Accelerating and verifying constant-time modular inversion

Daniel J. Bernstein[1,2(✉)], Han-Ting Chen[3(✉)], John R. Harrison[4(✉)], Cesare Huang[2,5(✉)], Gregory Maxwell[6(✉)], Bow-Yaw Wang[2,5(✉)], Pieter Wuille[7(✉)], and Bo-Yin Yang[2,5(✉)]

[1] University of Illinois at Chicago, Chicago, USA
[2] IIS, Academia Sinica, Taipei, Taiwan
djb@cr.yp.to
[3] National Taiwan University, Taipei, Taiwan
hantingchen0212@gmail.com
[4] Amazon Web Services, Seattle, USA
jargh@amazon.com
[5] CITI, Academia Sinica, Taipei, Taiwan
cesarehuang@icloud.com
bywang@iis.sinica.edu.tw
byyang@iis.sinica.edu.tw
[6] Independent Researcher, Rapid City, South Dakota, USA
gmaxwell@gmail.com
[7] Chaincode Labs, New York, USA
pieter@wuille.net

Abstract. This paper presents algorithms to compute the reciprocal of an integer g modulo an odd integer f. This paper also presents software speeds for $f = 2^{255} - 19$ as a case study. As in some previous work, the software timings are independent of g when $0 < g < f$. The speeds here are better than in the previous work. The algorithms and software here are formally verified to work correctly.

This paper's algorithms use the "divstep" iteration introduced by Bernstein and Yang in 2019, but use only 79.9% as many divsteps. This improvement relies on finding a better starting point for the divsteps, namely $(1/2, f, g)$ instead of $(1, f, g)$, and on a better technique to prove upper bounds on the number of divsteps required.

This paper's software takes under 4000 cycles on an Intel Skylake CPU core (with overclocking disabled) for inversion modulo $2^{255} - 19$. This is $1.5\times$ faster than the best previous speeds for constant-time inversion, and $2\times$ faster than the best previous speeds for divstep-based constant-time inversion. This improvement relies on the reduced number of divsteps, better data layout, and better instruction selection.

Keywords: constant-time algorithms, modular inversion, convex hulls, assembly language, formal verification

Author list is in alphabetical order; see https://www.ams.org/profession/leaders/culture/CultureStatement04.pdf. This work was funded by the Intel

J. Daemen and E. Thomé (Eds.): EUROCRYPT 2026, LNCS 16547, pp. 598–627, 2026.
https://doi.org/10.1007/978-3-032-25336-1_21

1 Introduction

If f and g are coprime integers then there is an integer r such that f divides $gr - 1$. This integer r is unique modulo f, and is referred to as the reciprocal of g modulo f, or equivalently the inverse of g modulo f. Computing r from f, g is called modular inversion.

Modular inversion is a fundamental number-theoretic operation with a wide range of cryptographic and non-cryptographic applications. Consider, e.g., the 1978 Rivest–Shamir–Adleman cryptosystem [48, Section V] generating an integer "e" as "the 'multiplicative inverse' of d, modulo $(p-1)\cdot(q-1)$"; or Miller's 1986 elliptic-curve cryptosystem [39, page 420] computing the ratio "$\frac{\phi_n}{f_n^2}$", which in context means "ϕ_n" times the inverse of "f_n^2" modulo a standard prime number.

In many cryptographic applications, modular inversion is applied to secret data, such as "d" and "$(p-1)\cdot(q-1)$" and "f_n^2" in the above examples. Textbook algorithms for modular inversion are based on Euclid's algorithm [22] for computing the greatest common divisor (gcd) and take variable time: some inputs take more time than others, even if one restricts to constant-size inputs f and g (e.g., 256-bit inputs, meaning integers in $\{0, 1, \ldots, 2^{256} - 1\}$), even if f is a standard prime number (as in [39]). This variability sometimes leads to security problems, as illustrated by the attacks mentioned in Section 1.1 below.

These problems have motivated study of constant-time algorithms for gcd and for modular inversion. Our starting point is a 2019 paper by Bernstein and Yang that (focusing on the case of odd f) defines a "division step" as follows [10, Section 8]:

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

Write $(\delta_n, f_n, g_n) = \operatorname{divstep}^n(\delta_0, f_0, g_0)$. If f_0, g_0 are integers with f_0 odd then by induction f_n, g_n are integers with f_n odd, and $\gcd\{f_n, g_n\} = \gcd\{f_0, g_0\}$.

If $\delta_0 = 1$ and $|f_0| \le 2^b$ and $|g_0| \le 2^b$ and $\lfloor (49b + 80)/17 \rfloor \le n$ then $g_n = 0$ by [10, Theorem 11.2]. One then has $\gcd\{f_0, g_0\} = |f_n|$. In the case $\gcd\{f_0, g_0\} = 1$, one also finds the reciprocal of g_0 modulo f_0 by tracking the coefficients of f_n, g_n as linear combinations of f_0, g_0.

Rewriting $\operatorname{divstep}(\delta, f, g)$ as $(1 + \delta - 2st\delta, f + (g-f)st, (g + (1-2s)tf)/2)$, where $s = [\delta > 0]$ and $t = g \bmod 2$, converts the above gcd computation into

Crypto Frontiers Research Center; by U.S. National Science Foundation grant 1913167; by the Taiwan's Executive Yuan Data Safety and Talent Cultivation Project (AS-KPQ-109-DSTCP); by Academia Sinica Grand Challenge Project (AS-GCP-114-M01); and by Taiwan National Science and Technology Council grants 114-2221-E-001-014-MY3, 113-2221-E-001-023-MY3, and 114-2221-E-001-022-MY3. "Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation" (or other funding agencies). Permanent ID of this document: `405f00df60bbdfac3971584c6fb662726db6303c`. Date: 2026-02-23.

a series of additions, subtractions, divisions by 2, bottom-bit extractions, δ-sign extractions, and multiplications by bits. The whole sequence of operations in computing (δ_n, f_n, g_n) for $n = \lfloor (49b + 80)/17 \rfloor$ is then determined entirely by b, taking constant time for constant b if each operation is carried out in constant time. Similar comments apply to modular inversion.

This approach to computing gcd and modular reciprocals avoids leaking secret inputs through timings. But this approach also raises questions: whether [10, Theorem 11.2], which relies on a large computation, is correct; how quickly the resulting modular-inversion software runs, in applications concerned with performance; whether the software, including any speed-induced complications, is correct; and whether divsteps are the best approach in the first place.

1.1 Competing techniques

A traditional Euclid-style sequence of divisions involves conditional branches to decide when to stop each division and when to stop the overall computation. This produces input-dependent timings, exploited by, e.g., the timing attack from 2013 Strenzke [52] (for polynomials). Small mistakes can easily produce infinite loops for some inputs, exploited (for integers) by the Windows denial-of-service attack from 2019 Ormandy reported in [55]. See also [54] finding a variable-time Euclid inverter (for integers) in Intel's IPP library. Replacing each division with a constant sequence of operations makes the computation much slower.

"Binary" gcd techniques (see, e.g., 1967 Stein [51] and 1995 Kaliski [36]) are easier to convert into constant-time computations (see, e.g., [14], [49], and [47]), although care is still required: see, e.g., the timing attack from 2018 Aldaya–García–Tapia–Brumley [2] (for integers). These techniques look similar to divsteps at first glance: they are built from additions, divisions by 2, sign extractions, etc. The critical difference is that the "binary" sign extractions are on big integers, usually $\Theta(b)$-bit integers, for example to check whether $f < g$, whereas divstep carries out sign extractions only on the small integer δ, which has just $O(\log b)$ bits.

One can think of δ as a very rough estimate of $\log_2|f| - \log_2|g|$. Working with this small integer δ, rather than the actual sizes of f and g, streamlines each divstep and enables the efficient "jumps" described in [10]. On the other hand, it is easy to prove that $2b$ iterations suffice for the algorithm from [51]; the much more complicated proof from [10] uses about $49b/17 \approx 2.882b$ divstep iterations.

Algorithms from pre-2000 Gosper (see [37, pages 355 and 645–646]), 2020 Pornin [47], and 2021 Harrison [30, 31] use the following intermediate possibility between the simplest-sounding "binary" gcd (full-length comparisons at each iteration) and divsteps (single-word δ as a proxy for relative sizes throughout). Compute single-word integers close to $f/2^i$ and $g/2^i$ as size proxies, for an appropriate choice of i. Use these proxies, along with the bottom bits of f and g, to control a limited number of iterations, during which the proxies may drift farther away from $f/2^i$ and $g/2^i$. Then recompute the actual sizes of f and g to regenerate accurate proxies for a new choice of i. Repeat as necessary.

The number of iterations between full-length updates of f and g is 32 in [47, 18 August 2020 version], 31 in [47, 23 August 2020 version] (which says that the previous version had a "gap in the proof" and an example "for which the algorithm failed" but that the revised algorithm and proof "are believed correct"), and 58 in the `bignum_modinv` software from [30] and [31] (which have computer-checked proofs of full functional correctness, including all necessary properties of the size proxies). The different numbers arise mainly from a different layout of algorithm words in CPU words—this is an important speed issue that we return to in Section 5—but also from differences in how the drift is analyzed.

1.2 Contributions of this paper

This paper accelerates divstep-based modular inversion, both at the algorithmic level and at the software level. Furthermore, to ensure correctness, this paper formally verifies the resulting algorithms and software. The resulting speeds for verified constant-time divstep-based inversion software are faster than any previous speeds for constant-time inversion software, whether or not verified, whether or not divstep-based.

Case study. We use inversion modulo $f = 2^{255} - 19$ on 64-bit Intel CPUs as a concrete case study to support software-speed comparisons. This case has a large word size, fast multiplier, sparse prime f, and only a few words in the prime; Fermat inversion $g \mapsto g^{f-2} \bmod f$ benefits from these features, as noted in [10]. This prime f is also widely deployed (see [15] and [16] regarding deployment of X25519 [4] and Ed25519 [7] respectively) and a frequent optimization target in the literature. Ed25519 signing takes only about 30000 cycles on Intel Skylake CPUs (see [8, Appendix E] and the references therein). We also take a Skylake CPU (an Intel Xeon E3-1220 v5) to maximize comparability.

Table 1.2.1 presents cycle counts for this Skylake (labeled `samba`) collected by our `testinv25519` toolkit (part of the supplement to this paper available from `https://gcd.cr.yp.to`), which measures our software and the previous state-of-the-art constant-time inverters modulo $2^{255} - 19$: namely, a Fermat inverter from 2022 Nath–Sarkar [42], divstep-based inverters from 2021 Dettman–Wuille [19] and 2023 Hvass–Aranha–Spitters [34], and faster inverters from [47] and [31] summarized in Section 1.1. Our software also supports 64-bit ARM, where we compare it on a Cortex-A76 CPU (Broadcom BCM2712 in a Raspberry Pi 5, labeled `pi5`) to [30], [19], and [34].

The `testinv25519` toolkit and Table 1.2.1 also include some other inverters for comparison: a variable-time divstep-based inverter from [19] verified by 2025 O'Connor–Poelstra [44]; two inverters from OpenSSL 3.6.1 [45], namely `BN_mod_inverse_no_branch` (which, despite its name, is variable-time) and `BN_mod_inverse`; and the inverters from GMP 6.3.0 [12], namely constant-time `mpn_sec_invert` and variable-time `mpn_gcdext`. Also, to illustrate the generality of our techniques, Table 1.2.2 presents cycle counts for inversion modulo $2^{521} - 1$; but beware that $2^{521} - 1$ has received less attention in the literature than $2^{255} - 19$.

