Accelerating the Delfs–Galbraith algorithm with fast subfield root detection

Maria Corte-Real Santos[1*], Craig Costello[2], and Jia Shi[3†]

[1] University College London, London, UK
maria.santos.20@ucl.ac.uk
[2] Microsoft Research, Redmond, USA
craigco@microsoft.com
[3] University of Waterloo, Waterloo, Canada
j96shi@uwaterloo.ca

Abstract. We give a new algorithm for finding an isogeny from a given supersingular elliptic curve $E/\mathbb{F}_{p^2}$ to a subfield elliptic curve $E'/\mathbb{F}_p$, which is the bottleneck step of the Delfs–Galbraith algorithm for the general supersingular isogeny problem. Our core ingredient is a novel method of rapidly determining whether a polynomial $f \in L[X]$ has any roots in a subfield $K \subset L$, while avoiding expensive root-finding algorithms. In the special case when $f = \Phi_{\ell,p}(X, j) \in \mathbb{F}_{p^2}[X]$, i.e., when f is the ℓ-th modular polynomial evaluated at a supersingular j-invariant, this provides a means of efficiently determining whether there is an ℓ-isogeny connecting the corresponding elliptic curve to a subfield curve. Together with the traditional Delfs–Galbraith walk, inspecting many ℓ-isogenous neighbours in this way allows us to search through a larger proportion of the supersingular set per unit of time. Though the asymptotic $\tilde{O}(p^{1/2})$ complexity of our improved algorithm remains unchanged from that of the original Delfs–Galbraith algorithm, our theoretical analysis and practical implementation both show a significant reduction in the runtime of the subfield search. This sheds new light on the concrete hardness of the general supersingular isogeny problem (i.e. the foundational problem underlying isogeny-based cryptography), and has immediate implications on the bit-security of schemes like B-SIDH and SQISign for which Delfs–Galbraith is the best known classical attack.

Keywords: Isogeny-based cryptography, supersingular isogeny problem, Delfs–Galbraith algorithm.

1 Introduction

In its most general form, the *supersingular isogeny problem* asks to find an isogeny

$$\phi\colon E_1 \to E_2$$

[*]Supported by EPSRC grant EP/S022503/1.

[†]Part of this work was done while Jia was an intern at Microsoft Research.

between two given supersingular curves, $E_1/\bar{\mathbb{F}}_p$ and $E_2/\bar{\mathbb{F}}_p$. We emphasize that this is the general problem, where we do not assume knowledge of the degree of the isogeny, or any torsion point information. The best known classical attack against the supersingular isogeny problem is the Delfs–Galbraith algorithm [13], which, for two curves E_1 and E_2 defined over $\mathbb{F}_{p^2}$, has two steps. The first step computes random walks in the ℓ-isogeny graph (for some choice of ℓ) to find isogenies $\phi_1 \colon E_1 \to E_1'$ and $\phi_2 \colon E_2 \to E_2'$, such that $E_1'/\mathbb{F}_p$ and $E_2'/\mathbb{F}_p$ are *subfield curves*. There are around $\lfloor p/12 \rfloor$ supersingular curves up to isomorphism and $O(p^{1/2})$ of them are subfield curves, therefore this step runs in $\tilde{O}(p^{1/2})$ bit operations. The second step searches for a *subfield* isogeny $\phi' \colon E_1' \to E_2'$ that connects ϕ_1 and ϕ_2, and it requires $\tilde{O}(p^{1/4})$ bit operations [13]. It follows that the entire algorithm runs in $\tilde{O}(p^{1/2})$ operations on average, with the cost dominated by the first step, i.e., the search for paths to subfield curves.

Solver. To our knowledge, a precise complexity analysis of the Delfs–Galbraith algorithm has not been conducted. We fill this gap by presenting an optimised implementation of the Delfs–Galbraith algorithm, called Solver, and conducting experiments over many thousands of instances of the subfield search problem to determine its concrete complexity. Though Solver finds the full path, we focus on the optimisation and complexity of the bottleneck step: finding subfield curves. These optimisations include:

- *Choice of ℓ.* In their high-level description of the algorithm, Delfs and Galbraith do not specify which ℓ-isogeny graph to walk in. Framing the problem of taking a step in the ℓ-isogeny graph as computing the roots of a polynomial of degree ℓ, in Solver we chose the simplest and most efficient choice: $\ell = 2$.
- *Fast square root finding in $\mathbb{F}_{p^2}$.* We use the techniques presented in [25, §5.3] to construct an optimised algorithm for finding square roots in $\mathbb{F}_{p^2}$, which only requires two $\mathbb{F}_p$ exponentiations and a few $\mathbb{F}_p$ multiplications and additions.
- *Random walks in the 2-isogeny graph.* We implement a depth-first search to find subfield nodes in the 2-isogeny graph and give a precise complexity analysis on the number of $\mathbb{F}_p$ operations required.

