Accelerating and Verifying Constant-Time Modular Inversion

Daniel J. Bernstein Han-Ting Chen John R. Harrison
Cesare Huang Gregory Maxwell Bow-Yaw Wang
Pieter Wuille Bo-Yin Yang

May 12, 2026

Introduction (1)

- If f and g are coprime integers then there exists an integer r such that $gr = 1 \pmod{f}$ and is denoted $g^{-1} \pmod{f}$.

Introduction (1)

- If f and g are coprime integers then there exists an integer r such that $gr = 1 \pmod{f}$ and is denoted $g^{-1} \pmod{f}$.
- Modular inversion is used in several cryptography systems, such as RSA and ECC.

Introduction (1)

- If f and g are coprime integers then there exists an integer r such that $gr = 1 \pmod{f}$ and is denoted $g^{-1} \pmod{f}$.
- Modular inversion is used in several cryptography systems, such as RSA and ECC.
- Besides, the element to invert is usually secret data in these systems, so modular inversion requires a constant-time implementation.

Introduction (1)

- If f and g are coprime integers then there exists an integer r such that $gr = 1 \pmod{f}$ and is denoted $g^{-1} \pmod{f}$.
- Modular inversion is used in several cryptography systems, such as RSA and ECC.
- Besides, the element to invert is usually secret data in these systems, so modular inversion requires a constant-time implementation.
- The traditional Extended GCD algorithm takes variable time, and the standard constant-time method is to use Fermat's Little Theorem, raising the element to the $(p - 2)$th power, but takes a lot of multiplications.

Introduction (2)

- This work extends the result of Bernstein and Yang *Fast constant-time gcd computation and modular inversion* in 2019.

Introduction (2)

- This work extends the result of Bernstein and Yang *Fast constant-time gcd computation and modular inversion* in 2019.
- In that paper, they use a `divstep`-based (division-step) approach to compute modular inverse, the pseudo-code is:

$\delta \leftarrow 1$
$f \leftarrow p$ the prime modulus
$g \leftarrow x$ the element to invert
`for` $i = 0, \ldots, N-1$ `do`
  $(\delta, f, g) \leftarrow$ `divstep`(δ, f, g)
  `collect the trace`
`get` r `from the trace`
`return` r

Introduction (2)

- This work extends the result of Bernstein and Yang *Fast constant-time gcd computation and modular inversion* in 2019.
- In that paper, they use a `divstep`-based (division-step) approach to compute modular inverse, the pseudo-code is:

$\delta \leftarrow 1$
$f \leftarrow p$ the prime modulus
$g \leftarrow x$ the element to invert
`for` $i = 0, \ldots, N-1$ `do`
  $(\delta, f, g) \leftarrow$ `divstep`(δ, f, g)
  `collect the trace`
`get` r `from the trace`
`return` r

- For b-bit inputs f and g, it takes $N \geq 2.88b$ steps to get the correct r from the `divstep` trace. Concretely, the 25519 prime requires $N = 722$ divsteps.

Introduction (3)

$$\texttt{divstep}(\delta, f, g) = \begin{cases} (1-\delta, g, (g-f)/2), & \text{if } \delta > 0 \text{ and } g \text{ is odd} \\ (1+\delta, f, (g+(g \bmod 2)f)/2), & \text{otherwise.} \end{cases}$$

- Key Observations:

Step	δ	f	g
0	1	19	9
1	0	9	-5
2	1	9	2
3	2	9	1
4	-1	1	-4
5	0	1	-2
6	1	1	-1
7	0	-1	-1
8	1	-1	-1
9	0	-1	0

Introduction (3)

$$\texttt{divstep}(\delta, f, g) = \begin{cases} (1-\delta, g, (g-f)/2), & \text{if } \delta > 0 \text{ and } g \text{ is odd} \\ (1+\delta, f, (g+(g \bmod 2)f)/2), & \text{otherwise.} \end{cases}$$

- Key Observations:
  1. `divstep` acts on (f, g) with linear transformations.

Step	δ	f	g
0	1	19	9
1	0	9	-5
2	1	9	2
3	2	9	1
4	-1	1	-4
5	0	1	-2
6	1	1	-1
7	0	-1	-1
8	1	-1	-1
9	0	-1	0

Introduction (3)

$$\texttt{divstep}(\delta, f, g) = \begin{cases} (1-\delta, g, (g-f)/2), & \text{if } \delta > 0 \text{ and } g \text{ is odd} \\ (1+\delta, f, (g+(g \bmod 2)f)/2), & \text{otherwise.} \end{cases}$$

- Key Observations:
  1. `divstep` acts on (f, g) with linear transformations.
  2. f is always odd and $\gcd(f_n, g_n) = \gcd(f_0, g_0)$.

Step	δ	f	g
0	1	19	9
1	0	9	-5
2	1	9	2
3	2	9	1
4	-1	1	-4
5	0	1	-2
6	1	1	-1
7	0	-1	-1
8	1	-1	-1
9	0	-1	0

Introduction (3)

$$\texttt{divstep}(\delta, f, g) = \begin{cases} (1-\delta, g, (g-f)/2), & \text{if } \delta > 0 \text{ and } g \text{ is odd} \\ (1+\delta, f, (g+(g \bmod 2)f)/2), & \text{otherwise.} \end{cases}$$

- Key Observations:
  1. `divstep` acts on (f, g) with linear transformations.
  2. f is always odd and $\gcd(f_n, g_n) = \gcd(f_0, g_0)$.
  3. The computation of `divstep` can be made constant-time.