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
[19]+[44] (divstep)	yes	no	4332.25	4447.68	4585.41	samba
[47] (semi-binary)	no	yes	5766.88	5788.78	5811.31	samba
[31] (semi-binary)	yes	yes	6012.25	6031.28	6057.10	samba
mpn_gcdext [12]	no	no	6570.58	6757.64	6939.16	samba
[19] (divstep)	no	yes	8844.80	8867.19	8890.62	samba
[42] (Fermat)	no	yes	8966.81	8975.10	8984.81	samba
[34] (divstep)	yes	yes	18475.84	18507.89	18541.16	samba
BN_mod_inverse [45]	no	no	57993.47	58850.63	59762.12	samba
mpn_sec_invert [12]	no	yes	74701.91	74954.04	75208.72	samba
BN_mod_inverse_no_branch [45]	no	no	72252.57	75197.39	78521.38	samba
this paper (divstep; Inverter 1A)	yes	yes	3955.25	3963.88	3973.15	pi5
mpn_gcdext [12]	no	no	4440.62	4544.22	4646.28	pi5
this paper (divstep; Inverter 2A)	yes	yes	5588.25	5600.96	5612.74	pi5
[19]+[44] (divstep)	yes	no	5915.92	6054.42	6191.94	pi5
[30] (semi-binary)	yes	yes	6660.37	6668.38	6677.19	pi5
[19] (divstep)	no	yes	11980.70	12005.45	12031.28	pi5
[34] (divstep)	yes	yes	20133.92	20154.97	20176.29	pi5
BN_mod_inverse [45]	no	no	57318.26	58021.03	58787.02	pi5
mpn_sec_invert [12]	no	yes	59600.23	65691.68	66700.73	pi5
BN_mod_inverse_no_branch [45]	no	no	78591.13	81971.99	85489.24	pi5

Table 1.2.1. Cycle counts on one core of an Intel Skylake (samba) and one core of an ARM Cortex-A76 (pi5) for various inverters modulo $2^{255} - 19$ compiled with clang -O3 -march=native. "Verif": Software is formally verified to work correctly. "Val": Software passes a valgrind test, i.e., avoids data flow from the input to branches and array indices. (Our inverters meet the stronger condition of being constant-time—for example, avoiding division instructions—but the "Val" column reports what was tested by our testinv25519 toolkit.) Numeric columns are stabilized quartiles as defined in [6]. Note that caches, out-of-order execution, etc. create variations in measurements even for code that has no data flow from inputs to timings; see [24].

It does not seem to have been pointed out before that recent constant-time code outperforms mpn_gcdext on some CPUs.[8] One should not underestimate the cost of random accesses, including branches.[9]

[8] [34, 2021 version, Table 2; 2024 version, Table II] reported only about 3000 cycles for GMP on Skylake, but this is unrealistic. We had posted a speed test that uses our constant-time code to repeatedly invert an input in place; [34] adapted this to measure GMP's variable-time code; but the repeated inputs inflate the success probability of the branch predictor for variable-time code. This paper's testinv25519 flips a few bits after each inversion; this avoids the pattern of repetitions, and we have not detected performance discrepancies between these inputs and inputs with further pseudorandomization. We contacted the authors of [34]; they concurred with our diagnosis, and posted an online revision [34, 2025 version] reporting about 6600 cycles.

[9] Whenever we outperform a variable-time inverter, obviously we also outperform a "masked" (or "blinded") version of the inverter that first multiplies the input by a

Software	Verif	Val	[1/8, 3/8]	[3/8, 5/8]	[5/8, 7/8]	Host
this paper (divstep; Inverter 2X)	yes	yes	10348.29	10386.80	10438.57	samba
`mpn_gcdext` [12]	no	no	13298.28	13579.98	13835.04	samba
[31] (semi-binary)	yes	yes	17645.28	17679.66	17729.85	samba
[42] (Fermat)	no	yes	52017.47	52059.78	52112.36	samba
[34] (divstep)	yes	yes	55124.18	55172.25	55218.55	samba
`BN_mod_inverse` [45]	no	no	112534.29	113807.63	115435.45	samba
`BN_mod_inverse_no_branch` [45]	no	no	164733.11	168985.43	174132.75	samba
`mpn_sec_invert` [12]	no	yes	281526.70	281752.06	282034.27	samba
`mpn_gcdext` [12]	no	no	9769.48	9906.42	10044.27	pi5
this paper (divstep; Inverter 2A)	yes	yes	17385.92	17394.99	17415.72	pi5
[30] (semi-binary)	yes	yes	23299.46	23313.10	23329.84	pi5
[34] (divstep)	yes	yes	62209.65	62248.66	62287.81	pi5
`BN_mod_inverse` [45]	no	no	115977.54	116961.04	118236.68	pi5
`BN_mod_inverse_no_branch` [45]	no	no	177065.03	181732.14	187531.68	pi5
`mpn_sec_invert` [12]	no	yes	205583.04	205984.71	206995.25	pi5

Table 1.2.2. Cycle counts as in Table 1.2.1, but for inverters modulo $2^{521} - 1$ rather than inverters modulo $2^{255} - 19$.

Algorithmic-level contributions. Our main theorem states that if $\delta_0 = 1/2$ and $0 \leq g_0 \leq f_0 \leq 2^b$ and $\lceil (9437b + 1)/4096 \rceil \leq n$ then $g_n = 0$. We provide a computer-checked proof (part of the supplement to this paper), specifically a proof checked by HOL Light. The ratio $9437/4096 \approx 2.304$ is better than the aforementioned ratio $49/17 \approx 2.882$ from [10].

The definitions in [10] required $\delta \in \mathbb{Z}$. Section 2 generalizes to $\delta \in \mathbb{R}$ and explains the discovery that $\delta_0 = 1/2$ is better than $\delta_0 = 1$. Section 3 presents a technique that, given n and δ_0, computes a useful bound H along with a proof that $g_n = 0$ if $0 \leq g_0 \leq f_0 \leq H$. Section 4 presents a technique to replace this sequence of proofs for each n with a single proof that handles all n simultaneously, the proof of our main theorem.

Software-level contributions. Section 5 explains how we quickly compute each divstep. This, along with the reduced number of divsteps, is critical for our speeds in Table 1.2.1: 1.5× better than previous constant-time inverters, and 2× better than previous divstep-based constant-time inverters. Finally, Section 6 presents our verification of software correctness.

random number and then multiplies the output by the same number. Furthermore, many variable-time inverters are not verified; masking would not have stopped the attack reported in [55], for example, whereas our software is immune to that attack and to subtler bugs. Masked variable-time inverters would also force timing-attack audits to include audits of random-number generation. Adding masking *on top of our inverters* could be useful as a countermeasure against other side-channel attacks, but evaluating this is outside the scope of this paper.

2 Generalizing and optimizing divsteps

This section presents two different motivations for considering $\delta \in \mathbb{R}$ rather than just $\delta \in \mathbb{Z}$. This section also reports quick computer calculations that, for very small sizes of (f, g), show by brute force that $\delta_0 = 1/2$ is a better choice than $\delta_0 = 1$. See Section 3 for how to efficiently analyze larger sizes of (f, g), and Section 4 for a proof that handles all sizes of (f, g) simultaneously.

2.1 Motivation 1: How to measure size

The paper [10] considers polynomial gcd before integer gcd. The traditional polynomial-gcd approach is the Euclid–Stevin algorithm, which writes down a sequence of remainders $R_0, R_1, R_2 = R_0 \bmod R_1, R_3 = R_1 \bmod R_2$, etc., starting from the inputs R_0, R_1. A traditional computation of $R_i \bmod R_{i+1}$ takes a variable number of polynomial subtractions, the number depending on the difference in degrees between R_i and R_{i+1}.

For, e.g., $R_0 = 31x^9 + 415x^8 + \cdots$ and $R_1 = x^8 + 10x^7 + \cdots$: a first subtraction clears the coefficient of x^9, obtaining $R_0 - 31xR_1 = 105x^8 + \cdots$; a second subtraction clears the coefficient of x^8, obtaining $R_2 = R_0 \bmod R_1 = R_0 - 31xR_1 - 105R_1$ of degree strictly below 8; one then swaps R_2, R_1 into R_1, R_2 and continues with a division of R_1 by R_2.

More generally, take any R_0, R_1 with $\deg R_0 > \deg R_1 \geq 0$. Then computing $R_0 \bmod R_1$ corresponds by [10, Theorem C.1] to a sequence of $2n$ divsteps where $n = \deg R_0 - \deg R_1$, using the definition of polynomial divsteps in [10, Section 3]. Inside this correspondence, δ starts with 1 before the divsteps, and then moves to $2, \ldots, n, -(n-1), -(n-2), \ldots, -1, 0, 1$, at each moment matching the difference in known upper bounds on degrees of two polynomials under consideration. The correspondence also reverses the order of coefficients of each polynomial: divsteps clear bottom coefficients rather than top coefficients.

What happens when polynomial gcd is replaced with integer gcd? Inside a gcd computation, one wants to track which input is larger, to be able to reduce the larger input by subtracting off the smaller input. For polynomials of degree at most b, tracking input sizes means tracking degrees, information that fits into $O(\log b)$ bits. For integers bounded by 2^b, size tracking is inherently more difficult: deciding which of two integers is larger sometimes requires inspecting $\Theta(b)$ bits. Using log or $\log_2$ to track sizes does not eliminate this difficulty.

As mentioned in Section 1, one can think of δ in integer divsteps as a fast estimate of $\log_2 |f| - \log_2 |g|$. This does not mean that it is optimal to remove *all* bits of the estimate after the decimal point, i.e., to require δ to be an integer. Mathematically, it is natural to allow δ to be a real number. One can then ask whether this generalization allows better optimization: perhaps a non-integer choice of δ_0 is more efficient than an integer choice.

For each $D \in \mathbb{Z}$, all choices of δ_0 strictly between D and $D + 1$ produce equivalent results, so it suffices to consider $\delta_0 = D + 1/2$. To summarize, beyond considering choices of δ_0 in $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$, it is natural to consider choices of δ_0 in $\{\ldots, -3/2, -1/2, 1/2, 3/2, \ldots\}$.

2.2 Motivation 2: Testing

A standard way to check whether software works is to test it.[10] How effective is a test? A common way to answer this question is to intentionally introduce small modifications into the software and to see whether the resulting bugs are caught by the test. These modifications are dubbed "mutations" in [18]. The point here is that bugs often arise from programmers accidentally introducing mutations, so it is useful for tests to be able to catch these bugs.

In the case of gcd computation, consider a test that checks whether software correctly computes gcd for randomly chosen pairs of b-bit inputs. We observe experimentally that, as expected, the test accepts our n-divstep software, when n is chosen in light of a proof that n divsteps always compute gcd on b-bit inputs. The test also rejects various examples of mutated software, demonstrating that the test has value. However, the test does not catch, inter alia, mutated software that includes the following three mutations simultaneously: mutate the $\delta > 0$ test into $\delta \geq 0$; mutate $1-\delta$ into $-\delta$; mutate the initialization $\delta_0 = 1$ into $\delta_0 = 0$. A closer look—see Section 2.3—indicates that this does not reflect limits upon the value of the test; the mutated computation works correctly, and continues to work correctly even when n is chosen somewhat smaller.[11]

To see that this is related to non-integer choices of δ, simply define $\delta' = \delta + 1/2$. Then this mutated divstep on (δ, f, g) corresponds to the original divstep on (δ', f, g). Specifically, stepping from δ to $-\delta$ corresponds to stepping from δ' to $1-\delta'$; stepping from δ to $1+\delta$ corresponds to stepping from δ' to $1+\delta'$; and the test $\delta \geq 0$ corresponds to the test $\delta' > 0$. The starting point $\delta = 0$ corresponds to $\delta' = 1/2$. To summarize, this mutated computation is equivalent to original divsteps starting from $(1/2, f_0, g_0)$.

An equivalent set of mutations is as follows: mutate $1 - \delta$ into $2 - \delta$, and mutate $1 + \delta$ into $2 + \delta$, while keeping $\delta_0 = 1$. To see that these mutations are equivalent to original divsteps starting from $1/2$, simply define $\delta' = \delta/2$. There are more ways to represent the same computation; we return to this theme in Section 5, which considers software optimizations.

We caution the reader that some inputs need more divsteps than most inputs do, so another type of mutation, namely further reducing n, can pass tests for many randomly chosen inputs while being incorrect for some other inputs. Useful responses include finding worst cases to use as tests (see Section 4.4) and verifying software (see Section 6).

[10] Proofs have the advantage of covering all inputs, but tests have the advantage of being computerized, with the computer systematically checking each step. Even when software has a *computer-checked* proof with both advantages, it is conceivable that a software bug hides behind a proof bug, that the proof bug is not caught because of a flaw in the checking mechanism, and that a test catches the software bug. It is difficult to recommend against tests.