SuperSolver. The main contribution of this paper is a new state-of-the-art algorithm for solving the general supersingular isogeny problem, called SuperSolver. This is a variant of the Delfs–Galbraith algorithm that exploits a combination of our new *subfield root detection* algorithm and the use of modular polynomials. We show that we can efficiently determine whether a polynomial $f \in L[X]$ has a root in a subfield $K \subset L$, without finding any roots explicitly. Though this algorithm works for general fields and polynomials (and may be of use in other contexts), we apply it to the case where $f = \Phi_{\ell,p}(X, j) \in \mathbb{F}_{p^2}[X]$, i.e., where f is the ℓ-th modular polynomial evaluated at a supersingular j-invariant. This provides a means of quickly determining whether there is an ℓ-isogeny connecting the corresponding elliptic curve to a subfield curve: we develop this NeighbourInFp subroutine in Section 4, and use it as the core of our SuperSolver algorithm in Section 5.

In Section 7, we conduct extensive experiments using both our Solver and SuperSolver libraries, all of which show that SuperSolver performs much faster than Solver. In Table 1, we give a taste of the types of improvements we see in searching for subfield nodes over supersingular sets of various sizes, taking a number of primes from the isogeny-based literature. These primes were specifically chosen because the Delfs–Galbraith algorithm for the general supersingular isogeny problem is the best known classical attack against the cryptosystems they target.

Our Solver and SuperSolver algorithms are written in Sage [30] and Python and can be found at

`https://github.com/microsoft/SuperSolver`.

Cryptographic implications. This paper has implications on the classical bit-security of any supersingular isogeny-based scheme for which the Delfs–Galbraith algorithm is the best known attack; this includes the key exchange scheme B-SIDH [11], the signature scheme in [16, §4], and the signature scheme SQISign [15]. For any proposed instantiation of such schemes, our SuperSolver suite allows the analysis in Section 7 to be conducted on input of any prime p, and determines a precise estimate on the number of operations required (on average) to solve the corresponding supersingular isogeny problem. This is especially accurate when the cardinality of the class group is known, which has recently been shown to be feasible for primes up to 512 bits [5]. On the other hand, we point out that the improvements in this paper have no direct impact on the classical security of SIDH [14] and SIKE [19]. Though the Delfs–Galbraith algorithm can be used to attack any supersingular isogeny-based cryptosystem, there are much faster *claw-finding* algorithms (see [14,1]) for solving the special instances of isogeny problems that arise in those schemes.

Roadmap. We give the preliminaries in Section 2. In Section 3, we present our optimised instantiation of the traditional Delfs–Galbraith algorithm, called Solver. In Section 4, we construct an efficient algorithm to detect whether a polynomial has a root in a subfield. We use this algorithm to build SuperSolver in Section 5. In Section 6, we present a worked example to highlight the differences between both algorithms, and in Section 7 we present a number of implementation results that illustrate the concrete improvements offered by SuperSolver.

2 Preliminaries

In this section we briefly set notation and give the requisite background for this paper. Readers familiar with the paragraph headings below are welcome to skip to the final two paragraphs.

	Solver		SuperSolver		
Prime p	Nodes inspected	$\mathbb{F}_p$-mults. per node	Fastest Sets	Nodes inspected	$\mathbb{F}_p$-mults. per node
B-SIDH-p247 [11]	248,913	402	{3,5,7,9,11}	1,716,751	55.6
			{3,5,7,9,11,13}	1,727,601	56.0
			{3,5,7,11,13}	1,731,625	57.8
TwinSmooth-p250 [12]	233,507	427	{3,5,7,9,11}	1,680,337	59.1
			{3,5,7,9,11,13}	1,699,825	59.1
			{3,5,7,11,13}	1,697,769	59.8
SQISign-p256 [15]	248,915	403	{3,5,7,9,11}	1,716,751	55.7
			{3,5,7,9,11,13}	1,727,601	55.9
			{3,5,7,11,13}	1,731,625	57.8
TwinSmooth-p384 [12]	163,331	610	{3,5,7,9,11,13}	1,529,025	63.1
			{3,5,7,9,11}	1,464,709	65.1
			{3,5,7,9,11,13,17}	1,487,919	65.4
TwinSmooth-p512 [12]	127,511	784	{3,5,7,9,11,13}	1,397,761	69.1
			{3,5,7,9,11,13,17}	1,391,645	70.0
			{3,5,7,9,11,13,19}	1,351,509	72.1

Table 1. The number of nodes inspected per 10^8 field multiplications and for primes targeting schemes where Delfs–Galbraith is the best known classical attack. The Solver column corresponds to optimised Delfs–Galbraith walks in $\mathcal{X}(\bar{\mathbb{F}}_p, 2)$ – see Section 3. The SuperSolver columns correspond to enabling our fast subfield root detection algorithm with the three fastest sets of ℓ's (top to bottom) – see Section 5. We also give the approximate number of $\mathbb{F}_p$-multiplications per node inspected at each step, as computed during the precomputation phase that predicts which sets will perform fastest; for Solver we have use the empty set $\{\}$.