Step	δ	f	g
0	1	19	9
1	0	9	-5
2	1	9	2
3	2	9	1
4	-1	1	-4
5	0	1	-2
6	1	1	-1
7	0	-1	-1
8	1	-1	-1
9	0	-1	0

Introduction (3)

$$\texttt{divstep}(\delta, f, g) = \begin{cases} (1-\delta, g, (g-f)/2), & \text{if } \delta > 0 \text{ and } g \text{ is odd} \\ (1+\delta, f, (g+(g \bmod 2)f)/2), & \text{otherwise.} \end{cases}$$

- Key Observations:
  1. `divstep` acts on (f, g) with linear transformations.
  2. f is always odd and $\gcd(f_n, g_n) = \gcd(f_0, g_0)$.
  3. The computation of `divstep` can be made constant-time.
- (Theorem 11.2 in [BY19]) If $\delta_0 = 1$ and $n \geq 2.88b$ where b is the size of input f, g, then $g_n = 0$.

Step	δ	f	g
0	1	19	9
1	0	9	-5
2	1	9	2
3	2	9	1
4	-1	1	-4
5	0	1	-2
6	1	1	-1
7	0	-1	-1
8	1	-1	-1
9	0	-1	0

Introduction (3)

$$\texttt{divstep}(\delta, f, g) = \begin{cases} (1-\delta, g, (g-f)/2), & \text{if } \delta > 0 \text{ and } g \text{ is odd} \\ (1+\delta, f, (g+(g \bmod 2)f)/2), & \text{otherwise.} \end{cases}$$

- Key Observations:
  1. `divstep` acts on (f, g) with linear transformations.
  2. f is always odd and $\gcd(f_n, g_n) = \gcd(f_0, g_0)$.
  3. The computation of `divstep` can be made constant-time.
- (Theorem 11.2 in [BY19]) If $\delta_0 = 1$ and $n \geq 2.88b$ where b is the size of input f, g, then $g_n = 0$.
- In that case, $\gcd(f_0, g_0) = \gcd(f_n, g_n) = \gcd(f_n, 0) = |f_n|$.

Step	δ	f	g
0	1	19	9
1	0	9	-5
2	1	9	2
3	2	9	1
4	-1	1	-4
5	0	1	-2
6	1	1	-1
7	0	-1	-1
8	1	-1	-1
9	0	-1	0

Introduction (3)

$$\texttt{divstep}(\delta, f, g) = \begin{cases} (1-\delta, g, (g-f)/2), & \text{if } \delta > 0 \text{ and } g \text{ is odd} \\ (1+\delta, f, (g+(g \bmod 2)f)/2), & \text{otherwise.} \end{cases}$$

- Key Observations:
  1. `divstep` acts on (f, g) with linear transformations.
  2. f is always odd and $\gcd(f_n, g_n) = \gcd(f_0, g_0)$.
  3. The computation of `divstep` can be made constant-time.
- (Theorem 11.2 in [BY19]) If $\delta_0 = 1$ and $n \geq 2.88b$ where b is the size of input f, g, then $g_n = 0$.
- In that case, $\gcd(f_0, g_0) = \gcd(f_n, g_n) = \gcd(f_n, 0) = |f_n|$.
- If inputs f_0 and g_0 are coprime, then $f_n = \pm 1$. By composing the linear transformations applied to f and g, one recovers the Bezout coefficients and the modular inverse.

Step	δ	f	g
0	1	19	9
1	0	9	-5
2	1	9	2
3	2	9	1
4	-1	1	-4
5	0	1	-2
6	1	1	-1
7	0	-1	-1
8	1	-1	-1
9	0	-1	0

Summary of our Contribution

- Algorithmic Improvement: It requires $2.88b$ `divsteps` to drive $g = 0$. Can we reduce the number of required `divsteps`?

Summary of our Contribution

- Algorithmic Improvement: It requires $2.88b$ `divsteps` to drive $g = 0$. Can we reduce the number of required `divsteps`?
  - YES! Instead of beginning from $\delta_0 = 1$, which requires $2.88b$ `divsteps`, beginning from $\delta_0 = \frac{1}{2}$ requires $2.304b$ steps, which is about 20% fewer.

Summary of our Contribution

- Algorithmic Improvement: It requires $2.88b$ `divsteps` to drive $g = 0$. Can we reduce the number of required `divsteps`?
  - YES! Instead of beginning from $\delta_0 = 1$, which requires $2.88b$ `divsteps`, beginning from $\delta_0 = \frac{1}{2}$ requires $2.304b$ steps, which is about 20% fewer.
- Technique Improvement: The bound $2.88b$ is obtained with large computation in the previous paper. Can we have a better analysis tool for this?

Summary of our Contribution

- Algorithmic Improvement: It requires $2.88b$ `divsteps` to drive $g = 0$. Can we reduce the number of required `divsteps`?
  - YES! Instead of beginning from $\delta_0 = 1$, which requires $2.88b$ `divsteps`, beginning from $\delta_0 = \frac{1}{2}$ requires $2.304b$ steps, which is about 20% fewer.
- Technique Improvement: The bound $2.88b$ is obtained with large computation in the previous paper. Can we have a better analysis tool for this?
  - YES! We propose two techniques: Convex Hull and Stable Hull. Convex Hull provides the number of required steps for a fixed input size b; Stable Hull proves that $2.304b$ steps suffice for any bit size b.