[11] The $\delta \geq 0$ mutation on its own already achieves part of this improvement.

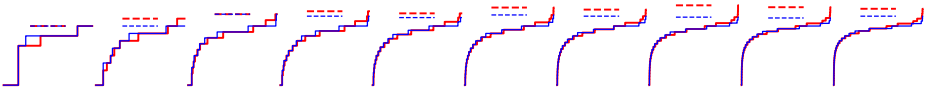

Fig. 2.3.1. For $b \in \{3, 4, \ldots, 12\}$ in order: Transposed cumulative distribution function of the number of divsteps needed to reach $(\ldots, \ldots, 0)$. Red curve starts with $(1, f, g)$ for all (f, g) with $f \in \{2^{b-1} + 1, 2^{b-1} + 3, \ldots, 2^b - 1\}$ and $g \in \{0, 1, \ldots, f - 1\}$. Blue curve starts with $(1/2, f, g)$ for the same pairs (f, g). Vertical scale is proportional to b. Dashed red and blue lines indicate maximum number of divsteps for the red and blue curves respectively.

2.3 Brute-force assessment

For any particular (δ_0, f_0, g_0), iterating divstep produces (δ_n, f_n, g_n) for each n in turn, quickly identifying the minimum n for which $g_n = 0$; in other words, quickly identifying the number of divsteps required for this input. Repeating this process with $(f_0, g_0) = (f, g)$ for all small choices of (f, g) identifies the maximum number of divsteps required across that pool of (f, g) for this choice of δ_0. Repeating all of that for $\delta_0 = 1/2$ and $\delta_0 = 1$ then shows which choice has a smaller maximum number of required divsteps across that pool of (f, g).

Figure 2.3.1 includes ten graphs showing the results for different sizes of (f, g). Specifically, the graphs are for pool 3, pool 4, and so on through pool 12, where pool b consists of all pairs (f, g) of integers with f odd, $2^{b-1} < f < 2^b$, and $0 \leq g < f$. The graphs are transposed cumulative distribution functions: each vertical axis is the number of divsteps required (divided by $3b$), and each horizontal axis (scaled to $[0, 1]$) gives equal space to each pair (f, g), sorted by the vertical axis.

For pool 3, $\delta_0 = 1/2$ and $\delta_0 = 1$ have the same maximum number of divsteps. The same is true for pool 5. For pool 4 and for pools 6 through 12, $\delta_0 = 1/2$ has a smaller maximum number of divsteps than $\delta_0 = 1$.

For most of these b, the graphs for $\delta_0 = 1$ and $\delta_0 = 1/2$ are much closer in average than in maximum. For example, for $b = 12$, the averages are approximately 22.55 and 22.26 respectively, while the maxima are 31 and 28 respectively. A computation using only slightly more than the average number of divsteps will fail for some inputs, and it is not obvious how to extrapolate from the average to the maximum.

One might guess from these graphs that $\delta_0 = 1/2$ is better than $\delta_0 = 1$ for each $b \geq 6$. The calculations reported above prove this for $6 \leq b \leq 12$. The calculations were on the scale of 1 second, certainly not the limit of feasibility; but we do not push brute force further, since subsequent sections explain efficient ways to handle much larger values of b.

3 Convex hulls: analyzing each iteration count

This section presents a technique that, given $\delta_0 \in \mathbb{Q}$ and an integer $b \geq 0$, computes an integer $n \geq 0$ such that $(\delta_n, f_n, g_n) = \text{divstep}^n(\delta_0, f_0, g_0)$ has $g_n = 0$ for all integers f_0, g_0 with f_0 odd and $0 \leq g_0 \leq f_0 \leq 2^b$.

One can straightforwardly find the minimum such n by brute force, as in Section 2.3, using time roughly 2^{2b}. The advantage of the technique in this section is that it is much faster. For example, we carried out these computations for $b = 256$, showing that 723 divsteps suffice for $\delta_0 = 1$ and that 590 divsteps suffice for $\delta_0 = 1/2$.

This section is organized as follows. Section 3.2 explains how to compute, for each n in turn and for each $\delta \in D_n$ (with D_n defined in Section 3.1), finite subsets $S_{n,\delta}$ of $\mathbb{Q} \times \mathbb{Q}$ having the following property: if $H > 0$ and $0 \le g_0 \le f_0 \le H$ then $(f_n/H, g_n/H) \in \operatorname{Hull} S_{n,\delta_n}$, where $\operatorname{Hull} X$ means the convex hull of X. Section 3.3 gives numerical examples of these hulls. Section 3.4 explains an endgame that identifies H for which any $(f_n/H, g_n/H) \in \operatorname{Hull} S_{n,\delta_n}$ has $g_n = 0$, for example concluding that 724 divsteps suffice for $\delta_0 = 1$ and $b = 256$. Section 3.5 explains a slightly better endgame, for example replacing 724 with 723. Section 3.6 analyzes and improves the cost of these hull computations.

One should not think that this section uses minimal time to obtain a proven bound. Given b, one can simply compute $n = \lfloor (49b + 80)/17 \rfloor$ and apply [**10**, Theorem 11.2] to conclude that n divsteps suffice for $\delta_0 = 1$: e.g., $\lfloor (49 \cdot 256 + 80)/17 \rfloor = 742$ divsteps suffice for $\delta_0 = 1$ and $b = 256$. The one-time computation to prove this theorem is shared across b, and is lower than the cost of this section's technique for all sufficiently large b.

The big advantage of this section's technique over [**10**, Theorem 11.2] is the reduced number of divsteps shown to be sufficient, such as 590 divsteps for $\delta_0 = 1/2$ and $b = 256$. Section 4 combines these advantages, showing how to merge this section's proofs for each b into a single proof that handles all b simultaneously while obtaining approximately the same divstep count as in this section.

3.1 The range of δ

As a preliminary matter, note that $\delta_n \in D_n$, where D_n is defined as follows: $D_0 = \{\delta_0\}$, and $D_n = \{n - 2i \pm \delta_0 : 0 \le i < n\}$ for $n \ge 1$.

Consequently, for each choice of δ_0 and each $n \ge 1$, there are at most $2n$ choices of δ_n. There are only $n + 1$ choices in the case $\delta_0 = 1$, and only n choices in the case $\delta_0 = 0$, but the gap between n and $2n$ does not matter for anything stated below.

3.2 Constructing hulls for divsteps

We now construct finite subsets S_{n,δ_n} of $\mathbb{Q} \times \mathbb{Q}$ having the property described at the beginning of this section. Specifically, for $n = 0$, define $S_{0,\delta_0} = \{(0,0), (1,0), (1,1)\}$. Then, for each $n \ge 0$, recursively define $S_{n+1,\epsilon}$ for $\epsilon \in D_{n+1}$ as the set of corners of $\operatorname{Hull} T_{n+1,\epsilon}$, where $T_{n+1,\epsilon}$ is the minimum set satisfying

the following conditions:

$$\text{if } \delta > 0 \text{ and } \delta \in D_n \text{ then } M_{-1}(S_{n,\delta}) \subseteq T_{n+1,1-\delta} \text{ where } M_{-1}(x,y) = \left(y, \frac{y-x}{2}\right);$$

$$\text{if } \delta \leq 0 \text{ and } \delta \in D_n \text{ then } M_1(S_{n,\delta}) \subseteq T_{n+1,1+\delta} \text{ where } M_1(x,y) = \left(x, \frac{y+x}{2}\right);$$

$$\text{if } \delta \in D_n \text{ then } M_0(S_{n,\delta}) \subseteq T_{n+1,1+\delta} \text{ where } M_0(x,y) = \left(x, \frac{y}{2}\right).$$

There are only finitely many $\delta \in D_n$, only finitely many $(x,y) \in S_{n,\delta}$, and only finitely many conditions in this list, so each $T_{n+1,\epsilon}$ is finite, so the set $S_{n+1,\epsilon}$ of corners is also finite as claimed.[12]

A straightforward induction now obtains the desired property, namely that if $H > 0$ and $0 \leq g_0 \leq f_0 \leq H$ then $(f_n/H, g_n/H) \in \operatorname{Hull} S_{n,\delta_n}$. The point is that if $(f_n/H, g_n/H) \in \operatorname{Hull} S_{n,\delta_n}$ then, for any linear transformation M, one has $M(f_n/H, g_n/H) \in \operatorname{Hull} M(S_{n,\delta_n})$. Take $M = M_{-1}$ if $\delta_n > 0$ and g_n is odd, or $M = M_1$ if $\delta_n \leq 0$ and g_n is odd, or $M = M_0$ if g_n is even, to see that $(f_{n+1}/H, g_{n+1}/H) \in \operatorname{Hull} T_{n+1,\delta_{n+1}} = \operatorname{Hull} S_{n+1,\delta_{n+1}}$. For the base case, $0 \leq g_0/H \leq f_0/H \leq 1$ so $(f_0/H, g_0/H) \in \operatorname{Hull} S_{0,\delta_0}$.

3.3 Examples of these hulls

Figure 3.3.1 displays the hulls $S_{n,\delta}$ for all small n for $\delta_0 = 1$. Figure 3.3.2 does the same for $\delta_0 = 1/2$.

Within these figures, the hull pictures are shrinking in area as n increases (i.e., as one moves from left to right). The shrinkage looks faster for $\delta_0 = 1/2$ than for $\delta_0 = 1$; Section 3.4 gives numerical examples of this for much larger n. One might also guess that the hulls shrink exponentially with n; Section 4 quantifies and proves this for $\delta_0 = 1/2$.

3.4 A simple endgame

Consider the following situation: one has found n such that, for each $\delta \in D_n$, one has $\max\{|x| : (x,y) \in S_{n,\delta}\} < 1/H$ or $\max\{|y| : (x,y) \in S_{n,\delta}\} < 1/H$.

In this situation, if f_0, g_0 are integers with f_0 odd and $0 \leq g_0 \leq f_0 \leq H$, then $(f_n/H, g_n/H) \in \operatorname{Hull} S_{n,\delta_n}$, so $|f_n/H| < 1/H$ or $|g_n/H| < 1/H$; so $|f_n| < 1$ or $|g_n| < 1$; but f_n and g_n are integers; so $f_n = 0$, which is impossible since f_n is odd, or $g_n = 0$ as desired. In short, n divsteps suffice to guarantee $g_n = 0$ when $0 \leq g_0 \leq f_0 \leq H$.

Note that the definition of $S_{n,\delta}$ is independent of H. For each n in turn, we compute $S_{n,\delta}$ and then compute H_n, where we define H_n as the largest integer H creating the situation above, i.e., the largest integer H such that, for all $\delta \in D_n$,

$$H < 1/\min\{\max\{|x| : (x,y) \in S_{n,\delta}, x \neq 0\}, \max\{|y| : (x,y) \in S_{n,\delta}, y \neq 0\}\}.$$

[12] More generally, one can handle other choices of starting inequalities for (f_0, g_0) or modifications of the divstep iteration, by appropriately modifying the definition of $S_{n,\delta}$. For example, taking $-f_0/2 \leq g_0 \leq f_0/2 \leq H/2$ would correspond to taking $S_{0,\delta_0} = \{(0,0), (1,-1/2), (1,1/2)\}$.