Modular polynomials. We will use $\Phi_\ell(X, Y) \in \mathbb{Z}[X, Y]$ to denote the classical modular polynomial (see [29]) that parameterises pairs of elliptic curves with cyclic ℓ-isogeny in terms of their j-invariants: $\Phi_\ell(j_1, j_2) = 0$ if and only if j_1 and j_2 are the j-invariants of ℓ-isogenous elliptic curves. Readers unfamiliar with modular polynomials are encouraged to look at Sutherland's database[4], which contains $\Phi_\ell(X, Y)$ for all $\ell \leq 300$ and for all primes $\ell \leq 1000$. The polynomial Φ_ℓ is symmetric in X and Y, i.e., $\Phi_\ell(X, Y) = \Phi_\ell(Y, X)$, and if $\ell = \prod_{i=1}^{n} \ell_i^{e_i}$ is

[4]See [28], a database computed using techniques from various joint works of his [29,6].

ℓ's prime decomposition, the degree of $\Phi_\ell(X,Y)$ in both X and Y is

$$N_\ell := \deg\left(\Phi_\ell(X,Y)\right) = \prod_{i=1}^{n}(\ell_i+1)\ell_i^{e_i-1}. \tag{1}$$

The difficulty in computing $\Phi_\ell(X,Y)$ is in the size, rather than the number, of its coefficients. As discussed in [29], storing $\Phi_\ell(X,Y)$ requires $O(\ell^3 \log \ell)$ bits, which corresponds to several gigabytes for $\ell \approx 1000$ and many terabytes for $\ell \approx 10^4$. Fortunately, for our purposes, the modular polynomials already contained in Sutherland's database are more than sufficient. Moreover, we will be using them in the context of cryptanalysing instances of the supersingular isogeny problem over a fixed finite field $\mathbb{F}_{p^2}$, meaning we can reduce all of the large coefficients modulo p as a precomputation. Indeed, even before the target j-invariants are known, $\Phi_\ell(X,Y) \in \mathbb{Z}[X,Y]$ will be preprocessed into

$$\Phi_{\ell,p}(X,Y) \in \mathbb{F}_p[X,Y],$$

where we note the additional subscript, defined by reducing all coefficients of $\Phi_\ell(X,Y)$ modulo p. By the symmetry of $\Phi_\ell(X,Y)$, this means we must store around $N_\ell^2/2$ coefficients in $\mathbb{F}_p$, requiring only $O(\ell^2 \log p)$ bits.

Supersingular isogeny graphs. Following [13, §1], let $p > 3$ be a prime and let S_{p^2} denote the set of all supersingular j-invariants in $\mathbb{F}_{p^2}$. The number of such j-invariants is $\#S_{p^2} = \lfloor p/12 \rfloor + b$, where $b \in \{0,1,2\}$ is determined by the value of p mod 12 [27, Theorem V.4.1(c)]. For any positive integer ℓ with $p \nmid \ell$, we use $\mathcal{X}(\bar{\mathbb{F}}_p, \ell)$ to denote the *supersingular isogeny graph* whose nodes correspond to the j-invariants in S_{p^2} and whose edges are ℓ-isogenies defined over $\bar{\mathbb{F}}_p$. When ℓ is prime, these graphs are fully connected [23], and (with the possible exception of a few nodes) are $(\ell+1)$-regular expander graphs that satisfy the Ramanujan property [24]. Crucial to both the Delfs–Galbraith algorithm and this paper is the subset S_p of supersingular j-invariants defined over $\mathbb{F}_p$. The size of this set is $\#S_p = \tilde{O}(p^{1/2})$ [13, Equation 1], and since $\#S_{p^2} = O(p)$, the expected number of randomly chosen elements in S_{p^2} we would have to take before finding one in S_p is in $\tilde{O}(p^{1/2})$.