Summary of our Contribution

- Algorithmic Improvement: It requires $2.88b$ `divsteps` to drive $g = 0$. Can we reduce the number of required `divsteps`?
  - YES! Instead of beginning from $\delta_0 = 1$, which requires $2.88b$ `divsteps`, beginning from $\delta_0 = \frac{1}{2}$ requires $2.304b$ steps, which is about 20% fewer.
- Technique Improvement: The bound $2.88b$ is obtained with large computation in the previous paper. Can we have a better analysis tool for this?
  - YES! We propose two techniques: Convex Hull and Stable Hull. Convex Hull provides the number of required steps for a fixed input size b; Stable Hull proves that $2.304b$ steps suffice for any bit size b.
- Software Contribution: On modern machines, how many cycles are used per `divstep`?

Summary of our Contribution

- Algorithmic Improvement: It requires $2.88b$ `divsteps` to drive $g = 0$. Can we reduce the number of required `divsteps`?
  - YES! Instead of beginning from $\delta_0 = 1$, which requires $2.88b$ `divsteps`, beginning from $\delta_0 = \frac{1}{2}$ requires $2.304b$ steps, which is about 20% fewer.
- Technique Improvement: The bound $2.88b$ is obtained with large computation in the previous paper. Can we have a better analysis tool for this?
  - YES! We propose two techniques: Convex Hull and Stable Hull. Convex Hull provides the number of required steps for a fixed input size b; Stable Hull proves that $2.304b$ steps suffice for any bit size b.
- Software Contribution: On modern machines, how many cycles are used per `divstep`?
  - Under 7 cycles! And we implement the `divstep`-based modular inversion on ARM and X86-64 architectures, see the next slides for the performance chart.

Summary of our Contribution

- Algorithmic Improvement: It requires $2.88b$ `divsteps` to drive $g = 0$. Can we reduce the number of required `divsteps`?
  - YES! Instead of beginning from $\delta_0 = 1$, which requires $2.88b$ `divsteps`, beginning from $\delta_0 = \frac{1}{2}$ requires $2.304b$ steps, which is about 20% fewer.
- Technique Improvement: The bound $2.88b$ is obtained with large computation in the previous paper. Can we have a better analysis tool for this?
  - YES! We propose two techniques: Convex Hull and Stable Hull. Convex Hull provides the number of required steps for a fixed input size b; Stable Hull proves that $2.304b$ steps suffice for any bit size b.
- Software Contribution: On modern machines, how many cycles are used per `divstep`?
  - Under 7 cycles! And we implement the `divstep`-based modular inversion on ARM and X86-64 architectures, see the next slides for the performance chart.
- Formal Verification: We provide machine-checked proofs

Summary of our Contribution

- Algorithmic Improvement: It requires $2.88b$ `divsteps` to drive $g = 0$. Can we reduce the number of required `divsteps`?
  - YES! Instead of beginning from $\delta_0 = 1$, which requires $2.88b$ `divsteps`, beginning from $\delta_0 = \frac{1}{2}$ requires $2.304b$ steps, which is about 20% fewer.
- Technique Improvement: The bound $2.88b$ is obtained with large computation in the previous paper. Can we have a better analysis tool for this?
  - YES! We propose two techniques: Convex Hull and Stable Hull. Convex Hull provides the number of required steps for a fixed input size b; Stable Hull proves that $2.304b$ steps suffice for any bit size b.
- Software Contribution: On modern machines, how many cycles are used per `divstep`?
  - Under 7 cycles! And we implement the `divstep`-based modular inversion on ARM and X86-64 architectures, see the next slides for the performance chart.
- Formal Verification: We provide machine-checked proofs
  - The theorem on required steps ($2.304b$ steps) is proved by HOL-light and both CryptoLine and HOL-light are used to verify software implementations.

Summary of our Contribution: invert modulo $2^{255} - 19$ on x86-64

Software	Verif	Val	$[1/8, 3/8]$	$[3/8, 5/8]$	$[5/8, 7/8]$	Host
this paper (divstep; Inverter 1X)	**yes**	**yes**	3822.63	3862.18	3901.18	samba
this paper (divstep; Inverter 2X)	**yes**	**yes**	4053.87	4074.33	4096.86	samba
[4]+[8] (divstep)	**yes**	**no**	4332.25	4447.68	4585.41	samba
[1] (semi-binary)	**no**	**yes**	5766.88	5788.78	5811.31	samba
[3] (semi-binary)	**yes**	**yes**	6012.25	6031.28	6057.10	samba
`mpn_gcdext` [7]	**no**	**no**	6570.58	6757.64	6939.16	samba
[4] (divstep)	**no**	**yes**	8844.80	8867.19	8890.62	samba
[5] (Fermat)	**no**	**yes**	8966.81	8975.10	8984.81	samba
[6] (divstep)	**yes**	**yes**	18475.84	18507.89	18541.16	samba
`BN_mod_inverse` [9]	**no**	**no**	57993.47	58850.63	59762.12	samba
`mpn_sec_invert` [7]	**no**	**yes**	74701.91	74954.04	75208.72	samba
`BN_mod_inverse_no_branch` [9]	**no**	**no**	72252.57	75197.39	78521.38	samba

