Helios-Selene Optimization Contest Decision

July 11th, 2025

https://github.com/j-berman/fcmp-plus-plus-optimization-competition
https://gist.github.com/tevador/4524c2092178df08996487d4e272b096

Judges

- j-berman <justinberman@protonmail.com>
- jeffro256 <jeffro256@tutanota.com>

Contestants

- Boog900 (1 submission)
- dimalinux (1 submission)
- lederstrumpf (1 submission)
- rafael-xmr (4 submissions)
- Tritonn204 (1 submission)

Thanks to contributors

We want to thank all the contestants who submitted to the competition. A competition cannot happen without competitors. The submissions were very insightful and will undoubtedly help to improve Monero's FCMP++ upgrade. We also want to thank anyone who helped promote the competition, especially xmrack. We want to thank kayabaNerve for helping with technical analysis of the submissions. Finally, we want to thank the Monero General Fund for the prize amount and anyone who has contributed to the fund.

Deadline Extension

The original deadline for submissions was Monday, June 30th 2025 at 17:00 UTC. However, in a Monero Research Lab (MRL) meeting on Wednesday, July 2nd 2025, the judges decided to extend the deadline to Thursday, July 10th 2025. The logs for that meeting can be viewed here or here. At the first

deadline, the only submissions present were from dimalinux and rafael-xmr. The justification for the deadline extension was due to the fact that A) both submissions optimized the Ed25519 field arithmetic, which wasn't intended to be in-scope, B) both submissions broke the API of the crate, C) neither submission implemented the recommended Crandall reduction technique, and D) rafael-xmr's submissions used variable-time `u128` arithmetic (though now we know that they both did). Regardless of technicalities, we, the judges, did not feel that we could award the prize to either of the submissions as-is, but that both submissions were on the right track. We also felt that rafael-xmr's first submission performed better and was better usable as a basis for future development work. We initially desired to extend the competition for only the two contestants who submitted on-time for the first deadline, but were convinced otherwise during the MRL meeting. We instead extended the deadline for any contestants in order to apply the rules equally to all potential contestants. We chose a shorter extension period of one week to still allow an advantage for the contestants who worked on, and submitted, their work on-time. Along with the deadline extension, additional rules were added on July 3rd to clarify the bounds of the competition. They can be found here and here. The submission referred to in this document as belonging to dimalinux was submitted before the first deadline, on June 25th, and has not been updated since. It is considered as invalid, but we include it in this report to be comprehensive.

Field25519 rule clarifications

In the new competition rules introduced with the deadline extension, we explicitly forbade optimizing arithmetic for the field defined over the prime $2^{255} - 19$, AKA Field25519. This is because there is absolutely no shortage of high-quality, high-performance, cross-platform libraries for performing operations on Field25519. The X25519 elliptic curve Diffie-Helman (ECDH) key exchange algorithm was published in 2006, which would have been the first popular cryptosystem to use this field. Its bi-rationally equivalent cousin curve, Ed25519, which Monero uses as its main curve for signatures, was published in 2012. Ed25519 also uses Field25519. The popularization and proliferation of these cryptosystems in highly-used cryptographic applications over the years has resulted in a great number of extremely high-quality implementations of Field25519. An excellent example of this is the "ed25519_dalek" Rust crate. There were plans to switch out the Field25519 arithmetic to the Dalek implementation at some point as well. We knew this from the beginning and meant for Field25519 arithmetic to be out of scope for the competition, only desiring speed-ups for the actual Helios-Selene field. Unfortunately, we made the oversight of assuming that contestants knew that Field25519 would be out of scope and did not explicitly prohibit contestants from optimizing Field25519. It did help the "fairness" of the situation that both first deadline submissions worked to optimized Field25519, so we think that the effect of applying this new rule had somewhat of a equitable impact on both contestants.

Nominal Benchmark Data

This benchmark section only includes data collected from unmodified submissions, invalid or not.