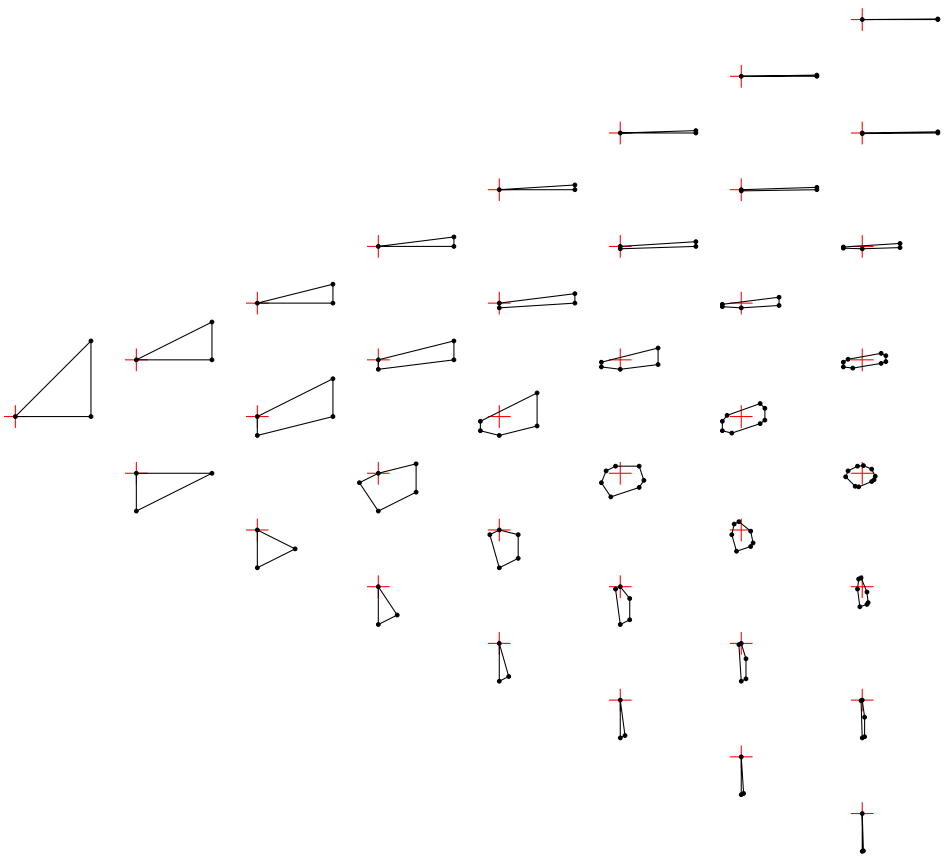

Fig. 3.3.1. For $n = 0$ (first column), $n = 1$ (second column), $n = 2$ (third column), etc.: Hulls $S_{n,\delta}$ when $\delta_0 = 1$, in increasing order of δ from bottom to top. Each hull has its own axes but is on the same scale. Each axis (red) is drawn from -0.15 to 0.15.

For $\delta_0 = 1$, we find $\min\{n : H_n \geq 2^{255}\} = 722$ and $\min\{n : H_n \geq 2^{256}\} = 724$; in hex, H_{724} is

`1030596cf6d817d1357f908ef70cdb00b38d047fbba852139babb6c8646fb15b2`.

For $\delta_0 = 1/2$, we find $\min\{n : H_n \geq 2^{255}\} = 588$ and $\min\{n : H_n \geq 2^{256}\} = 590$.

3.5 An improved endgame

Consider the following conditions on a finite subset S of $\mathbb{Q} \times \mathbb{Q}$: each $(x, y) \in \operatorname{Hull} S$ has $|x| < 3/H$ and $|y| < 2/H$; and $\operatorname{Hull} S$ does not contain the points $(1/H, 1/H)$, $(1/H, -1/H)$, $(-1/H, 1/H)$, or $(-1/H, -1/H)$. If these conditions are satisfied, and f, g are integers with f odd and $(f/H, g/H) \in \operatorname{Hull} S$, then

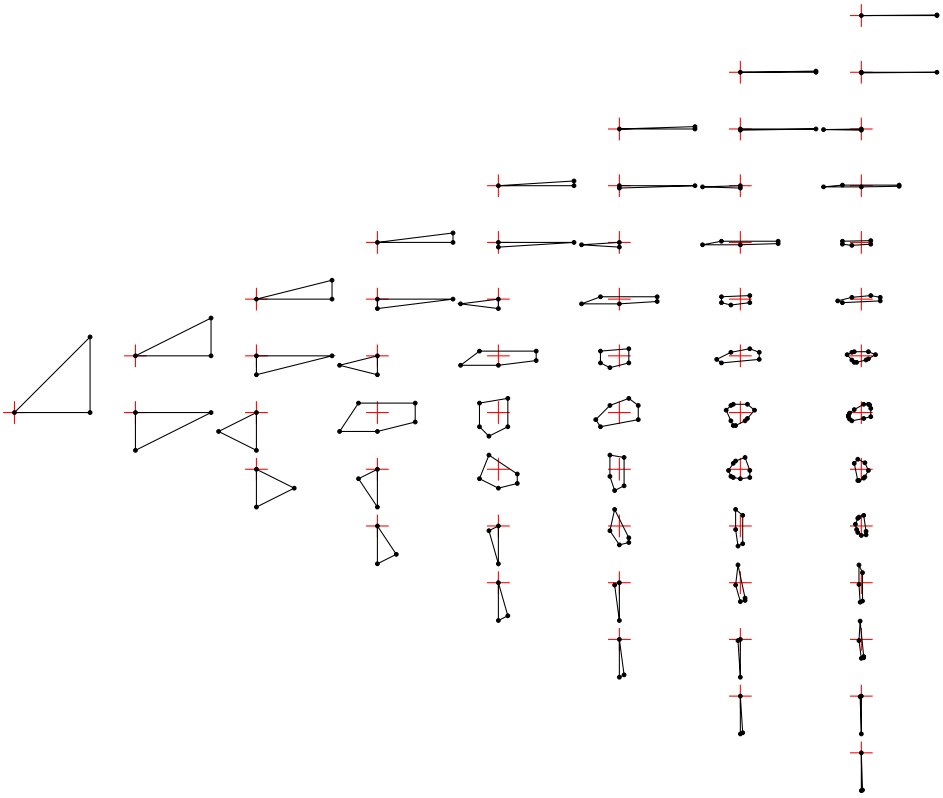

Fig. 3.3.2. For $n = 0$ (first column), $n = 1$ (second column), $n = 2$ (third column), etc.: Hulls $S_{n,\delta}$ when $\delta_0 = 1/2$, in increasing order of δ from bottom to top. Each hull has its own axes but is on the same scale. Each axis (red) is drawn from -0.15 to 0.15.

$g = 0$. Indeed, $|x| < 3/H$ forces $|f| < 3$, so $f \in \{-1, 1\}$ since f is odd; $|y| < 2/H$ forces $|g| < 2$, so $g \in \{-1, 0, 1\}$; the last condition then rules out the case $g \neq 0$.

It is easy to compute, given S as input, the cutoff for H satisfying these conditions. The first and second conditions are equivalent to $H < 3/X$ and $H < 2/Y$ respectively where

$$X = \max\{|x| : (x, y) \in \operatorname{Hull} S\} = \max\{|x| : (x, y) \in S\},$$
$$Y = \max\{|y| : (x, y) \in \operatorname{Hull} S\} = \max\{|y| : (x, y) \in S\}.$$

For the third condition, intersect the edges of Hull S with the line $y = x$ to find the (potentially empty) interval of $\lambda \in \mathbb{R}$ for which $(\lambda, \lambda) \in \operatorname{Hull} S$, and similarly use $y = -x$ for $(\lambda, -\lambda)$.

These cutoffs supplement, and do not supersede, the conditions from Section 3.4: our improved endgame is to accept any hull that meets the cutoffs in the previous paragraph *or* that meets the cutoffs from Section 3.4. Figure 3.5.1 depicts the condition from Section 3.4 (left), namely $X < 1/H$ or $Y < 1/H$, and

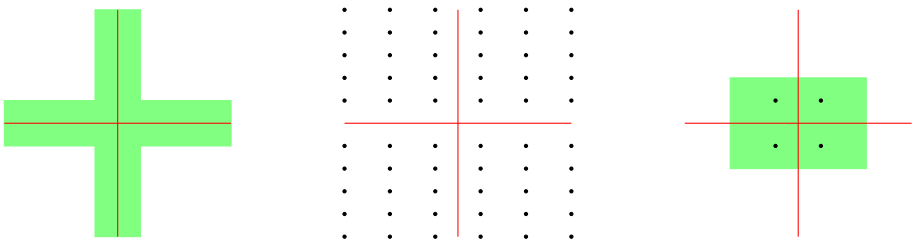

Fig. 3.5.1. Left diagram: Points (x, y) with $|x| < 1$ or $|y| < 1$. Middle diagram: Points (x, y) where x, y are integers with x odd and y nonzero. Right diagram: Points (x, y) with $|x| < 3$ and $|y| < 2$. Each diagram has its own axes but is on the same scale. The middle diagram has no points in common with the left diagram, and only four points in common with the right diagram.

the condition above (right), namely $X < 3/H$ and $Y < 2/H$. Comparing these pictures to Figure 3.3.2 suggests that some hulls with $X \geq 1/H$ and $Y \geq 1/H$ might still fit into $X < 3/H$ and $Y < 2/H$, while avoiding $(1/H, 1/H)$ etc. More generally, one can replace $2/H$ and $3/H$ by other bounds, adjusting the list of exceptional points accordingly.

We have carried out various experiments to evaluate the effect of applying these cutoffs to $S = S_{n,\delta}$. Compared to the simple endgame from Section 3.4, this improved endgame increases H by about $1.1\times$ for $\delta_0 = 1/2$. For $\delta_0 = 1$, there is a smaller increase if n is even, only about $1.015\times$, but a larger increase if n is odd, about $1.5\times$. Sometimes this pushes H above a power-of-2 boundary: for example, 723 divsteps suffice for $\delta_0 = 1$ and $b = 256$, improving upon 724 from Section 3.4.

3.6 Cost of hull computations

We focus here on costs as a function of n rather than as a function of b. We will see in Section 4 that $n \in \Theta(b)$ as $b \to \infty$, at least when $\delta_0 = 1/2$. (In particular, this section's computation terminates for each b when $\delta_0 = 1/2$.)

Moving from the hulls for n to the hulls for $n + 1$ involves considering three layers of data: there can be many δ values; for each δ value, there can be many elements of $S_{n,\delta}$; each element can involve many bits of precision. The following paragraphs consider the data volume at each layer, along with various ways to reduce the data volume. The number of bit operations in standard algorithms (e.g., computing corners via sorting as in [**3**], and using FFT-based multiplication) is essentially linear in the data volume.

Number of δ values. The set D_n defined in Section 3.1 has cardinality $\Theta(n)$, and always contains δ_n.

When the goal is to handle $0 \le g \le f \le 2^b$ for a specific b, one can immediately prune any set $S_{n,\delta}$ that meets the endgame conditions for $H \ge 2^b$, reducing the number of choices of δ to consider.[13]

Sometimes the only reason that a hull cannot be pruned is that it contains some exceptional points, such as the points $(-1/H, -1/H)$ etc. from Section 3.5. We have experimented with tracing each exceptional point separately through the rest of the computation, in effect replacing the hull with a (usually non-convex) union of line segments. We found some cases for $\delta_0 = 1$ for which this improves the algorithm's output: for example, we carried out computations showing that 719 iterations suffice for $b = 255$.

Number of elements of $S_{n,\delta}$. One might guess that the cardinality of $S_{n,\delta}$ grows exponentially with n (for "middle" values of δ), since each element of $S_{n,\delta}$ has two rules producing elements of $T_{n+1,\epsilon}$, with no evident reason for those elements to collide. However, $S_{n+1,\epsilon}$ consists only of the corners of Hull $T_{n+1,\epsilon}$, so it can have much smaller cardinality.

Experimentally, we observe a slow growth of $\max\{\#S_{n,\delta} : \delta \in D_n\}$ as n increases. For example, for $n = 7$ (the rightmost columns in Figures 3.3.1 and 3.3.2), this quantity is 10 for $\delta = 1$ and 12 for $\delta = 1/2$. For $n = 64$, this quantity is 35 for $\delta = 1$ and 72 for $\delta = 1/2$. For $n = 512$, this quantity is 237 for $\delta = 1$ and 300 for $\delta = 1/2$.

Many techniques are known to replace convex polytopes with "simplifications" or "approximations" of those polytopes. Some of those techniques guarantee that the replacement polytope contains the original. Applying those techniques to Hull $S_{n,\delta}$ replaces $S_{n,\delta}$ with a set of controlled size, for example limiting the cardinality to 100 for all n, at the expense of expanding the area of Hull $S_{n,\delta}$ somewhat.[14]

Expanding intermediate hulls could end up producing worse bounds on the number of divsteps required. On the other hand, the bounds in Section 3.4 rely only on the minimum and maximum y for (x, y) in the final hulls, and it is not obvious that this would be affected by a small expansion of area of an intermediate hull. Similar comments apply to Section 3.5.