Summary of our Contribution: invert modulo $2^{255} - 19$ on ARM A-profile

Software	Verif	Val	$[1/8, 3/8]$	$[3/8, 5/8]$	$[5/8, 7/8]$	Host
this paper (divstep; Inverter 1A)	**yes**	**yes**	3955.25	3963.88	3973.15	pi5
`mpn_gcdext` [7]	**no**	**no**	4440.62	4544.22	4646.28	pi5
this paper (divstep; Inverter 2A)	**yes**	**yes**	5588.25	5600.96	5612.74	pi5
[4]+[8] (divstep)	**yes**	**no**	5915.92	6054.42	6191.94	pi5
[2] (semi-binary)	**yes**	**yes**	6660.37	6668.38	6677.19	pi5
[4] (divstep)	**no**	**yes**	11980.70	12005.45	12031.28	pi5
[6] (divstep)	**yes**	**yes**	20133.92	20154.97	20176.29	pi5
`BN_mod_inverse` [9]	**no**	**no**	57318.26	58021.03	58787.02	pi5
`mpn_sec_invert` [7]	**no**	**yes**	59600.23	65691.68	66700.73	pi5
`BN_mod_inverse_no_branch` [9]	**no**	**no**	78591.13	81971.99	85489.24	pi5

Begin from $\delta_0 = \frac{1}{2}$ (1)

- There are several reasons to use non-integer δ:

Begin from $\delta_0 = \frac{1}{2}$ (1)

- There are several reasons to use non-integer δ:
- In `divsteps`, δ is a quick estimate to the size difference, i.e., $\log g - \log f$, it is natural to allow non-integer values.

Begin from $\delta_0 = \frac{1}{2}$ (1)

- There are several reasons to use non-integer δ:
- In `divsteps`, δ is a quick estimate to the size difference, i.e., $\log g - \log f$, it is natural to allow non-integer values.
- Brute-force computation for small bit sizes also shows $\delta_0 = \frac{1}{2}$ requires fewer steps than $\delta_0 = 1$.

Begin from $\delta_0 = \frac{1}{2}$ (1)

- There are several reasons to use non-integer δ:
- In `divsteps`, δ is a quick estimate to the size difference, i.e., $\log g - \log f$, it is natural to allow non-integer values.
- Brute-force computation for small bit sizes also shows $\delta_0 = \frac{1}{2}$ requires fewer steps than $\delta_0 = 1$.
- Here, we present an interesting story related to program testing.

Begin from $\delta_0 = \frac{1}{2}$ (2): An Interesting Story

- A standard way to check whether a program is correct is to test it. However, *how effective is a test?*

Begin from $\delta_0 = \frac{1}{2}$ (2): An Interesting Story

- A standard way to check whether a program is correct is to test it. However, *how effective is a test?*
- A common way is to intentionally introduce some small modifications (mutations) into the software.

Begin from $\delta_0 = \frac{1}{2}$ (2): An Interesting Story

- A standard way to check whether a program is correct is to test it. However, *how effective is a test?*
- A common way is to intentionally introduce some small modifications (mutations) into the software.
- For instance, in a big-number adder, one introduces the mutation:

```
...          ...
adcs         adcs
adcs    →     adc
adcs         adcs
...          ...
```

Begin from $\delta_0 = \frac{1}{2}$ (2): An Interesting Story

- A standard way to check whether a program is correct is to test it. However, *how effective is a test?*
- A common way is to intentionally introduce some small modifications (mutations) into the software.
- For instance, in a big-number adder, one introduces the mutation:

```
...          ...
adcs         adcs
adcs    →     adc
adcs         adcs
...          ...
```

- The testing software should reject this mutated software. We test the tests

Begin from $\delta_0 = \frac{1}{2}$ (3): An Interesting Story

- One of our authors reported the following.

Begin from $\delta_0 = \frac{1}{2}$ (3): An Interesting Story

- One of our authors reported the following.
- We used the above strategy to test the original ($\delta_0 = 1$) modular inversion program.

Begin from $\delta_0 = \frac{1}{2}$ (3): An Interesting Story

- One of our authors reported the following.
- We used the above strategy to test the original ($\delta_0 = 1$) modular inversion program.
- We expected all mutated programs to be rejected.

Begin from $\delta_0 = \frac{1}{2}$ (3): An Interesting Story

- One of our authors reported the following.
- We used the above strategy to test the original ($\delta_0 = 1$) modular inversion program.
- We expected all mutated programs to be rejected.
- However, one of them passes the tests.

Begin from $\delta_0 = \frac{1}{2}$ (3): An Interesting Story

- One of our authors reported the following.
- We used the above strategy to test the original ($\delta_0 = 1$) modular inversion program.
- We expected all mutated programs to be rejected.
- However, one of them passes the tests.
- But it was not a flaw in the tests, but a correct and even faster program!

Begin from $\delta_0 = \frac{1}{2}$ (3): An Interesting Story

- One of our authors reported the following.
- We used the above strategy to test the original ($\delta_0 = 1$) modular inversion program.
- We expected all mutated programs to be rejected.
- However, one of them passes the tests.
- But it was not a flaw in the tests, but a correct and even faster program!
- It turns out that the passing mutated program is equivalent to `divstep` with $\delta_0 = \frac{1}{2}$.

Convex Hull (1)

- Convex Hull is a technique used to calculate and prove the required steps for certain input bit sizes.

Convex Hull (1)

- Convex Hull is a technique used to calculate and prove the required steps for certain input bit sizes.
- Take 5-bit odd prime p and $\delta_0 = \frac{1}{2}$ for instance. Let $f_0 = p$ and $g_0 = x$ (the element to invert), and let $(\delta_n, f_n, g_n) = \texttt{divstep}^n(\delta_0, f_0, g_0)$.