Benchmarks for AMD Ryzen 5600G, Rust 1.84.1

Percentage improvement in unmodified submissions' run-times as compared to the reference implementation on a *AMD Ryzen 5600G*:

Section	Boog900	dimalinux	lederstrumpf	rafael-xmr *no-lazy*	rafael-xmr *no-inv-no-lazy (inferred)*	Tritonn204
Selene Point Add	18.77%	17.77%	28.39%	45.68%	45.68%	18.7%
Helios Point Add	5.15%	21.3%	14.89%	15%	15%	15.03%
Field Multiply	14.26%	18.82%	21%	21.64%	21.64%	-8.97%
Field Invert	-3.88%	6.41%	-1.14%	52.55%	0%	20.39%
Selene Point Decompression	57.39%	36.76%	33.4%	40.48%	40.48%	44.66%
Helios Point Decompression	-0.3%	30.35%	0.3%	0.15%	0.15%	40.02%
Field Add	11.29%	58.73%	57.64%	11.36%	11.36%	93.34%
Field Subtract	-11.18%	63.02%	53.74%	7.48%	7.48%	92.69%
Selene Scalar-Point Multiply	10.53%	14.33%	16.44%	34.65%	34.65%	-2.99%
Helios Scalar-Point Multiply	1.65%	18.38%	6.03%	1.56%	1.56%	3.11%
Overall	**12.75%**	**23.93%**	**22.44%**	**29.35%**	**24.09%**	**24.21%**

Benchmarks for WASM Cycles, Rust 1.84.1

Percentage improvement in unmodified submissions' WebAssembly cycle count as compared to the reference implementation:

Section	Boog900	dimalinux	lederstrumpf	rafael-xmr *no-lazy*	Rafael-xmr *no-inv-no-lazy (inferred)*	Tritonn204
Selene Point Add	39.07%	30.28%	62.69%	44.98%	44.98%	36.20%
Helios Point Add	6.08%	38.44%	17.92%	17.84%	17.84%	16.89%
Field Multiply	32.37%	22.06%	56.99%	30.31%	30.31%	18.44%
Field Invert	-79.63%	48.12%	1.36%	68.70%	0.00%	58.33%
Selene Point Decompression	34.56%	48.57%	57.94%	53.01%	53.01%	54.13%
Helios Point Decompression	0.03%	51.92%	0.00%	0.00%	0.00%	34.74%
Field Add	50.90%	53.12%	34.68%	45.50%	45.50%	78.83%
Field Subtract	33.68%	58.95%	22.63%	43.68%	43.68%	72.63%
Selene Scalar-Point Multiply	37.92%	31.86%	59.22%	36.16%	36.16%	27.38%
Helios Scalar-Point Multiply	2.13%	39.80%	7.10%	6.68%	6.68%	2.50%
Overall	**17.35%**	**37.90%**	**39.05%**	**37.09%**	**30.22%**	**36.98%**

On `u128`'s variable run-time operations

Rust provides a primitive 128-bit unsigned integer type, `u128`. It is exactly analogous to `u64`, `u32`, `u16`, and `u8`. However, this type became a sort of stumbling block for this competition. The Rust compiler does not emit machine code that is directly readable by CPUs, but instead "intermediate representation" (IR) code. IR is a architecture-agnostic program format, rich with computer-meaningful information, which can be converted to actual machine code by a "backend" compiler. Specifically, the reference Rust toolchain uses the Rust compiler to convert Rust code to IR, which is then inputted to LLVM backend. Since operations on `u128s` are not guaranteed to be constant-time by contract, and since most CPUs do not have native non-vector instructions to operate on 128-bit integers, LLVM gets to decide how to implement `u128`. On most modern 64-bit CPUs, some of these operations can be implemented as constant-time in a performant manner. However, this is much harder to do in 32-bit architectures. It can be demonstrated concretely that for certain 32-bit architectures, LLVM produces branching instructions when doing operations on u128s. For example, observe the assembly code generated by Godbolt for `u128` addition on RISC-V here. It contains the `beq` "branch if equal" instruction dependent on the carry-bit. This means that the run-time of the operation is dependent on the operands. This opens the door to Timing Attacks.