Precision of elements of $S_{n,\delta}$. For each δ, each point in $S_{n,\delta}$ is easily expressed as $O(n)$ bits: the coordinates x, y are multiples of $1/2^n$ and are in

[13] This also removes the main rationale for checking whether $\max\{|x| : (x, y) \in S_{n,\delta}\} < 1/2^b$. Hulls where x has much smaller limits than y, such as the bottom hulls in Figures 3.3.1 and 3.3.2, arise from earlier hulls where y has much smaller limits than x, and disappear if those earlier hulls are pruned.

[14] For example, under suitable conditions on angles, one can simplify a polygon by replacing adjacent edges (e_0, e_1, e_2) with edges (e'_0, e'_2), where e'_0 and e'_2 are extensions of e_0 and e_2. This replaces two corners with one corner, and expands the polygon by the area enclosed by e'_0, e_1, e'_2. One can greedily choose to remove the edge minimizing this area, and repeat until the number of edges is below 100. Instead of minimizing area, one can minimize, e.g., the sum of squares of distances of the new corner to the two old corners, or the angle covered by e_1.

the interval $[-1, 1]$. One can scale the coordinates by 2^n to work with integers rather than rational numbers.

One way to save time here is to reduce precision: choose $p < n$ and replace the hull corners with multiples of $1/2^p$. Care is required here to avoid accidentally shrinking the hull. In investigating related algorithms from the literature, we noticed that the algorithm from [25], which says it computes the corners of $\text{Hull}(\mathbb{Z}^2 \cap \text{Hull}\, S)$ given S, does not always work correctly: for example, on input $(-4, 2), (5/2, -1), (-2, 5)$, the algorithm's output omits $(-2, 5)$. This is another example of the importance of computer-checked proofs.

4 Stable hulls: analyzing all iteration counts

The point of this section is the following theorem:

Theorem 1. *Let b, f, g be integers with f odd and $0 \le g \le f \le 2^b$. Define $(\delta_n, f_n, g_n) = \text{divstep}^n(1/2, f, g)$. Then $g_n = 0$ for all $n \ge \lceil (9437b + 1)/4096 \rceil$.*

The cutoff $\lceil (9437b + 1)/4096 \rceil$ is, e.g., 590 for $b = 256$, the same cutoff that was achieved in Section 3 for this value of b. The obvious advantage of Theorem 1, compared to Section 3, is that Theorem 1 handles all values of b at once, providing an easy-to-evaluate formula for the cutoff as a function of b. The same advantage was provided by [10], but with a worse bound on n starting with $(1, f, g)$, as noted in Section 1.

Section 4.1 explains how we found the number $9437/4096$. Section 4.2 explains how we found the "stable hulls" at the heart of the proof. Section 4.3 explains the proof. Section 4.4 presents a perspective that unifies the proof strategy used in [10] with the more efficient proof strategy used here; also, as a computational spinoff of this perspective, Section 4.4 constructs 256-bit inputs that require exactly 590 divsteps, matching the upper bound from Theorem 1.

4.1 The scaling factor

We actually have a marginally stronger theorem that replaces the constant $9437/4096$ in front of b with $-1/\log_2 \lambda$ where

$$\lambda = \left((1591853137 + 3\sqrt{273548757304312537})/2^{55}\right)^{1/54} \approx 0.74015.$$

We now explain where these numbers come from.

Along with computing hulls in Section 3.2, one can keep track of which of the linear transformations M_{-1}, M_0, M_1 was applied to each point $(x, y) \in S_{n,\delta}$ to obtain each new point. One can then recursively compute matrices for composed transformations (not necessarily unique) that, when applied to points in $S_{0,1/2}$, produce each point in $S_{n,1/2}$.

Our Sage script `shrinkage.sage` (part of the supplement to this paper) does this for $n = 1$, $n = 2$, and so on through $n = 100$, while printing the largest values of $(\text{radius}\, M)^{1/n}$ for matrices M found in this way. Here $\text{radius}\, M$ means

the spectral radius of M, i.e., the maximum absolute eigenvalue of M. The script also prints the corresponding matrices M.

The largest values printed, around 0.74015, are achieved by two matrices for $n = 54$, one of which has top entries $-708868496/2^{54}$, $-913264476/2^{54}$ and bottom entries $-665640252/2^{54}$, $-882984641/2^{54}$. One can check that the eigenvalues are exactly $(-1591853137 \pm 3\sqrt{273548757304312537})/2^{55}$, i.e., $-\lambda^{54}$ and $-1/(2\lambda)^{54}$; this matrix has $(\text{radius } M)^{1/n} = \lambda$.

For $n/54 \in \{1, 2, 3, \ldots\}$, the $(n/54)$th power of this matrix takes n steps to shrink some input vectors, various eigenvectors of the matrix, to λ^n times their original size.[15] If the input vectors have length 2^b then the output vectors have length $\lambda^n 2^b = 2^{b+n\log_2 \lambda}$, reaching 2^0 only when n reaches $(-1/\log_2 \lambda)b \approx 2.3035b$.

An eigenvector of this matrix cannot be in $\mathbb{Q} \times \mathbb{Q}$ (otherwise $\lambda \in \mathbb{Q}$ so $\lambda^{54} \in \mathbb{Q}$ so 273548757304312537 is a square, contradiction), so it is still conceivable that no *integer* inputs need this number of divsteps. However, we are able to construct an input for, e.g., $b = 256$ that needs 590 divsteps. See Section 4.4.

Our proof strategy ends up with matching upper bounds on n: all input vectors in $\mathbb{R} \times \mathbb{R}$ must, when measured appropriately, shrink to $\leq \lambda$ times the original size on each iteration, so the required number of iterations is at most $(-1/\log_2 \lambda)b \approx 2.3035b$. We stated Theorem 1 with a marginally larger constant $9437/4096 \approx 2.3040$ because this is easier to use, has practically the same performance, and is proven with less computation.

4.2 Finding stable hulls

At a high level, our strategy for proving Theorem 1 is similar to Section 3. We build sets $S_{n,\delta}$ such that if $H > 0$ and $0 \leq g_0 \leq f_0 \leq H$ then $(f_n/H, g_n/H) \in \text{Hull } S_{n,\delta_n}$; these sets shrink with n, forcing $g_n = 0$ once n is big enough.

However, the shrinkage in Section 3 was a numerical observation for each n considered. To obtain a *formula* for this shrinkage for all n, we want a formula for $S_{n,\delta}$ as a function of n and δ. This might seem difficult to reconcile with Figure 3.3.2, where the sets $S_{n,\delta}$ seem to become increasingly complicated as n increases, and with Section 3.6, which reported hundreds of edges in some sets $S_{n,\delta}$ for $n = 512$.

Figure 4.2.1 shows the shapes of the sets $S_{n,1/2}$ for $56 \leq n < 64$. Here the shape of S means the scaled set $X^{-1}S$ where $X = \max\{|x| : (x, y) \in S\}$, assuming $X \neq 0$; i.e., the shape of S is S scaled so that its maximum x-coordinate in absolute value is 1. The shapes in Figure 4.2.1 are all similar-looking tilted ovals, but are not identical.

The crux of our proof of Theorem 1—actually of the marginally stronger theorem mentioned in Section 4.1—is that we replace the sets from Section 3

[15] Spectral radii were used for a similar calculation for $\delta = 1$ in [**10**, Appendix F.1], but the $n = 14$ matrix there was found by a simple brute-force search. A brute-force search would still be feasible for the $n = 54$ matrix found here, but tracking matrices used for hull corners is a faster way to find this matrix.

Fig. 4.2.1. The sets $S_{n,1/2}$ from Section 3.2 when $\delta_0 = 1/2$, for $56 \le n < 64$. Each set is rescaled in the diagram to have maximum absolute x-coordinate 1.

with finite sets $S_{n,1/2} \subseteq \mathbb{R} \times \mathbb{R}$ having an *exactly* stable shape as n increases. Quantitatively, $S_{n,1/2} = \lambda^n S_{1/2}$ for a single finite set $S_{1/2}$, where $\lambda \approx 0.74015$ is the constant from Section 4.1. More generally, we give a formula for S_δ in terms of δ; we define $S_{n,\delta} = \lambda^n S_\delta$; and we show that the divstep linear transformations map the hulls for n to the hulls for $n+1$.

It was not obvious in advance that this proof strategy could work at all, i.e., that any finite sets have stable hulls, beyond the useless set $\{(0,0)\}$. We now explain how our Sage script `fixedpoint.sage` (part of the supplement to this paper) constructs $S_{1/2}$. We defer further comments on the proof of Theorem 1 to Section 4.3.

The script starts with the four points $(-1,-1), (1,-1), (1,1), (-1,1)$. All subsequent computations in the script are symmetric under vector negation. Note that the endgames from Sections 3.4 and 3.5 also have this symmetry. We noticed that different choices of initial points can produce different stable hulls; for this paper, one stable hull suffices.

Starting from this set S of points, the script expands $\operatorname{Hull} S$ as follows: compute $M(S)$ for each of the linear transformations M listed below, and replace S with the corners of $\operatorname{Hull}\bigl(S \cup \bigcup_M M(S)\bigr)$. The script iterates this process in interval arithmetic with 1000 bits of precision. The number of points stabilizes at 102 after 35 iterations, and the point coordinates stabilize after 1507 iterations at this level of precision. The script computes the maximum x-coordinate across the resulting points, and divides all points by that x-coordinate, scaling the points to a shape.

The script then uses standard tools (as in [**17**, Section 2.7.2]) to recognize each of the high-precision point coordinates as an element of the number field $\mathbb{Q}(\lambda)$, and more specifically as a power of λ times an element of $\mathbb{Q}(\sqrt{273548757304312537})$. The script prints out the coordinates, and checks using exact arithmetic in $\mathbb{Q}(\lambda)$ that $M(S) \subseteq \operatorname{Hull} S$ for each M.

We did not have a guarantee in advance that this would work, but it did work. The resulting hull looks similar to the tilted ovals in Figure 4.2.1.

The transformations M used in the script are, up to the appropriate scaling by a power of λ, the linear transformations for all divstep sequences that start and end with $\delta = 1/2$ without passing through $\delta = 7/2$: concretely, M_{-1}/λ; $M_0M_{-1}M_0/\lambda^3$; $M_1M_{-1}M_0/\lambda^3$; $M_0M_0M_{-1}M_0M_0/\lambda^5$; $M_0M_1M_{-1}M_0M_0/\lambda^5$; $M_1M_0M_{-1}M_0M_0/\lambda^5$; and $M_1M_1M_{-1}M_0M_0/\lambda^5$. The $n = 54$ matrix highlighted in Section 4.1 does not pass through $\delta = 7/2$.

We emphasize that the point of this script is to *find* a stable hull. The script does not *prove* stability of the hull: the script considers only a few values of δ

rather than all possible δ. However, our proof of Theorem 1 (see Section 4.3) does consider all possible δ.

4.3 Formal verification of the main theorem

There are several programs called "proof assistants" that claim to check mathematical proofs written from first principles. Writing those proofs often takes longer than writing proofs in the usual way, but provides much higher assurance of correctness: all of the proofs checked by a proof assistant are guaranteed to be correct, unless there is a flaw in the logical core of the proof assistant.

We selected HOL Light, a proof assistant whose logical core is particularly small, stable, and well reviewed; see generally [27], [28], and [1]. Our file `theorem1/README.md` (part of the supplement to this paper) explains how to re-run HOL Light to check our proofs of Theorem 1 and of the marginally stronger theorem mentioned in Section 4.1. These checks take roughly 2^{41} CPU cycles and 2^{46} CPU cycles respectively.

The centerpiece of the stronger proof is the stable hull $S_{1/2}$ constructed in Section 4.2. All sets S_δ are defined as simple linear transformations of $S_{1/2}$, except that $S_{-1/2}$ is separately defined as a related hull with 104 points; and then $S_{n,\delta}$ is defined as $\lambda^n S_\delta$. The overall definition is annoyingly long since $S_{1/2}$ has many points with coordinates such as

$$\left(-\frac{1444331613929472}{16526448172540041922855}\sqrt{273548757304312537}-\frac{6045696}{132913}\right)\lambda^{15}.$$

The faster proof of Theorem 1 uses a replacement $S_{1/2}$ that approximates each point and removes short edges as in Section 3.6, but this replacement still has 80 points such as $(-22123/32768, 7107/65536)$. One would not want to check the calculations by hand.