Convex Hull (1)

- Convex Hull is a technique used to calculate and prove the required steps for certain input bit sizes.
- Take 5-bit odd prime p and $\delta_0 = \frac{1}{2}$ for instance. Let $f_0 = p$ and $g_0 = x$ (the element to invert), and let $(\delta_n, f_n, g_n) = \texttt{divstep}^n(\delta_0, f_0, g_0)$.
- How many steps are required to drive $g_n = 0$?

Convex Hull (1)

- Convex Hull is a technique used to calculate and prove the required steps for certain input bit sizes.
- Take 5-bit odd prime p and $\delta_0 = \frac{1}{2}$ for instance. Let $f_0 = p$ and $g_0 = x$ (the element to invert), and let $(\delta_n, f_n, g_n) = \texttt{divstep}^n(\delta_0, f_0, g_0)$.
- How many steps are required to drive $g_n = 0$?
- Recall that `divstep` acts on f, g with linear transformations

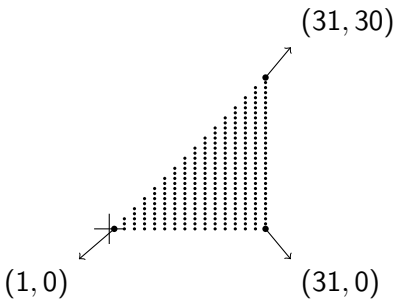
$$\texttt{divstep}(\delta, f, g) = \begin{cases} (1-\delta, g, (g-f)/2), & \text{if } \delta > 0 \text{ and } g \text{ is odd} \\ (1+\delta, f, (g+(g \bmod 2)f)/2), & \text{otherwise.} \end{cases}$$

and in fact, there are only three possible linear transformations.

$$(x,y) \overset{M_{-1}}{\longmapsto} (y,(y-x)/2), \quad (x,y) \overset{M_0}{\longmapsto} (x,y/2), \quad (x,y) \overset{M_1}{\longmapsto} (x,(y+x)/2)$$

Convex Hull (2)

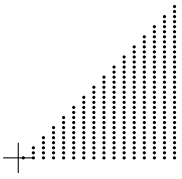
- We can plot the (f_n, g_n) pairs on the plane and evolve them with `divstep`.
- Note that $0 \leq g_0 < f_0 < 2^5$.



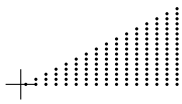
$n = 0$, $\delta_0 = \frac{1}{2}$

Convex Hull (2)

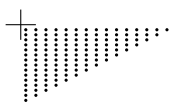
- We can plot the (f_n, g_n) pairs on the plane and evolve them with `divstep`.
- Note that $0 \leq g_0 < f_0 < 2^5$.



$n = 0,\ \delta_0 = \frac{1}{2}$



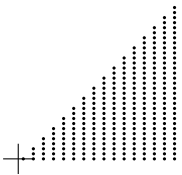
$n = 1,\ \delta_1 = \frac{3}{2}$



$n = 1,\ \delta_1 = \frac{1}{2}$

Convex Hull (2)

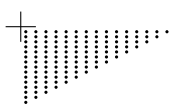
- We can plot the (f_n, g_n) pairs on the plane and evolve them with `divstep`.
- Note that $0 \leq g_0 < f_0 < 2^5$.



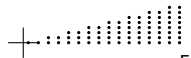
$n = 0,\ \delta_0 = \frac{1}{2}$



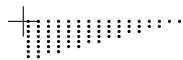
$n = 1,\ \delta_1 = \frac{3}{2}$



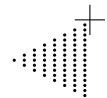
$n = 1,\ \delta_1 = \frac{1}{2}$



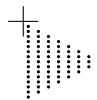
$n = 2,\ \delta_2 = \frac{5}{2}$



$n = 2,\ \delta_2 = \frac{3}{2}$



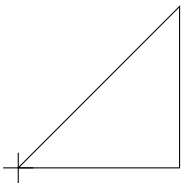
$n = 2,\ \delta_2 = \frac{1}{2}$



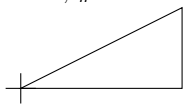
$n = 2,\ \delta_2 = \frac{-1}{2}$

Convex Hull (3)

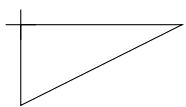
- Better: Use convex hulls to cover the states on the plane and hence reduce the cost of computation used to analyze the algorithm.
- We build the set of vertices S_{n,δ_n} such that for every possible state (δ_n, f_n, g_n) in `divstep`, (f_n, g_n) lies in $\mathrm{Hull} S_{n,\delta_n}$.