This is why the rule "Implementations must run in constant time" is rule number 1. And since `u128` operations are not constant time, submissions which use this are technically invalid, and at the least not preferred. Unfortunately, four out of the five submitters used `u128` in their submissions:

Boog900, dimalinux, rafael-xmr, and Tritonn204. The only exception was lederstrumpf. Because of how widespread its use was in this competition, regardless of it being variable-time, we didn't want to outright disqualify submissions for only this reason. However, it was a factor in the final decision.

Omitting detailed submission analysis

We compiled quite a lot of detailed feedback on the submissions, some objective, some subjective. However, we chose to omit detailed feedback on submissions from the public decision announcement out of respect for the contestants. Upon permission of the contestants, we would be happy to release more details, or we may release details at discretion if the judges are contacted directly.

Eliminating some submissions

At this point, we decide to eliminate Boog900 and dimalinux's submission. dimalinux's because it would take too much reworking to get it usable for the FCMP++ integration: variable-time arithmetic, reverting the Ed25519 changes, cleaning up the vendored `crypto_bigint` fork, etc. Additionally, we expect the code's performance metrics to be impacted negatively after making such changes. Boog900's since it runs into some of the same variable-time pitfalls as the other submissions, but generally doesn't perform as well.

We also chose not to include rafael-xmr's "lazy reduction" variants for further consideration, since we found some issues with the his implementation that causes incorrect results in some cases, and the general methodology is must harder to establish that it would stay correct in all cases as we use it in FCMP++. It was only a few percentage points faster in the overall speed as compared to excluding lazy reduction, so it shouldn't have a major impact on judging.

Refining submissions

This leaves us with three submissions: rafael-xmr, lederstrumpf, and Tritonn204. As according to rule 1, both rafael-xmr's and Tritonn204's submissions are invalid because of the variable-time arithmetic. However, we wanted to see if relatively small tweaks to rafael-xmr's and Tritonn204's submissions could result in submissions that were still competitive in performance. For rafael-xmr's submissions, we decided to go with the no-inversion, no-lazy-reduction submission. We didn't have any specific issue with rafael-xmr's inversion algorithm, but it would require much more refactoring of the arithmetic to remove `u128` from it. For Tritonn204's submission, we reverted the `Field25519` square-root and exponentiation functions. We didn't try removing `u128` completely since that would also require a lot of effort. The results can be seen in the following sections.

We also note that Tritonn204's method of "lazy addition" is not safe when exposed to a public API as it would be used in FCMP++. As written, we can break field addition in test cases, but as is used internally in point operations, it is okay. This would have to be revisited if we chose to move forward with Tritonn204's submission.

Modified Benchmark Data

This benchmark section includes data collected from lederstrumpf's unmodified submission and rafael-xmr and Tritonn204's modified submissions, as outlined in the "Refining Submissions" section.

Benchmarks for AMD Ryzen 5600G, Rust 1.84.1

Percentage improvement in *modified* submissions' run-times as compared to the reference implementation on a *AMD Ryzen 5600G*:

Section	lederstrumpf	rafael-xmr (no inverse, no lazy, reverted u128)	Tritonn204 (reverted Field25519 changes, still uses u128)
Selene Point Add	28.39%	39.38%	18.10%
Helios Point Add	14.89%	13.49%	14.78%
Field Multiply	21.00%	19.98%	-6.40%
Field Invert	-1.14%	0.43%	21.96%
Selene Point Decompression	33.40%	37.05%	44.63%
Helios Point Decompression	0.30%	-0.23%	-0.23%
Field Add	57.64%	11.36%	93.45%
Field Subtract	53.74%	7.48%	92.60%
Selene Scalar-Point Multiply	16.44%	28.28%	-2.74%
Helios Scalar-Point Multiply	6.03%	1.97%	1.18%
Overall	**22.44%**	**21.34%**	**21.48%**

On lederstrumpf's submission

lederstrumpf's submission was the only submission not to use variable-time arithmetic. Its performance was competitive against unmodified submissions on real machines (except with Rust 1.69.0 for some reason) without changes to Field25519 arithmetic or a specific field inversion algorithm. It was one of the smallest submissions in terms of line count, the code was arguably the easiest to follow, and it performed the best on WASM. And it was one of the best performing submissions on real machines after correcting some of the issues with the other submissions. lederstrumpf's submission is very much a blank slate for future improvements, and largely doesn't have much in it that we would want to revert before FCMP++ integration.