The main inductive step in the proof, assuming $(x, y) \in S_\delta$, says that $(y, (y-x)/2) \in \operatorname{Hull} \lambda S_{1-\delta}$ if $\delta > 0$, and $(x, (y+x)/2) \in \operatorname{Hull} \lambda S_{1+\delta}$ if $\delta \le 0$, and $(x, y/2) \in \operatorname{Hull} \lambda S_{1+\delta}$. This statement for $(x, y) \in S_\delta$ implies the same statement for $(x, y) \in \operatorname{Hull} S_\delta$, and then scaling by λ^n shows that if $(f_n/H, g_n/H) \in S_{n,\delta_n}$ then $(f_{n+1}/H, g_{n+1}/H) \in S_{n+1,\delta_{n+1}}$. The faster proof uses modified hulls that are not exactly stable, but obtains the same containments after slightly increasing λ to $30902639/41749730 \approx 0.74019$.

For the initial condition, the proof computes H as a constant times 2^b so that $0 \le g \le f \le 2^b$ implies $(f_0/H, g_0/H) \in S_{0,1/2}$. The same constant shows up in the endgame, scaling $S_{1/2}$ to touch the point $(1, 1)$ in the right diagram in Figure 3.5.1 while still comfortably fitting into the green area in the diagram.

Some simplifications and speedups are possible in the proof details. For δ outside the range used in Section 4.1, there is some flexibility in the definition of S_δ, and we inserted a $32/33$ factor into the definition in the hope of improving the endgame; we did not end up using this, and removing the factor would remove some cases from the proof. It might be better to scale $S_{1/2}$ in the

first place so that $(1, 1)$ is on the edge of the hull, rather than scaling to have maximum x-coordinate 1. We could have taken more advantage of symmetry under negation. Many numerical comparisons in the proof are of numbers that are far apart, and we could have reduced the precision of arithmetic used for those numbers. We could have sped up arithmetic as in 2011 Solovyev–Hales [**50**, Section 3].

We actually began by posting examples of even simpler proofs verified by HOL Light for specific small values of b, using the proof strategy from Section 3 without stable hulls. Similar proofs then appeared in 2021 O'Connor–Poelstra [**43**], using Coq rather than HOL Light. The Coq proof-verification time for $b = 256$ (with $n = 590$) was reportedly around 2^{49} CPU cycles (reduced to about 2^{48} in [**34**, April 2021 version]), or 2^{54} cycles in "paranoid mode".

Then 2023 Hvass–Aranha–Spitters [**34**, June 2023 version] announced Coq verification of almost [**10**, Theorem 11.2], using about 2^{47} CPU cycles. The reason we say "almost" is that [**10**, Theorem 11.2] is actually two results, one assuming $b \geq 21$ and $\lfloor (49b + 57)/17 \rfloor \leq n$ (allowing, e.g., $n = 741$ for $b = 256$), the other assuming $b < 21$ and $\lfloor (49b + 80)/17 \rfloor \leq n$; the computer-checked proof in [**34**] covered only the first result.

After [**43**] and [**34**], a software author handling arbitrary b could confidently use $\delta_0 = 1/2$ for $b \leq 256$, and switch over to $\delta_0 = 1$ for larger b using more iterations, while relying on computer-checked proofs of the iteration counts in all cases. But these results did not provide an all-b theorem (whether or not computer-checked) stating iteration counts for $\delta_0 = 1/2$.

All of those proofs are superseded by our proof of Theorem 1. Our proof (1) handles $\delta_0 = 1/2$ for all b at once; (2) reaches essentially state-of-the-art iteration counts, such as $n = 590$ for $b = 256$; (3) is verified by the simple logical core of HOL Light; and (4) takes only about 2^{41} CPU cycles to verify.

4.4 Measuring size, revisited

One way to describe the 2-norm of a vector v is as the smallest real number $r \geq 0$ such that $v \in rB$, where B is the unit ball. More generally, for any closed bounded convex set C containing a neighborhood of 0, one can define the C-norm of v as the smallest $r \geq 0$ such that $v \in rC$. This is a standard generalization, also called a "gauge" or (when C is symmetric under negation) a "Minkowski functional"; see, e.g., [**13**, page 127].

The point of our proof is then that the divstep linear transformations reduce norm r to norm at most λr, where norm is measured relative to the stable hulls that we construct. Our faster proof uses approximations to those hulls and shows almost as much reduction.

For comparison, [**10**, Appendix F] used conventional 2-norms, which are much farther from being stable, and compensated by considering sequences of many divsteps. That approach is capable of coming arbitrarily close to the correct scaling (the gap between norms is bounded, while many scaling steps have an exponential effect), but incurs extra expense in the proof. Quantitatively,

[10] obtained ≈2.882b, about 2% above the presumed optimum of ≈2.828b for $\delta = 1$; [10, proof of Theorem F.22] reported 2^{49} cycles (in Sage, not with a computer-checked proof), and further computation was required in [10, Theorem G.4] to handle small values of b.

Using a norm adapted to the algorithm has a side benefit beyond improving the proof: it lets us efficiently find input examples that require many algorithm steps. We start with an output such as $(1/2, 1, 0)$, and work backwards to possibilities for the previous step, possibilities for the step before that, etc. To prune this search, we keep only the smallest possibilities at each step, with smallness measured by our stable norm. We find, for example, that the 256-bit inputs

$f =$ `0xeec9f80577a885d22f8d37c1946187e26805ea27b26c5ae10aa38a02e2ea3157`,

$g =$ `0xeb40350da50b11d23183ae8e88ffced0ad11263b6d62cde5e5dc1e934ef8229c`

require 590 divsteps. Such examples are useful as algorithm tests, and show that 590 cannot be improved for this algorithm.

Other modular-inversion algorithms use fewer iterations: recall, for example, that $2b$ iterations suffice for [51]. But 590 is not much larger than 512, and divsteps allow particularly streamlined software, as we explain in Section 5.

5 Laying out the data and selecting instructions

As one can see from Table 1.2.1, our inverters modulo $p = 2^{255} - 19$ take under 7 cycles per divstep on Intel Skylake. This section explains how we achieve this speed.

Section 5.1 reviews the basic structure of a modular-inversion algorithm. Section 5.2 explains how we speed up the critical inner loop. Beyond the inner loop, we take two different approaches to save time: Section 5.3 explains "Strategy 1" using vectorized multiplication instructions, and Section 5.4 explains "Strategy 2" using serial multiplication instructions. Strategy 1 is faster, but Strategy 2 is easier to port to other architectures and other modulus sizes.

We label the full inversion software as "Inverter 1X" for AMD/Intel, "Inverter 2X" for AMD/Intel, "Inverter 1A" for ARM, and "Inverter 2A" for ARM. We posted the software for Inverter 1X [11] in 2021-01 (along with an adaptation handling arbitrary odd moduli between 2^{255} and 2^{256}), for Inverter 2X [31] and Inverter 2A [30] in 2023-07 (along with adaptations to the prime moduli for NIST curves P-256, P-384, and P-521), and for Inverter 1A [33] in 2026-01. See the full version of this paper for further timeline information. For concreteness, this section focuses on $2^{255} - 19$ on Skylake.

5.1 Modular inversion

It is easy to recursively define coefficients u_n, v_n, q_n, r_n such that $f_n = u_n f_0 + v_n g_0$ and $g_n = q_n f_0 + r_n g_0$, for example by stating that linear operations on f and g are applied to the vectors (f, u, v) and (g, q, r), as in, e.g., [37, Algorithm

4.5.2–X]. These are also the coefficients of the linear transformations tracked in Section 4.1.

If $\gcd\{f_0, g_0\} = 1$ and $g_n = 0$ then these "extended gcd" coefficients immediately produce modular reciprocals: one has $f_n = \pm 1$, so $\pm v_n$ (specifically, $f_n v_n$) is the desired reciprocal of g_0 modulo f_0. For this application, u_n and q_n are irrelevant: one recursively defines v_n and r_n to satisfy the invariants $f_n \equiv v_n g_0 \pmod{f_0}$ and $g_n \equiv r_n g_0 \pmod{f_0}$.

A standard speedup, starting with 1938 Lehmer [38], is to jump from f_n, g_n to (say) f_{n+10}, g_{n+10}, by applying a linear transformation computed from low-precision approximations to f_n, g_n. This is particularly straightforward for divsteps since, as [10] emphasizes, 10 low bits of f_n and 10 low bits of g_n always determine the next 10 divsteps. The same linear transformation jumps from v_n, r_n to v_{n+10}, r_{n+10}. In this context, we relabel f_n, g_n, v_n, r_n as F_n, G_n, V_n, R_n, while writing f, g, u, v, q, r for the low-precision quantities used for the inner loop.

5.2 Six divstep words in two CPU registers

We display here, in `qhasm` [5] format, a divstep inner loop with Skylake latency just 4 cycles. We saved many instructions by packing (f, u, v) into a single 64-bit CPU register $\texttt{fuv} = f - 2^{41}u - 2^{62}v$ and packing (g, q, r) into $\texttt{gqr} = g - 2^{41}q - 2^{62}r$. We use $-$ rather than $+$ to fit better in two's-complement arithmetic. We unroll this loop 20 times to carry out 20 divsteps on `fuv` and `gqr` as if those were f and g; these are congruent to f and g modulo 2^{20}.

```
z = -1
oldg = gqr
h = gqr + fuv
            =? gqr & 1
z = m   if !=
h = gqr if  =
mnew = m - 1
gqr -= fuv
(int64) gqr >>= 1
(int64) h >>= 1
m = -m
                signed<? z - 0
fuv = oldg if  signed<
gqr = h    if !signed<
m = mnew   if !signed<
```

Unlike many SWAR applications, this packing has bit boundaries in continual motion: u, v, q, r start as $1, 0, 0, 1$ and are multiplies of 2^{-k} after k of the 20 steps, always with $-1 < u, v, q, r \leq 1$. After 20 steps we have $\texttt{fuv} = f - 2^{21}u' - 2^{42}v'$ and $\texttt{gqr} = g - 2^{21}q' - 2^{42}r'$ for integers $u' = 2^{20}u$ etc. Unpacking u', v', q', r' costs some instructions, but that cost is amortized across the 20 divsteps.[16]

In the code, `h = gqr + fuv` and `mnew = m - 1` (here $m = \delta - 1/2$) are 3-register add instructions (`LEA`) to avoid having to copy registers. The shifts are all signed. A third of the instructions are conditional moves (`CMOV`) to avoid branching, so the code is somewhat slower on Intel Haswell CPUs. (The

[16] As a followup, Peter Dettman suggested a variant that allows 30 divsteps—but at the cost of extra inner-loop instructions, making this variant slightly slower in our experiments.

semi-binary code from 2020 Pornin [47] also relies heavily on CMOV. That code packs low and high bits of f into one register, and packs low and high bits of g into another register.)

We handle 60 divsteps by extracting the bottom 60 bits of f and g and then applying three 20-divstep jumps. Each jump is straightforwardedly handled with the IMUL instruction and some 3-register adds (LEA).

5.3 Vectorized 60-divstep jumps (Strategy 1)

After carrying out 60 divsteps as above, we apply the resulting linear transformation to jump from F_n, G_n to F_{n+60}, G_{n+60} and from V_n, R_n to V_{n+60}, R_{n+60}. The coefficients in the linear transformation are 61-bit signed integers (scaled by -2^{60}).

Our fastest jump software on Intel chips without AVX-512 uses the vectorized $32 \times 32 \to 64$ multipliers. To handle signed coefficients, we use *signed* limbs and the PMULDQ instruction, which multiplies the lower half of 64-bit lanes as signed integers into a signed 64-bit integer. We use radix 2^{30} with lazy carries, representing each of the 61-bit signed integers as two limbs and accumulating many products within 64 bits before carrying.

