applied to poly. with integer coeff. drawn from $[-2^{15}, 2^{15})$.

Interval Arithmetic

- Floating-point add./sub./mul.
- For non-zeros:
  - Upper-bound of the abs.
  - Lower-bound of the abs. $\Longrightarrow$ Tell us the smallest value (in abs.).

Range Checking

1. Model floating-point add/sub/mul with CryptoLine.
2. Compute intervals of intermediate floats with interval arithmetic built upon native FPU.
3. Verify the correctness of input-output intervals w.r.t. CryptoLine modeling.
4. Determine the union of intervals.

If inputs of FFT are integers drawn from $\left[-2^{15}, 2^{15}\right)$, then all the intermediate floats have abs. in

$$\left[2^{-476}, 2^{27}(2^{52} + 605182448294568)\right] .$$

Far away from the smallest (positive) normal value $2^{-1023} \implies \neg\exists$ incorrect zeroization.

Verification Time

Operation	Number of instances	Verification time (avr. / total in seconds)
FP addition	767	0.297 886 / 228.478 732
FP multiplication	511	2.589 009 / 1 322.983 371

Equivalence Proofs

Equivalences Between Implementations

Floating-point add./sub./mul. implementations.

Opt

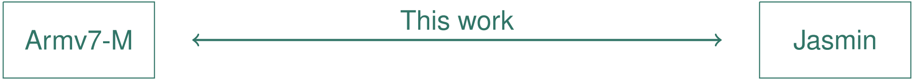
Ref

assembly
crazy opt

intuitive

Equivalences Between Implementations in This Work

Floating-point add./sub./mul. implementations.



Armv7-M

This work

Jasmin

assembly
crazy opt

intuitive

Equivalences via CryptoLine Modeling

Floating-point add./sub./mul. implementations.

Armv7-M ⟷ CryptoLine ⟷ Jasmin

assembly
crazy opt

intuitive

Verification Time

Programming langauge	Verification time (in seconds)
Floating-point addition	
Jasmin	53.946 560
Assembly	59.863 976
Floating-point multiplication	
Jasmin	57.108 668
Assembly	5.333 913

Future Work

- More rounding modes (straightforward).
- More floating-point arithmetic:
  - Halving, doubling, flooring, truncating, all straightforward.
  - Floating-point divisions, no obvious difficulties but a lot more instructions.
- More float-based operations:
  - Falcon tree (requires floating-point divisions).
  - Fast Fourier sampling (while-loop, other tooling required).
- More schemes:
  - ModFalcon, Mitaka (hybrid sampler).

Thanks for listening

Paper (IACR ePrint): `https://eprint.iacr.org/2024/321`

Paper (proceedings):
`https://link.springer.com/chapter/10.1007/978-981-97-7737-2_7`

Artifact: `https://github.com/vincentvbh/Float_formal`

CryptoLine

- Domain-specific language for modeling programs.
- Only accept straight-line programs (loops with fixed number of iterations).
- Very close to assembly:
  - An assembly instruction → one or more CryptoLine instructions.
- Declarative.
- At least two backend formal verification tools.

CryptoLine Verification

- Declare what we have and what we want as
  - algebraic predicates;
  - range predicates.
- Annotations. An algebraic predicate `P` and a range predicated `Q`.
  - `assume P & & Q` adds `P` and `Q` to the backend tools.
  - `assert P & & Q` asks to verify
    - `P` with the associated computer algebra system (CAS) and
    - `Q` with the associated SMT solver.
  - Transfer predicates with `assert P & & true; assume true & & P`.

Equivalences of Floating-Point Mul. Implementations

1. Translate Armv7-M assembly into CryptoLine.
2. Insert our CryptoLine model of floating-point mul.
3. Verify the equivalences of the two.
    - Question: where and what we should declare?
    - Bad annotations:
        - Verification does not halt.
    - Our annotations:
        - Verify the multi-limb splitting with SMT.
        - Transfer the range predicates to CAS.
        - Verify the long product with CAS.
        - Transfer the algebraic predicates to SMT.
        - Verify the remaining operations (rounding, zeroizing) with SMT.