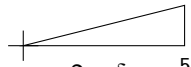
$n = 0,\ \delta_0 = \frac{1}{2}$



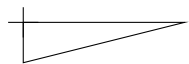
$n = 1,\ \delta_1 = \frac{3}{2}$



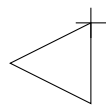
$n = 1,\ \delta_1 = \frac{1}{2}$



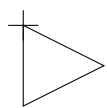
$n = 2,\ \delta_2 = \frac{5}{2}$



$n = 2,\ \delta_2 = \frac{3}{2}$

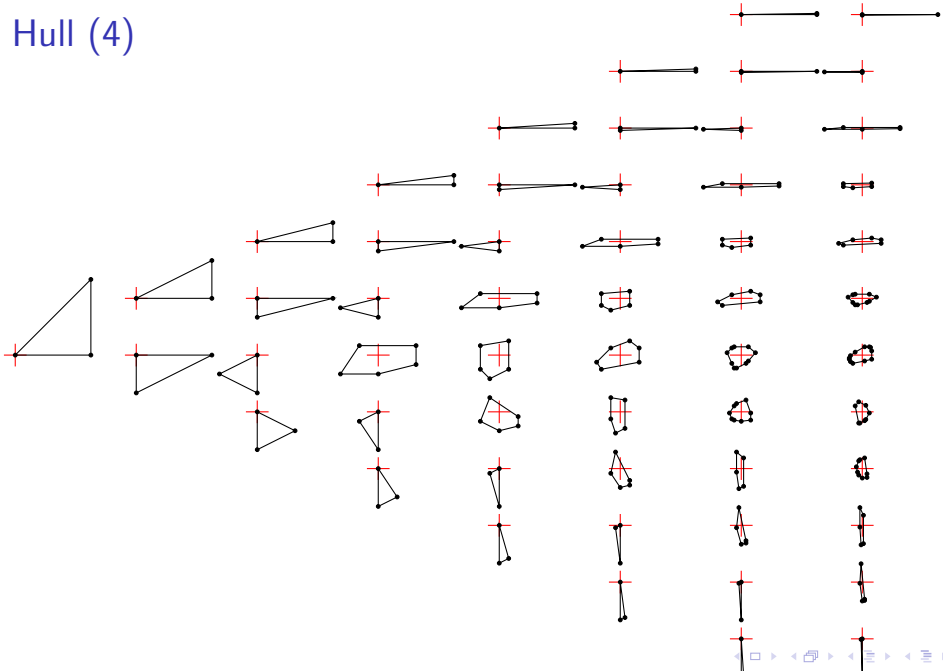


$n = 2,\ \delta_2 = \frac{1}{2}$



$n = 2,\ \delta_2 = \frac{-1}{2}$

Convex Hull (4)



Convex Hull (5)

- With these hulls at hand, how to conclude the required steps?

Convex Hull (5)

- With these hulls at hand, how to conclude the required steps?
- Suppose we compute an N such that each convex hull (corresponding to various δ_N) satisfies

$$\max |x| < 1 \text{ or } \max |y| < 1$$

That is, the hull is thin either horizontally or vertically.

Convex Hull (5)

- With these hulls at hand, how to conclude the required steps?
- Suppose we compute an N such that each convex hull (corresponding to various δ_N) satisfies

$$\max|x| < 1 \text{ or } \max|y| < 1$$

  That is, the hull is thin either horizontally or vertically.
- The state (f_N, g_N) is guaranteed to lie in one of these hulls, implying

$$|f_N| < 1 \text{ or } |g_N| < 1$$

Convex Hull (5)

- With these hulls at hand, how to conclude the required steps?
- Suppose we compute an N such that each convex hull (corresponding to various δ_N) satisfies

$$\max|x| < 1 \text{ or } \max|y| < 1$$

  That is, the hull is thin either horizontally or vertically.
- The state (f_N, g_N) is guaranteed to lie in one of these hulls, implying

$$|f_N| < 1 \text{ or } |g_N| < 1$$

- But f_N is always an odd number, eliminating the possibility of $|f_N| < 1$, and $|g_N| < 1$ implies $g_N = 0$ immediately.

Convex Hull (5)

- With these hulls at hand, how to conclude the required steps?
- Suppose we compute an N such that each convex hull (corresponding to various δ_N) satisfies

$$\max|x| < 1 \text{ or } \max|y| < 1$$

  That is, the hull is thin either horizontally or vertically.
- The state (f_N, g_N) is guaranteed to lie in one of these hulls, implying

$$|f_N| < 1 \text{ or } |g_N| < 1$$

- But f_N is always an odd number, eliminating the possibility of $|f_N| < 1$, and $|g_N| < 1$ implies $g_N = 0$ immediately.
- Hence, the N obtained in the second bullet is the required number of steps for 5-bit inputs.

Stable Hull

We shall briefly mention the essence of Stable Hull here:

- The idea is similar to Convex Hull, we maintain the set of vertices such that (f, g) lies in the hull of those vertices.

Stable Hull

We shall briefly mention the essence of Stable Hull here:

- The idea is similar to Convex Hull, we maintain the set of vertices such that (f, g) lies in the hull of those vertices.
- But, instead of expecting the hull to become thin either horizontally or vertically, we expect the hull shrinks during each `divstep` but keeps the same shape.

Stable Hull

We shall briefly mention the essence of Stable Hull here:

- The idea is similar to Convex Hull, we maintain the set of vertices such that (f, g) lies in the hull of those vertices.
- But, instead of expecting the hull to become thin either horizontally or vertically, we expect the hull shrinks during each `divstep` but keeps the same shape.
- Suppose we found such a hull (named stable hull), with shrinking factor λ. If the input length is 2^b, then $n = (-1/\log(\lambda))b$ steps make $\lambda^n 2^b = 1$ and hence make $g_n = 0$.

Stable Hull

We shall briefly mention the essence of Stable Hull here:

- The idea is similar to Convex Hull, we maintain the set of vertices such that (f, g) lies in the hull of those vertices.
- But, instead of expecting the hull to become thin either horizontally or vertically, we expect the hull shrinks during each `divstep` but keeps the same shape.
- Suppose we found such a hull (named stable hull), with shrinking factor λ. If the input length is 2^b, then $n = (-1/\log(\lambda))b$ steps make $\lambda^n 2^b = 1$ and hence make $g_n = 0$.
- We found such a λ and stable hull and deduced $n = 2.304b$ from that.