The Delfs–Galbraith algorithm. The Delfs–Galbraith paper largely focusses on the problem of finding an isogeny $\phi' \colon E_1' \to E_2'$ between two supersingular curves, $E_1'/\mathbb{F}_p$ and $E_2'/\mathbb{F}_p$, whose j-invariants are in S_p. One of their main results is an algorithm [13, Algorithm 1] that computes such a ϕ' in $\tilde{O}(p^{1/4})$ bit operations. At the end of their paper [13, Section 4], they show how this can be used as a subroutine to give an algorithm for the general supersingular isogeny problem, which asks to find an isogeny

$$\phi \colon E_1 \to E_2$$

between two supersingular curves, $E_1/\mathbb{F}_{p^2}$ and $E_2/\mathbb{F}_{p^2}$, whose j-invariants are in S_{p^2}. The idea is to perform simple non-backtracking random walks in $\mathcal{X}(\bar{\mathbb{F}}_p, \ell)$

until hitting an elliptic curve with a j-invariant defined over $\mathbb{F}_p$. Finding a walk from $E_1/\mathbb{F}_{p^2}$ to $E_1'/\mathbb{F}_p$ yields an isogeny $\psi_1 \colon E_1 \to E_1'$, and finding a walk from $E_2/\mathbb{F}_{p^2}$ to $E_2'/\mathbb{F}_p$ yields an isogeny $\psi_2 \colon E_2 \to E_2'$. A full isogeny $\phi \colon E_1 \to E_2$ is then found as the composition $\phi = (\hat{\psi}_2 \circ \phi' \circ \psi_1)$, where $\hat{\psi}_2 \colon E_2' \to E_2$ is the dual of ψ_2, and $\phi' \colon E_1' \to E_2'$ is the subfield isogeny above that can be computed in $\tilde{O}(p^{1/4})$ bit operations. The bottleneck in the Delfs–Galbraith algorithm is finding the paths from the curves with $j \in S_{p^2} \setminus S_p$ to the curves with $j \in S_p$. From the above discussion, the number of j-invariants in S_{p^2} we expect to search over before finding one in S_p is $\tilde{O}(p^{1/2})$. Following [13, Section 4], the steps taken in $\mathcal{X}(\bar{\mathbb{F}}_p, \ell)$ are non-backtracking, meaning that one stores the current j-invariant, j_c, and the previous j-invariant j_p. To take the next step, one then chooses one of the $N_\ell - 1$ roots (see Equation 1) of

$$\Phi_\ell(X, j_c)/(X - j_p)$$

at random. Since ℓ and N_ℓ are fixed and small, it follows that the asymptotic complexity of the search for subfield j-invariants is $\tilde{O}(p^{1/2})$. Before presenting our improved search for subfield j-invariants, in Section 3 we present an optimised version of this algorithm, and subsequently replace the $\tilde{O}$ above with a precise, concrete complexity.

Factoring polynomials in finite fields. Let $f(X) \in \mathbb{F}_q[X]$ be a monic polynomial of degree ℓ with $q = p^k$ for a prime p, and for the purposes of this paper, assume that p is very large (i.e., cryptographically sized) and ℓ is relatively small (i.e., $\ell < 100$). The literature contains a number of methods for finding the irreducible factors of f in $\mathbb{F}_q[x]$, and we briefly mention the most applicable and well-known algorithms for our scenario. Berlekamp's algorithm [4] factors f using an expected number of $O(\ell^3 + \ell^2 \log \ell \log q)$ operations in $\mathbb{F}_q$ [26, Theorem 20.12]. This appears to be superior to the Cantor-Zassenhaus algorithm [8], which uses an expected number of $O(\ell^3 \log q)$ operations in $\mathbb{F}_q$ [26, Theorem 20.9], however one can take advantage of certain time-memory trade-offs to implement Cantor-Zassenhaus so that it requires $O(\ell^3 + \ell^2 \log q)$ operations in $\mathbb{F}_q$ [26, Exercise 20.13]. Note that both of these big-O complexities hide a number of subtleties, that $\mathbb{F}_q$-inversions are included as $\mathbb{F}_q$ operations, and moreover that both of these algorithms are probabilistic. Their deterministic variants have worse complexities [26, §20.6].

Polynomial GCD. Euclid's integer GCD algorithm is easily adapted to compute polynomial GCD's [26, §17.3]. Computing the GCD of two polynomials $g, h \in \mathbb{F}_q[x]$ requires $O(\deg(g) \cdot \deg(h))$ operations in $\mathbb{F}_q$. Again, here each $\mathbb{F}_q$ inversion is counted as an $\mathbb{F}_q$ operation. In order to make our algorithms run as fast as possible, one of the necessary subroutines we derive in Section 4 is an inversion-free polynomial GCD algorithm, for which we state a tight upper bound on the concrete complexity.