A carry involves shifting a signed integer 30 bits to the right. The vectorized 64-bit shifts in AVX2 are only unsigned (PSRLQ), so we add -2^{63} (exchanging positive x with $x - 2^{63}$), shift unsigned 30 bits, then subtract 2^{33}. Usually the subtraction can be merged into a subsequent addition.

Multi-limb Montgomery reduction. As usual, we reduce modulo $p = 2^{255} - 19$ to a limited range ("squeezing"), but we defer full reduction to $\{0, \ldots, p-1\}$ ("freezing") until the end of the computation.

We transpose the limbs of F, V, G, R, putting the ith limbs into a 4-lane vector $\mathtt{FVGR}_i$. After we finish computing $\mathtt{NewFVGR}_0$ and $\mathtt{NewFVGR}_1$, we add a suitable multiple of the prime p, setting the other multiplicand L to be the limb multiplied by the precomputed reciprocal of p modulo 2^{30}. We then compute $\mathtt{NewFVGR}_i$ for $i \geq 2$ as

$$\mathtt{uurr}_1\mathtt{FVGR}_{i-1} + \mathtt{vvqq}_1\mathtt{GRFV}_{i-1} + \mathtt{uurr}_0\mathtt{FVGR}_i + \mathtt{vvqq}_0\mathtt{GRFV}_i + \mathtt{L}_1\mathtt{p}_{i-1} + \mathtt{L}_0\mathtt{p}_i + \mathtt{carry}_{i-1}.$$

Then we carry the over-wide result.

The modulus $p = 2^{255} - 19$ has signed limbs $2^{15}, 0, 0, 0, 0, 0, 0, 0, -19$ in radix 2^{30}. We save time in reductions by skipping multiplications by zero limbs. On the other hand, reduction is only a small part of divstep-based inversion, unlike Fermat inversion. We have written (but not verified) a Strategy 1 inverter that handles any 256-bit odd modulus with only 5% more cycles.

Interleaving vector and non-vector instructions. Only a small portion of the jump computation matters for the next divsteps, and the vector registers are disjoint from the registers used in the divsteps loop. We interleave vector

computations with other computations, partially masking the jump latency. Such interleaving can also be found in, e.g., 2012 Bernstein–Schwabe [9, Section 3], but the Skylake pipeline is much more complicated than the Cortex-A8 pipeline targeted in [9]; we use a genetic algorithm for instruction scheduling.

5.4 Non-vectorized 59-divstep jumps (Strategy 2)

The AMD/Intel $64 \times 64 \to 128$ integer multiplier might sound as efficient as a 4-way vectorized $32 \times 32 \to 64$ multiplier, since $64^2 = 4 \cdot 32^2$. However, the surrounding instructions for additions etc. are typically less efficient, especially for signed arithmetic, and allow very little interleaving with other operations.

So it is unsurprising that our non-vectorized Inverter 2X is not quite as fast on Skylake as our Inverter 1X. However, as noted above, avoiding vectors simplifies adaptations of the software to other contexts.

For $p = 2^{255} - 19$, we represent each of the quantities F, G, V, R as an array of four 64-bit limbs in "saturated" form, i.e., radix 2^{64}, since this is what fits most straightforwardly with the integer instructions.[17] Here F, G are two's-complement, and V, R are squeezed modulo p into the range $\{0, 1, \ldots, 2p-1\}$. To apply a linear transformation with signed coefficients, we break each coefficient into a sign-magnitude combination and then perform appropriate combinations of optional bitwise complements and unsigned multiplications and additions.

There are some smaller software differences. Perhaps most noticeable is that, while we use 10 iterations of a main loop for both Inverter 1X and Inverter 2X, this main loop has 60 divsteps in Inverter 1X and 59 divsteps in Inverter 2X. Recall that 590 iterations handle all odd moduli f_0 below 2^{256}, assuming frozen g_0, i.e., $0 \le g_0 < f_0$.

6 Verifying the software

Software sometimes has bugs—accidental overflows of CPU words, for example. Verifying software that claims to compute modular inverses via divsteps thus requires not just verifying how many divsteps are required, as in Sections 3 and 4, but also verifying that the software correctly computes those divsteps.

Section 6.1 briefly reviews software verification. We tried two verification tools, in particular using CryptoLine [53] to verify Inverter 1X/1A (see Section 6.2) and using HOL Light to verify Inverter 2X/2A (see Section 6.3). Both approaches were successful. Beware that each verification is relative to a specification of CPU behavior, so if a physical CPU deviates from the specification (see, e.g., [40] regarding faults triggered by low voltage, or [21] and [46] as examples of outright CPU bugs) then the CPU might still miscompute modular inverses.

Some divstep-based inverters written in C were verified in [34] and (after our work) [44]. The C code is more portable than our code, but it is slower, and the

[17] For the P-256 prime, we still use 4×64 bits for V and R but end up using 5×64 bits for F and G to allow unambiguous two's-complement representation.

verification relies on a specification of C behavior rather than just a specification of CPU behavior. We instead use tools that can handle verifying machine code.

6.1 Preconditions and postconditions

The literature starting with 1967 Floyd [23] and 1969 Hoare [32] often expresses program correctness (modulo notation) as a "Hoare triple" $\{P\}\ C\ \{Q\}$, meaning that if a program C is executed in any state satisfying the *precondition* P, then it will eventually terminate in a state satisfying the *postcondition* Q. (A variant, "partial correctness", says that *if* the program terminates then Q is satisfied.) In Hoare's formulation, there is a corresponding set of program-verification rules associated with structured-programming constructs such as sequencing and while-loops. Despite its association with structured programs, Hoare logic can be adapted to the setting of machine code (see 2007 Myreen–Fox–Gordon [41]). The tools that we use here adopt such an approach to specification and verification.

6.2 CryptoLine verification of Inverter 1A and Inverter 1X

CryptoLine is an automatic verification tool designed for straight-line cryptographic assembly programs. Given user-specified assumptions and properties, CryptoLine utilizes Satisfiability Modulo Theories (SMT) solvers and computer-algebra systems to prove properties automatically. If external solvers fail to prove specified properties, additional properties can be added, proved, and then used as lemmas to help external solvers. The tool has been used to verify field arithmetic [26] and number-theoretic-transform assembly programs [35].

To verify an assembly program using CryptoLine, the user first constructs a straight-line program model using the CryptoLine language. The model can be translated from traces extracted from GDB or written by the user manually. After the program model is built, pre- and post-conditions need to be specified by the user. The model can then be sent to the CryptoLine tool for verification. Sometimes the tool accepts the Hoare triple without further input, so the verification is done. Otherwise the user has to annotate the program model with more properties to help verification, such as by adding assertions or cutting the program into separately verified components. The annotation process continues until postconditions are verified or a bug is found by the tool.

We use CryptoLine to formally verify our Inverter 1A and Inverter 1X; the following description focuses on Inverter 1X. Recall from Section 5 that the software packs (f, u, v) and (g, q, r) into 64-bit registers `fuv` and `gqr` respectively. Then 20 divsteps are performed on `fuv` and `gqr`. For the ith divstep, after scaling notation by a negated power of 2 to use integer variables, we have $\texttt{fuv} = f + 2^{41-i}u + 2^{62-i}v$, $\texttt{gqr} = g + 2^{41-i}q + 2^{62-i}r$ with $-2^i \le u, v, q, r < 2^i$ and odd f. We verify the assembly code satisfies three postconditions (the notation x' here means the resulting value of variable x): (1) If $\texttt{m} > 0$ and g is odd then $f' = g$, f' is odd, $2g' = g - f$, $\texttt{m}' = 0 - \texttt{m}$, $u' = 2q$, $v' = 2r$, $q' = q - u$, $r' = r - v$. (2) If $\texttt{m} < 0$ and g is odd then $f' = f$, $2g' = f + g$, $\texttt{m}' = 1 + \texttt{m}$, $u' = 2u$, $v' = 2v$,

$q' = q + u$, $r' = r + v$. (3) If g is even then $f' = f$, $2g' = g$, $\mathtt{m}' = 1 + \mathtt{m}$, $u' = 2u$, $v' = 2v$, $q' = q$, $r' = r$.

In all cases, CryptoLine also verifies $-2^{i+1} \leq u', v', q', r' < 2^{i+1}$, $\mathtt{fuv}' = f' + 2^{40-i}u' + 2^{61-i}v'$, and $\mathtt{gqr}' = g' + 2^{40-i}q' + 2^{61-i}r'$. The postconditions allow divstep to be repeated 20 times on 64-bit registers. Assuming $-2^{60} \leq \mathtt{m} \leq 2^{60}$, odd f, and $(u, v, q, r) = (-1, 0, 0, -1)$, the conditions $-2^{60} \leq u' \pm v', q' \pm r' \leq 2^{60}$ are also verified after 60 divsteps.

After 60 divsteps, we obtain u, v, q, r such that $uF_n + vG_n \equiv qF_n + rG_n \equiv 0 \bmod 2^{60}$ where $|F_n|, |G_n| < 2^{255}$. Inverter 1X uses vector instructions on limbs in radix 2^{30} to compute $F_{n+60}, G_{n+60}, V_{n+60}, R_{n+60}$ with multi-limb Montgomery reduction (Section 5.3). Assume previously verified $-2^{60} \leq u' \pm v', q' \pm r' \leq 2^{60}$ and $-2^{29} < V_n, R_n < 2^{255} + 2^{29}$. The assembly code computes $F_{n+60}, G_{n+60}, V_{n+60}, R_{n+60}$ with the following postconditions: $2^{60}F_{n+60} = uF_n + vG_n$, $2^{60}G_{n+60} = qF_n + rG_n$, $2^{60}V_{n+60} \equiv uV_n + vR_n \bmod p$, $2^{60}R_{n+60} \equiv qV_n + rR_n \bmod p$ with $-2^{29} < V_{n+60}, R_{n+60} < 2^{255} + 2^{29}$. The postconditions again allow us to repeat the linear transformation after every 60 divsteps. The conclusion of verification is that the code correctly computes 600 divsteps; it is then up to the user to see that this suffices by Theorem 1.

6.3 HOL Light verification of Inverter 2A and Inverter 2X

We have a theorem verified by HOL Light stating that the 32-byte output array produced by Inverter 2A contains the inverse modulo $p = 2^{255} - 19$ of the 32-byte input array, or 0 if the input is divisible by p. We have a similar theorem for Inverter 2X. We also have various theorems for other moduli, but for concreteness we focus here on $2^{255} - 19$.

Working in a unified theorem-proving environment, with every proof checked in terms of logical primitives, convincingly addresses the risk of proof errors, but one still has to ask whether real CPUs match the machine model used to state the theorem. We reuse general definitions of machine-code semantics from the s2n-bignum library [**29**] introduced by Harrison in 2021; Section 6.3.1 reviews the general structure of those definitions. Section 6.3.2 presents highlights of our proof.

6.3.1 Verification infrastructure in s2n-bignum. The s2n-bignum equivalent of a Hoare triple is "`ensures` M P Q F", meaning that if the machine defined by M is in any state $\sigma \in P$ then, after some finite sequence of instructions, the machine will reach some state $\sigma' \in Q$ with $(\sigma, \sigma') \in F$. The components are as follows:

- $M \subseteq \Sigma \times \Sigma$ is the machine model, such as `arm` or `x86`. Here Σ is the set of all conceivable machine states, and $(\sigma, \sigma') \in M$ means that σ' is a possible "next" state that could arise after executing a single instruction in state σ. The successor state is not always uniquely determined by σ: e.g., if the Intel/AMD documentation for an instruction leaves a flag undefined then `x86` allows multiple choices of σ' with different flag settings.

- $P \subseteq \Sigma$ is a *precondition*: for example, an assumption that the ARM program counter points to particular bytes of code, that ARM register `X30` contains a return address, and that another register points to an input array with the little-endian representation of a particular integer n.
- $Q \subseteq \Sigma$ is a *postcondition*: for example, a conclusion about the program counter now pointing to the return address and about another register pointing to an output array with particular contents related to n.
- $F \subseteq \Sigma \times \Sigma$ is a *frame condition*: for example, a statement that the initial state matches the final state outside the output array, a specified region of the stack, and the ARM caller-save registers.