Stable Hull

We shall briefly mention the essence of Stable Hull here:

- The idea is similar to Convex Hull, we maintain the set of vertices such that (f, g) lies in the hull of those vertices.
- But, instead of expecting the hull to become thin either horizontally or vertically, we expect the hull shrinks during each `divstep` but keeps the same shape.
- Suppose we found such a hull (named stable hull), with shrinking factor λ. If the input length is 2^b, then $n = (-1/\log(\lambda))b$ steps make $\lambda^n 2^b = 1$ and hence make $g_n = 0$.
- We found such a λ and stable hull and deduced $n = 2.304b$ from that.
- Many questions arise: How to determine λ ? How to find such Stable Hull ? How to prove the main theorem ?

Stable Hull

We shall briefly mention the essence of Stable Hull here:

- The idea is similar to Convex Hull, we maintain the set of vertices such that (f, g) lies in the hull of those vertices.
- But, instead of expecting the hull to become thin either horizontally or vertically, we expect the hull shrinks during each `divstep` but keeps the same shape.
- Suppose we found such a hull (named stable hull), with shrinking factor λ. If the input length is 2^b, then $n = (-1/\log(\lambda))b$ steps make $\lambda^n 2^b = 1$ and hence make $g_n = 0$.
- We found such a λ and stable hull and deduced $n = 2.304b$ from that.
- Many questions arise: How to determine λ ? How to find such Stable Hull ? How to prove the main theorem ?

Stable Hull

We shall briefly mention the essence of Stable Hull here:

- The idea is similar to Convex Hull, we maintain the set of vertices such that (f, g) lies in the hull of those vertices.
- But, instead of expecting the hull to become thin either horizontally or vertically, we expect the hull shrinks during each `divstep` but keeps the same shape.
- Suppose we found such a hull (named stable hull), with shrinking factor λ. If the input length is 2^b, then $n = (-1/\log(\lambda))b$ steps make $\lambda^n 2^b = 1$ and hence make $g_n = 0$.
- We found such a λ and stable hull and deduced $n = 2.304b$ from that.
- Many questions arise: How to determine λ ? How to find such Stable Hull ? How to prove the main theorem ? See the paper for more detail!

Software Optimization

We shall briefly mention one of the optimization tricks:

- Besides iterating `divstep` on (δ, f, g), the program also accumulates the linear transformation applied to f, g.

Software Optimization

We shall briefly mention one of the optimization tricks:

- Besides iterating `divstep` on (δ, f, g), the program also accumulates the linear transformation applied to f, g.
- Suppose the composed linear transformation is denoted by $\begin{bmatrix} u & v \\ r & s \end{bmatrix}$

Software Optimization

We shall briefly mention one of the optimization tricks:

- Besides iterating `divstep` on (δ, f, g), the program also accumulates the linear transformation applied to f, g.
- Suppose the composed linear transformation is denoted by $\begin{bmatrix} u & v \\ r & s \end{bmatrix}$
- It can be proved that, to maintain this linear transformation is to apply linear transformations induced by current `divstep` on pairs (u, r) and (v, s).

Software Optimization

We shall briefly mention one of the optimization tricks:

- Besides iterating `divstep` on (δ, f, g), the program also accumulates the linear transformation applied to f, g.
- Suppose the composed linear transformation is denoted by $\begin{bmatrix} u & v \\ r & s \end{bmatrix}$
- It can be proved that, to maintain this linear transformation is to apply linear transformations induced by current `divstep` on pairs (u, r) and (v, s).
- So, instead of storing f, g, u, v, r, s in six scalar registers, pack them as

$$\texttt{fuv} = f \pmod{2^{20}} + u \cdot 2^{21} + v \cdot 2^{42}$$
$$\texttt{grs} = g \pmod{2^{20}} + r \cdot 2^{21} + s \cdot 2^{42}$$

then one can update them simultaneously. Parallelization in scalar registers!

Software Optimization

We shall briefly mention one of the optimization tricks:

- Besides iterating `divstep` on (δ, f, g), the program also accumulates the linear transformation applied to f, g.
- Suppose the composed linear transformation is denoted by $\begin{bmatrix} u & v \\ r & s \end{bmatrix}$
- It can be proved that, to maintain this linear transformation is to apply linear transformations induced by current `divstep` on pairs (u, r) and (v, s).
- So, instead of storing f, g, u, v, r, s in six scalar registers, pack them as

$$\texttt{fuv} = f \pmod{2^{20}} + u \cdot 2^{21} + v \cdot 2^{42}$$
$$\texttt{grs} = g \pmod{2^{20}} + r \cdot 2^{21} + s \cdot 2^{42}$$

  then one can update them simultaneously. Parallelization in scalar registers!
- There are more assembly-level optimization tricks, including unsaturated multi-limb Montgomery multiplication and vector/scalar hybrid implementation.