Measuring complexity. Throughout this paper we will avoid stating asymptotic (i.e., big-O-style) complexities in favour of stating concrete ones. One of our goals in Section 3 is to replace the $\tilde{O}(p^{1/2})$ complexity of the original Delfs–Galbraith algorithm with a closed formula that can be used to give precise estimates on the classical security of the relevant cryptographic instantiations. We will use the metric of $\mathbb{F}_p$ multiplications as convention, noting that it is relatively straightforward to convert this into a more fine-grained metric (e.g. bit operations, machine operations, cycle counts, gate counts, circuit depth, etc.) depending on the context and on the implementation of the $\mathbb{F}_p$ arithmetic. For simplicity, we will count $\mathbb{F}_p$ squarings as multiplications and ignore additions. We justify this by noting that, roughly speaking, the ratio of multiplications to additions in all of the algorithms in this work are similar, and the complexity of $\mathbb{F}_p$ additions have a minimal impact on any of the aforementioned metrics.

Subfield search complexity determines concrete bit security. Both the Solver implementation detailed in Section 3 and the SuperSolver implementation detailed in Section 5 solve *all* instances of the general supersingular isogeny problem. On input of any prime p and any two supersingular j-invariants in S_{p^2}, both implementations will always terminate with an isogeny that solves the corresponding problem. We emphasise that henceforth our sole focus is on the $\tilde{O}(p^{1/2})$ subfield search phase of the Delfs–Galbraith algorithm. Finding a path between subfield nodes requires $\tilde{O}(p^{1/4})$ operations, which is negligible in both the asymptotic sense and in the sense of obtaining cryptographic security estimates. To see this, suppose the asymptotic $\tilde{O}(p^{1/2})$ complexity of the first phase is replaced by a concrete complexity of $c_p \cdot p^{1/2}$, and the asymptotic $\tilde{O}(p^{1/4})$ complexity of the second phase is replaced by a concrete complexity of $d_p \cdot p^{1/4}$, where c_p and d_p are polynomials in $\log p$. The total complexity of the Delfs–Galbraith algorithm is then

$$c_p \cdot p^{1/2} + d_p \cdot p^{1/4}.$$

For primes of cryptographic size, small changes in c_p have an immediate influence on the total runtime of the algorithm, while much larger changes in d_p will not play a part in the bit security of the problem. For $p > 2^{200}$, a factor 2 change in c_p changes the bit security of the problem by 1, while d_p would have to change by a factor of at least 2^{50} to have the same impact on the bit security.

3 Solver: optimised Delfs–Galbraith subfield searching in $\mathcal{X}(\bar{\mathbb{F}}_p, 2)$

Recall from the previous section that the non-backtracking walks in $\mathcal{X}(\bar{\mathbb{F}}_p, \ell)$ store the current j-invariant, j_c, and the previous j-invariant j_p, and then take a step in $\mathcal{X}(\bar{\mathbb{F}}_p, \ell)$ by choosing one of the $N_\ell - 1$ roots of $\Phi_\ell(X, j_c)/(X - j_p)$. In determining the asymptotic $\tilde{O}(p^{1/2})$ complexity of these walks, Delfs and Galbraith did not need to analyse the cost of a single step. However, to set the stage for our improved search in Section 5, we must optimise this process and determine its concrete cost. The first parameter that must be specified is ℓ, i.e., the isogeny graph to walk around in. Considering both Equation (1) and the complexity of the factorisation algorithms in Section 2, we chose $\ell = 2$ to obtain most efficient and simplest choice where we are able to take advantage of fast explicit methods for computing square roots in $\mathbb{F}_{p^2}$.

Scott's fast square roots in $\mathbb{F}_{p^2}$. Optimal computation of square roots in extension fields of large characteristic requires careful attention to detail. A 2013 paper by Adj and Rodríguez-Henríquez [2] cost the process of computing square roots in $\mathbb{F}_{p^2}$ at two $\mathbb{F}_p$ residuosity tests, two $\mathbb{F}_p$ square roots, and one $\mathbb{F}_p$ inversion, for a total of five exponentiations in $\mathbb{F}_p$. In [25, §5.3], Scott shows that these operations can be combined in a clever way to significantly reduce this cost. The inputs into the Tonelli-Shanks $\mathbb{F}_p$ square root algorithm [22, Algorithm 3.34] can be tweaked in such a way that the two residuosity tests are absorbed into the two square roots. Moreover, he shows that most of the inversion cost can also be absorbed by application of Hamburg's combined 'square-root-and-inversion' trick [17]. This reduces the bulk of cost of an $\mathbb{F}_{p^2}$ square root from five $\mathbb{F}_p$ exponentiations to just two. In addition, there are a handful of $\mathbb{F}_p$ multiplications and additions that either update the Tonelli-Shanks outputs depending on the residuosity outcomes or collect and combine the results according to the "complex" formula in [25, §5.3]. We use this to construct a general square root algorithm in our implementation that is highly optimised with respect to the number of $\mathbb{F}_p$ operations it incurs[5].