Typical s2n-bignum proofs of an `ensures` statement use intermediate `ensures` statements to decompose the proof, e.g., setting up loop invariants. Significant subcomponents, ranging from small basic blocks to relatively large functions (possibly with loops that are in effect unrolled), have `ensures` verified directly by symbolic simulation in the style of SAW from [20].

6.3.2 Verifying divsteps. Our core theorem regarding Inverter 2A has the following conclusion:

```
ensures arm
  (\s. aligned_bytes_loaded s (word pc) bignum_inv_p25519_mc /\
       read PC s = word pc /\
       read SP s = stackpointer /\
       read X30 s = returnaddress /\
       C_ARGUMENTS [z; x] s /\
       bignum_from_memory(x,4) s = n)
  (\s. read PC s = returnaddress /\
       bignum_from_memory(z,4) s =
       (if p_25519 divides n then 0 else inverse_mod p_25519 n))
  (MAYCHANGE_REGS_AND_FLAGS_PERMITTED_BY_ABI ,,
   MAYCHANGE [memory :> bytes(z,8 * 4);
              memory :> bytes(word_sub stackpointer (word 160),160)])
```

The machine code for our `bignum_inv_p25519` function is defined before the theorem as an explicit array of numbers, `bignum_inv_p25519_mc`. The machine model in the theorem is `arm`. The precondition (using "`/\`" for "and") states that `bignum_inv_p25519_mc` is loaded into memory at address `pc`, that various input arguments are in the expected registers, and that a 4-word bignum starting at address `x` has some generic value `n`. The postcondition states that the final program counter contains the appropriate return address while the 4-word result at address `z` is the inverse we seek, explicitly proved to be zero in cases where there is no inverse. The frame condition states that, besides ABI-permitted registers and flags, the only state components that could have changed between initial and final states are the output buffer itself and an additional 160 bytes of stack that is used for temporary storage. The theorem also has a hypothesis (not shown above) requiring an aligned stack and non-overlaps of various memory regions.

The proof uses Hoare's while-rule to set up a loop invariant I_i as a formal rendering of the properties from Section 5 such as $(\delta_i, f_{59i}, g_{59i}) = \text{divstep}^{59i}(1/2, f_0, g_0)$ and $f_{59i} \equiv v_{59i} g_0 \pmod{p}$, with all parameters associated with specific memory addresses and registers. To show that one iteration of the main loop starting in a state satisfying I_i will reach a state satisfying I_{i+1}, the proof uses symbolic simulation, broken into three main parts, each with a slightly different style: (1) word-level 59-fold divstep; (2) multiplying the resulting word matrix into big-integer vectors (F, G) and (V, R); (3) squeezing V and R below $2p$.

Recall that each of F, G, V, R is represented in saturated form. Saturated representations involve carry chains, which are a common pattern in s2n-bignum, with proofs effectively automated by some s2n-bignum proof tactics that can accumulate sum-carry combinations and where appropriate show that carries are zero via bounds reasoning. To handle signed coefficients of linear transformations, our proof performs case splits over all sign combinations and applies the same tactic in all four cases.

To squeeze V and R modulo p into the range $\{0, 1, \ldots, 2p-1\}$, the software uses the fact that if $|n| \leq 2^{316}$ then $0 \leq n - qp < 2p$ where $q = \lfloor n/2^{255} \rfloor - [n < 0]$. This has a 1-line HOL Light proof, on top of 8 lines to state the lemma:

```
let p25519signedredlemma = prove
 (`forall n.
        abs(n) <= &2 pow 316
        ==> let p_25519 = &2 pow 255 - &19 in
            let q = n div &2 pow 255 in
            let q' = if n < &0 then q - &1 else q in
            abs(q') < &2 pow 62 /\
            &0 <= n - q' * p_25519 /\
            n - q' * p_25519 < &2 * p_25519`,
  REWRITE_TAC[LET_DEF; LET_END_DEF] THEN INT_ARITH_TAC);;
```

The correctness proof `arm64_bignum_inv_p25519.ml` for Inverter 2A has 3324 lines in total, including 1036 automatically generated lines that define `bignum_inv_p25519_mc`, the machine code being verified. The original assembly code, including macros and comments, is over 1200 lines. The formal-verification effort for this function is thus about 2 proof lines per line of code—on top of the preexisting s2n-bignum infrastructure and our proof of Theorem 1. We also reuse the same background, along with 1854 lines from the Inverter 2A proof, in the proof of an analogous theorem for Inverter 2X.

References

[1] Oskar Abrahamsson, Magnus O. Myreen, Ramana Kumar, Thomas Sewell, *Candle: A verified implementation of HOL Light*, in ITP 2022 (2022), 3:1–3:17.

[2] Alejandro Cabrera Aldaya, Cesar Pereida García, Luis Manuel Alvarez Tapia, Billy Bob Brumley, *Cache-timing attacks on RSA key generation*, IACR TCHES **2019.4** (2019), 213–242.

[3] Alex M. Andrew, *Another efficient algorithm for convex hulls in two dimensions*, Information Processing Letters **9** (1979), 216–219.
[4] Daniel J. Bernstein, *Curve25519: new Diffie-Hellman speed records*, in PKC 2006 (2006), 207–228.
[5] Daniel J. Bernstein, *Writing high-speed software* (2006).
[6] Daniel J. Bernstein, *Robust statistics for rejection-sampling timings* (2025).
[7] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, Bo-Yin Yang, *High-speed high-security signatures*, Journal of Cryptographic Engineering **2** (2012), 77–89.
[8] Daniel J. Bernstein, Tanja Lange, *Safe curves for elliptic-curve cryptography*, in Dawson proceedings (2025), 124–191.
[9] Daniel J. Bernstein, Peter Schwabe, *NEON crypto*, in CHES 2012 (2012), 320–339.
[10] Daniel J. Bernstein, Bo-Yin Yang, *Fast constant-time gcd computation and modular inversion*, IACR TCHES **2019** (2019), 340–398.
[11] Daniel J. Bernstein, Bo-Yin Yang, *Fast constant-time gcd computation and modular inversion: Software* (2020).
[12] Marco Bodrato, Marc Glisse, Torbjörn Granlund, Niels Möller, *GNU MP: The GNU Multiple Precision Arithmetic Library, version 6.3.0* (2023).
[13] Jonathan M. Borwein, Jon D. Vanderwerff, *Convex functions: constructions, characterizations and counterexamples*, Cambridge University Press, 2010.
[14] Joppe W. Bos, *Constant time modular inversion*, Journal of Cryptographic Engineering **4** (2014), 275–281.
[15] Nicolai Brown, *Things that use Curve25519* (2025), accessed 2025-10-02.
[16] Nicolai Brown, *Things that use Ed25519* (2025), accessed 2025-10-02.
[17] Henri Cohen, *A course in computational algebraic number theory*, Graduate Texts in Mathematics, 138, Springer, Berlin, 1993.
[18] Richard A. DeMillo, Richard J. Lipton, Frederick G. Sayward, *Hints on test data selection: help for the practicing programmer*, Computer **11** (1978), 34–41.
[19] Peter Dettman, Pieter Wuille, *This file implements modular inversion* (2021).
[20] Robert Dockins, Adam Foltzer, Joe Hendrix, Brian Huffman, Dylan McNamee, Aaron Tomb, *Constructing semantic models of programs with the Software Analysis Workbench*, in VSTTE 2016 (2016), 56–72.
[21] Alan Edelman, *The mathematics of the Pentium division bug*, SIAM Review **39** (1997), 54–67.
[22] Euclid, *Elements*, about 300 B.C.
[23] Robert W. Floyd, *Assigning meanings to programs*, in PSAM 19 (1967), 19–32.
[24] Agner Fog, *The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers* (2025).
[25] Clinton Freeman, David L. Millman, Jack Snoeyink, *Gift wrapping the integer hull in the plane*, in EuroCG 2014 (2014).
[26] Yu-Fu Fu, Jiaxiang Liu, Xiaomu Shi, Ming-Hsien Tsai, Bow-Yaw Wang, Bo-Yin Yang, *Signed cryptographic program verification with typed* CryptoLine, in CCS 2019 (2019), 1591–1606.
[27] John Harrison, *HOL Light: A tutorial introduction*, in FMCAD 1996 (1996), 265–269.
[28] John Harrison, *Towards self-verification of HOL Light*, in IJCAR 2006 (2006), 177–191.
[29] John Harrison, *s2n-bignum* (2021).
[30] John Harrison, *Invert modulo m* (2021), software for 64-bit ARM.

[31] John Harrison, *Invert modulo m* (2021), software for 64-bit AMD/Intel.
[32] C. A. R. Hoare, *An axiomatic basis for computer programming* , CACM **12** (1967), 576–580.
[33] Cesare Huang, *Untitled* (2026).
[34] Benjamin Salling Hvass, Diego F. Aranha, Bas Spitters , *High-assurance field inversion for curve-based cryptography*, in CSF 2023 (2023), 552–567.
[35] Vincent Hwang, Jiaxiang Liu, Gregor Seiler, Xiaomu Shi, Ming-Hsien Tsai, Bow-Yaw Wang, Bo-Yin Yang, *Verified NTT multiplications for NISTPQC KEM lattice finalists: Kyber, SABER, and NTRU*, IACR Transactions on Cryptographic Hardware and Embedded Systems **2022** (2022), 718–750.
[36] Burton S. Kaliski, *The Montgomery inverse and its applications*, IEEE Transactions on Computers **44** (1995), 1064–1065.
[37] Donald E. Knuth, *The art of computer programming, volume 2: seminumerical algorithms*, 3rd edition, Addison-Wesley, 1997.
[38] Derrick H. Lehmer, *Euclid's algorithm for large numbers*, American Mathematical Monthly **45** (1938), 227–233.
[39] Victor S. Miller, *Use of elliptic curves in cryptography*, in Crypto 1985 (1986), 417–426.
[40] Kit Murdock, David F. Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, Frank Piessens, *Plundervolt: software-based fault injection attacks against Intel SGX*, in IEEE S&P 2020 (2020), 1466–1482.
[41] Magnus O. Myreen, Anthony C. J. Fox, Michael J. C. Gordon, *Hoare logic for ARM machine code*, in FSEN 2007 (2007), 272–286.
[42] Kaushik Nath, Palash Sarkar , *Efficient arithmetic in (pseudo-)Mersenne prime order fields*, Advances in Mathematics of Communications **16** (2022), 303–348.
[43] Russell O'Connor, Andrew Poelstra, *A formal proof of safegcd bounds* (2021).
[44] Russell O'Connor, Andrew Poelstra, *Formal verification of the Safegcd implementation* (2025).
[45] 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* (2026).
[46] Tavis Ormandy, *Zenbleed* (2023).
[47] Thomas Pornin, *Optimized binary GCD for modular inversion* (2020).
[48] Ronald L. Rivest, Adi Shamir, Leonard M. Adleman , *A method for obtaining digital signatures and public-key cryptosystems*, CACM **21** (1978), 120–126.
[49] Martin Roetteler, Michael Naehrig, Krysta M. Svore, Kristin E. Lauter , *Quantum resource estimates for computing elliptic curve discrete logarithms*, in ASIACRYPT 2017 (2017), 241–270.
[50] Alexey Solovyev, Thomas C. Hales, *Efficient formal verification of bounds of linear programs*, in MKM 2011 (2011), 123–132.
[51] Josef Stein, *Computational problems associated with Racah algebra*, Journal of Computational Physics **1** (1967), 397–405.
[52] Falko Strenzke, *Timing attacks against the syndrome inversion in code-based cryptosystems*, in PQCrypto 2013 (2013), 217–230.
[53] Ming-Hsien Tsai, Bow-Yaw Wang, Bo-Yin Yang, *Certified verification of algebraic properties on low-level mathematical constructs in cryptographic programs*, in CCS 2017 (2017), 1973–1987.
[54] Jan Wichelmann, Ahmad Moghimi, Thomas Eisenbarth, Berk Sunar, *MicroWalk: a framework for finding side channels in binaries*, in ACSAC 2018 (2018), 161–173.
[55] Davey Winder, *Warning: Google researcher drops Windows 10 zero-day security bomb* (2019).