Decision

The judges unanimously decide to declare lederstrumpf as the winner of the Helios-Selene optimization contest. In our view, lederstrumpf's submission is the strongest foundation to build on. Congratulations on an excellent submission! Again, we want to thank all of the contestants who participated for making this such a challenging and interesting competition.

Appendix A: Other nominal benchmark data

Below is some benchmark data for unmodified submissions on platforms not explicitly targeted in the competition, but which may be insightful nonetheless.

Benchmarks for Intel i7-1355U, Rust 1.84.1

Percentage improvement in unmodified submissions' run-times as compared to the reference implementation on a *13th Gen Intel(R) Core(TM) i7-1355U*:

Section	Boog900	dimalinux	lederstrumpf	rafael-xmr *no-lazy*	rafael-xmr *no-inv-no-lazy*	Tritonn204
Selene Point Add	25.62%	19.69%	31.56%	47.89%	47.30%	22.23%
Helios Point Add	4.95%	24.92%	15.46%	16.06%	15.89%	16.38%
Field Multiply	18.24%	21.91%	14.14%	19.96%	18.28%	-14.04%
Field Invert	-7.60%	3.26%	0.17%	45.00%	10.04%	13.17%
Selene Point Decompression	57.02%	39.26%	31.72%	40.19%	39.77%	38.56%
Helios Point Decompression	-0.34%	34.50%	0.27%	2.31%	2.74%	41.34%
Field Add	12.83%	63.17%	65.25%	12.24%	10.64%	93.37%
Field Subtract	-15.27%	72.56%	60.61%	6.68%	8.15%	92.35%
Selene Scalar-Point Multiply	17.12%	17.83%	16.08%	36.70%	35.74%	0.05%
Helios Scalar-Point Multiply	1.73%	22.38%	6.29%	-0.55%	-3.27%	3.87%
Overall	**15.00%**	**26.58%**	**23.18%**	**29.31**	**25.26%**	**23.71%**

Benchmarks for Intel i7-1355U, Rust 1.69.0

Percentage improvement in unmodified submissions' run-times as compared to the reference implementation on a *13th Gen Intel Core i7-1355U*:

Section	Boog900	dimalinux	lederstrumpf	rafael-xmr *no-lazy*	Rafael-xmr *no-inv-no-lazy*	Tritonn204
Selene Point Add	31.89%	26.56%	-4.71%	37.36%	38.68%	27.36%
Helios Point Add	7.18%	44.01%	7.15%	12.17%	11.29%	7.08%
Field Multiply	19.00%	4.52%	1.65%	18.46%	18.29%	-13.71%
Field Invert	69.30%	73.10%	-0.33%	84.87%	-1.08%	75.13%
Selene Point Decompression	63.87%	52.17%	35.43%	41.29%	41.04%	51.41%
Helios Point Decompression	0.19%	31.63%	-0.37%	0.03%	0.11%	28.41%
Field Add	31.65%	67.78%	1.99%	26.22%	26.38%	71.98%
Field Subtract	16.38%	71.31%	-4.77%	30.55%	30.57%	67.92%
Selene Scalar-Point Multiply	22.92%	26.27%	-12.18%	30.66%	30.72%	9.93%
Helios Scalar-Point Multiply	1.92%	42.75%	2.82%	2.91%	1.62%	2.67%
Overall	**28.25%**	**37.52%**	**2.13%**	**31.06%**	**22.67%**	**28.02%**