Taking a step in $\mathcal{X}(\bar{\mathbb{F}}_p, 2)$. After stepping from $j_p \in \mathbb{F}_{p^2}$ to $j_c \in \mathbb{F}_{p^2}$, a non-backtracking walk in $\mathcal{X}(\bar{\mathbb{F}}_p, 2)$ will step to one of two new nodes: j_0 and j_1. These are computed by solving the quadratic equation that arises from the modular polynomial $\Phi_\ell(X, Y)$ with $\ell = 2$:

$$\begin{aligned}\Phi_2(X,Y) = -X^2Y^2 + X^3 + Y^3 + 1488 \cdot (X^2Y + Y^2X) - 162000 \cdot (X^2 + Y^2)\\ + 40773375 \cdot XY + 8748000000 \cdot (X+Y) - 157464000000000.\end{aligned}$$

The three neighbours of j_c in $\mathcal{X}(\bar{\mathbb{F}}_p, 2)$ are j_p, j_0, and j_1, meaning that $\Phi_2(X, j_c)$ factorises as

$$\Phi_2(X, j_c) = (X - j_p)(X - j_0)(X - j_1).$$

This yields a quadratic equation, whose solutions are j_0, j_1, defined by $X^2 + \alpha X + \beta = 0$, where

$$\begin{aligned}\alpha &= -j_c^2 + 1488 \cdot j_c + j_p - 162000,\\ \beta &= j_p^2 - j_c^2 j_p + 1488 \cdot (j_c^2 + j_c j_p) + 40773375 \cdot j_c - 162000 \cdot j_p + 8748000000.\end{aligned}$$

[5]Note that the fixed exponentiations that take place in the calls to Tonelli-Shanks could be further optimised for a specific p by tailoring a larger window or a different addition chain, but the impact (for our purposes and comparisons) of this improvement would be minor.

Computing these coefficients costs a small, constant number of $\mathbb{F}_p$ operations, so the process of computing both j_0 and j_1 from j_p and j_c boils down to solving the quadratic equation, which essentially requires one $\mathbb{F}_{p^2}$ square root. Since this square root incurs two $\mathbb{F}_p$ exponentiations and a few additional $\mathbb{F}_p$ operations, it follows that the cost of computing each new $j \in S_{p^2}$ during the walks in $\mathcal{X}(\bar{\mathbb{F}}_p, 2)$ is (on average) approximately one $\mathbb{F}_p$ exponentiation.

The depth first search in $\boldsymbol{\mathcal{X}(\bar{\mathbb{F}}_p, 2)}$. Repeating the process described above allows us to perform the search for subfield nodes using a depth first search in a binary tree with d levels as follows. We write $j_{m,n}$ for the n-th node at level m, where $0 \leq m \leq d$ and $0 \leq n \leq 2^m - 1$. The first three levels are depicted in Figure 1. We initialise the root node $j_{0,0}$ as the target $j \in S_{p^2}$, and set $j_{1,0}$ and

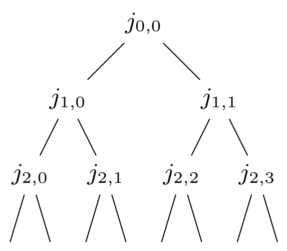

Fig. 1. Levels 0, 1, and 2 of the binary tree in the depth first search of $\mathcal{X}(\bar{\mathbb{F}}_p, 2)$.

$j_{1,1}$ as two of its three neighbours[6] in $\mathcal{X}(\bar{\mathbb{F}}_p, 2)$. The depth first search starts by setting $j_c = j_{1,0}$ and $j_p = j_{0,0}$. We then solve the quadratic equation above to obtain $j_{2,0}$ and $j_{2,1}$, and repeat this procedure with $j_c = j_{i+1,0}$ and $j_p = j_{i,0}$ for $1 \leq i \leq d-1$ until the leftmost leaf $j_{d,0}$ is computed and the `path` stack is fully initialised as

$$\texttt{path} = [j_{0,0},\ j_{1,0}, \ldots, j_{d-1,0},\ j_{d,0}].$$

To avoid any waste, we also maintain a stack of the other solution to the quadratic equations that were computed along the way, which we call sibling nodes

$$\texttt{siblings} = [j_{1,1}, \ldots, j_{d-1,1},\ j_{d,1}].$$

The algorithm then proceeds back up the levels by popping `path` until its last element is the root of a subtree that has not been checked in its entirety. At this point `siblings` is popped and pushed into `path`. When the last element of `path` is the root of a subtree that has not been exhausted, we initialise the process of solving quadratic equations, pushing one of the two solutions into

[6]Initially we do not have a j_p, so all three neighbours can be computed using generic root finding; our code does this during the setup phase.