Software Optimization

We shall briefly mention one of the optimization tricks:

- Besides iterating `divstep` on (δ, f, g), the program also accumulates the linear transformation applied to f, g.
- Suppose the composed linear transformation is denoted by $\begin{bmatrix} u & v \\ r & s \end{bmatrix}$
- It can be proved that, to maintain this linear transformation is to apply linear transformations induced by current `divstep` on pairs (u, r) and (v, s).
- So, instead of storing f, g, u, v, r, s in six scalar registers, pack them as

$$\texttt{fuv} = f \pmod{2^{20}} + u \cdot 2^{21} + v \cdot 2^{42}$$
$$\texttt{grs} = g \pmod{2^{20}} + r \cdot 2^{21} + s \cdot 2^{42}$$

  then one can update them simultaneously. Parallelization in scalar registers!
- There are more assembly-level optimization tricks, including unsaturated multi-limb Montgomery multiplication and vector/scalar hybrid implementation.

Software Optimization

We shall briefly mention one of the optimization tricks:

- Besides iterating `divstep` on (δ, f, g), the program also accumulates the linear transformation applied to f, g.
- Suppose the composed linear transformation is denoted by $\begin{bmatrix} u & v \\ r & s \end{bmatrix}$
- It can be proved that, to maintain this linear transformation is to apply linear transformations induced by current `divstep` on pairs (u, r) and (v, s).
- So, instead of storing f, g, u, v, r, s in six scalar registers, pack them as

$$\texttt{fuv} = f \pmod{2^{20}} + u \cdot 2^{21} + v \cdot 2^{42}$$
$$\texttt{grs} = g \pmod{2^{20}} + r \cdot 2^{21} + s \cdot 2^{42}$$

  then one can update them simultaneously. Parallelization in scalar registers!
- There are more assembly-level optimization tricks, including unsaturated multi-limb Montgomery multiplication and vector/scalar hybrid implementation. See the paper for more!

Formal Verification

HOL-light and CryptoLine proof scripts are provided in the supplement of the paper.

- We model the notion of "`divstep`", "Stable Hull", and even assembly instruction semantics into a format accepted by proof assistants.

Formal Verification

HOL-light and CryptoLine proof scripts are provided in the supplement of the paper.

- We model the notion of "`divstep`", "Stable Hull", and even assembly instruction semantics into a format accepted by proof assistants.
- The proof assistants, CryptoLine and HOL-light, then prove the correctness of theorems, including the correctness of $n = 2.304b$ and the correctness of the software implementation.

Formal Verification

HOL-light and CryptoLine proof scripts are provided in the supplement of the paper.

- We model the notion of "`divstep`", "Stable Hull", and even assembly instruction semantics into a format accepted by proof assistants.
- The proof assistants, CryptoLine and HOL-light, then prove the correctness of theorems, including the correctness of $n = 2.304b$ and the correctness of the software implementation.
- Some questions arise: How to model those notions? How to model CPU? Won't it take too long to prove the correctness of those statements? How to accelerate verification process?

Formal Verification

HOL-light and CryptoLine proof scripts are provided in the supplement of the paper.

- We model the notion of "`divstep`", "Stable Hull", and even assembly instruction semantics into a format accepted by proof assistants.
- The proof assistants, CryptoLine and HOL-light, then prove the correctness of theorems, including the correctness of $n = 2.304b$ and the correctness of the software implementation.
- Some questions arise: How to model those notions? How to model CPU? Won't it take too long to prove the correctness of those statements? How to accelerate verification process?

Formal Verification

HOL-light and CryptoLine proof scripts are provided in the supplement of the paper.

- We model the notion of "`divstep`", "Stable Hull", and even assembly instruction semantics into a format accepted by proof assistants.
- The proof assistants, CryptoLine and HOL-light, then prove the correctness of theorems, including the correctness of $n = 2.304b$ and the correctness of the software implementation.
- Some questions arise: How to model those notions? How to model CPU? Won't it take too long to prove the correctness of those statements? How to accelerate verification process? See the paper and supplement for more detail!

Thanks, I am ready to take questions!

References I


Thomas Pornin.
Optimized binary GCD for modular inversion.
https://eprint.iacr.org/2020/972, 2020.


John Harrison.
Invert modulo m, software for 64-bit ARM.
https://github.com/awslabs/s2n-bignum/blob/main/arm/generic/bignum_modinv.S, 2021.


John Harrison.
Invert modulo m, software for 64-bit AMD/Intel.
https://github.com/awslabs/s2n-bignum/blob/main/x86_att/generic/bignum_modinv.S, 2021.


Peter Dettman and Pieter Wuille.
This file implements modular inversion.
https://github.com/bitcoin-core/secp256k1/blob/master/src/modinv64_impl.h, 2021.


Kaushik Nath and Palash Sarkar.
Efficient arithmetic in (pseudo-)Mersenne prime order fields.
Advances in Mathematics of Communications 16, 303–348, 2022.


Benjamin Salling Hvass, Diego F. Aranha, and Bas Spitters.
High-assurance field inversion for curve-based cryptography.
CSF 2023, 552–567, 2023.

References II


Marco Bodrato, Marc Glisse, Torbjorn Granlund, and Niels Moller.
GNU MP: The GNU Multiple Precision Arithmetic Library, version 6.3.0.
https://gmplib.org/, 2023.


Russell O'Connor and Andrew Poelstra.
Formal verification of the Safegcd implementation.
https://arxiv.org/abs/2507.17956, 2025.


OpenSSL Project team.
OpenSSL release announcement for 3.6.1, 3.5.5, 3.4.4, 3.3.6, 3.0.19, 1.1.1ze and 1.0.2zn.
https://openssl-library.org/post/2026-01-27-release-announcement/, 2026.