`path` and the other into `siblings` until `path` contains $d+1$ elements. Each time the quadratic equation solver is called, the two roots (i.e., j-invariants) are immediately checked; if either of them lie in $\mathbb{F}_p$, it is added to `path` and the process is terminated. Otherwise, the process is repeated recursively until `path` $= [j_{0,0}]$, in which case the $2^{d+1}-1$ nodes in the tree have been exhausted without finding a solution. To guarantee that a solution is found, one could increase d and start again, but our code proceeds by simply storing the first (leftmost) leaf and its parent in separate memory so that the process can restart here and avoid recomputing any prior j's. As Delfs and Galbraith point out, setting the depth $d = \frac{1}{2}\log_2 p$ should be enough. Since the number of nodes in the tree is 2^d, increasing d by ϵ makes the failure probability diminish by $1/2^\epsilon$. Setting $\epsilon = 10$ was sufficient in all of our experiments. Finally, as pointed out by Delfs and Galbraith in [13, §4], this process parallelises perfectly. For P processors, one can simply compute a binary tree of depth $\lceil \log_2 P \rceil$ during setup and distribute P of the leaf nodes as individual starting points.

The concrete complexity of Delfs–Galbraith. Table 2 reports on experiments conducted using Solver, the optimised instantiation of the traditional Delfs–Galbraith walk. For each bitlength between 21 and 40, we solved 10,000 instances of the subfield search. In each case we chose 100 random primes and, for each prime, 100 pseudo-random j-invariants in S_{p^2}. The numbers in each column report the averages (as base-2 logarithms) of these search complexities.

Bitlengths of primes p	21	22	23	24	25	26	27	28	29	30
Av. number of nodes visited	8.8	9.4	10.0	10.3	10.9	11.4	11.9	12.3	13.1	13.5
Av. number of $\mathbb{F}_p$-multiplications	14.5	15.0	15.7	16.0	16.7	17.2	17.8	18.2	19.0	19.5
Bitlengths of primes p	**31**	**32**	**33**	**34**	**35**	**36**	**37**	**38**	**39**	**40**
Av. number of nodes visited	13.5	14.2	14.7	15.3	15.8	16.3	17.1	17.3	17.6	18.1
Av. number of $\mathbb{F}_p$-multiplications	19.5	20.5	20.8	21.3	21.9	22.4	23.2	23.6	24.1	24.6

Table 2. The concrete cost of the subfield search phase of the Delfs–Galbraith over small fields of various bitlengths. Further explanation in text.

In all cases the number of $\mathbb{F}_p$ multiplications is found to be

$$\#(\mathbb{F}_p \text{ muls.}) = c \cdot \sqrt{p} \cdot \log_2 p,$$

with $0.75 \leq c \leq 1.05$. In Section 7, we shed more light on the concrete complexity of both Solver and SuperSolver.

Remark 1 (Vélu's formulas). There is no traditional elliptic curve arithmetic found in either Solver or SuperSolver. All of the steps taken within $\mathcal{X}(\bar{\mathbb{F}}_p, 2)$

and the rapid inspections conducted in $\mathcal{X}(\bar{\mathbb{F}}_p, \ell)$ use the modular polynomials. We point out there *may* be specific instances of p where one could perform walks faster than repeatedly solving the $\Phi_{2,p}(X, j)$ quadratic by, say, employing Vélu's formulas [31] with the optimal strategies of De Feo–Jao–Plût [14]. For example, with a prime $p = 2^e 3^f - 1$, the price of computing a 2^e-isogeny (i.e., walking through e nodes in $\mathcal{X}(\bar{\mathbb{F}}_p, 2)$) in this way may be cheaper than the price of computing e square roots in $\mathbb{F}_{p^2}$ (note that the latter reveals 2 nodes each time). However, we argue that these scenarios are likely to only exist for special instances of the supersingular isogeny problem that are geared towards cryptosystems like SIDH [14] and SIKE [19]. As discussed in Section 1, here there are claw-finding algorithms that are much faster than the Delfs–Galbraith algorithm (though the number of $\mathbb{F}_p$ operations required to compute an ℓ^e-isogeny still grows with p, and therefore our fast subfield root detection would also be useful in that context). In the case of both general primes and the types of primes in Table 1, it is highly unlikely that using Vélu's formulas [31] will be competitive with the binary tree depth-first search in $\mathcal{X}(\bar{\mathbb{F}}_p, 2)$; computing general $(\prod \ell_i^{e_i})$-isogenies from kernel elements is much more expensive than ℓ^e-isogenies when $\ell \in \{2, 3\}$, and one travels through fewer nodes in S_{p^2} per $(\prod \ell_i^{e_i})$-isogeny when the ℓ_i grow larger.

Remark 2 (Radical isogenies). Another alternative to solving the quadratic equation that arises from $\Phi_2(X, j_c)/(X - j_p)$ is to instead take steps in $\mathcal{X}(\bar{\mathbb{F}}_p, 2)$ using formulas for radical isogenies [10]. For example, given a supersingular Montgomery curve parameterised as $E_A : y^2 = x^3 + Ax^2 + x$ or as $E_\alpha = x(x - \alpha)(x - 1/\alpha)$, one can compute non-backtracking chains of 2-isogenies as either $A \to A' \to A'' \ldots$, or as $\alpha \to \alpha' \to \alpha'' \ldots$, rather than computing the chain of j-invariants $j \to j' \to j'' \ldots$, as we do. Computing the next value in all of these chains requires one square root (which dominates the cost for primes of cryptographic size) and a small handful of additional field operations, the number of which depends on the choice of chain. In the case of computing the chains $A \to A' \to A'' \ldots$ or $\alpha \to \alpha' \to \alpha'' \ldots$, the number of additional operations are fewer (see [9] and [7]) than those which we incur using the modular polynomial, however we have not opted to exploit this minor speedup for the following reasons. Indeed, it is not true in general that $j(E_A) \in \mathbb{F}_p$ implies $A \in \mathbb{F}_p$ or that $j(E_\alpha) \in \mathbb{F}_p$ implies $\alpha \in \mathbb{F}_p$. Since $j(E_A) = 256(A^2-3)^3/(A^2-4)$, in general there are six values of A corresponding to a given j. Similarly, since $j(E_\alpha) = 256(\alpha^4 - \alpha^2 + 1)^3/(\alpha^4(\alpha^2 - 1)^2)$, in general there are twelve values of α corresponding to a given j. For large primes it is typically the case that most (or all) of the A's and α's corresponding to a given $j \in S_p$ are not defined over $\mathbb{F}_p$. Thus, if radical isogenies were used to compute chains of α's or A's in the context of Delfs–Galbraith, we would need to compute a value that determines whether the corresponding j lies in $\mathbb{F}_p$. We note that this can be achieved without inverting the denominators in the expressions for $j(E_\alpha)$ or $j(E_A)$, i.e., $(a+b\cdot\beta)/(c+d\cdot\beta)$ is in $\mathbb{F}_p$ if and only if $ad = bc$ for $a, b, c, d \in \mathbb{F}_p$ and $\mathbb{F}_{p^2} = \mathbb{F}_p(\beta)$. Thus, the original Delfs–Galbraith walk in $\mathcal{X}(\bar{\mathbb{F}}_p, 2)$ is likely to save a small, fixed number of multiplications per 2-isogeny by computing chains of A's or α's instead of j's. However, when invoking our fast subfield root detection in the sections that follow, it is critical (for Algorithm 2) that the j-invariants of each node are computed explicitly, so that the higher ℓ-degree modular polynomials can be used to probe for ℓ-isogenous subfield neighbours. This subsequent computation of the j-invariant seems to require an additional field exponentiation (we could not see a way to merge the square roots and inversions into one exponentiation in these instances), which would kill the potential advantage of radical isogenies in the optimised SuperSolver algorithm.

Remark 3 (Alternative modular functions). There are several well-known modular functions other than the j-function – see [29]. A natural question in the context of this paper is whether any such functions can be used to make the search for subfield nodes in supersingular isogeny graphs more efficient. For example, the modular polynomials for Weber's f-function [28] are the same degree as those of the j-function, but have *much* smaller coefficients, many of which are zero. If these more compact modular polynomials could be used in the same way as those for the j-function, the practical gains would be significant. However, their applicability in the context of SuperSolver appears to be hampered by reasons similar to those discussed in Remark 2. Weber's f is related to j via $j = (f^{24} - 16)^3/f^{24}$, meaning there can be as many as 72 f's corresponding to a single j-invariant, and it is not true in general that given $j \in \mathbb{F}_p$, the corresponding $f \in \mathbb{F}_p$. Although this makes the Weber polynomials unreliable replacements in the context of the SuperSolver algorithm, our search for alternative modular functions that would be compatible with SuperSolver was far from exhaustive, and it is likely that the j-function is not optimal across all of them. We leave any further investigations in this direction as future work.

4 Fast subfield root detection

In this section we derive a method for determining whether a polynomial $f(X) = a_nX^n + ... + a_1X + a_0 \in \mathbb{F}_{q^d}[X]$ with $d \geq 2$ has a root lying in the subfield $\mathbb{F}_q$, where q is a power of prime p. Though this can be achieved by factoring the polynomial, the methods described in Section 2 become too costly for our purposes; the number of $\mathbb{F}_q$ operations required depends on the size of q, which hampers their relative efficiency as q grows large. Our aim in this section is to detail a much faster algorithm that detects whether a root lies in a subfield and show that the number of $\mathbb{F}_q$ operations required by our algorithm only depends on the degree of f and the degree of the extension d.

As the algorithms in this section may be of independent interest, we leave them as general as possible before specialising back to the application at hand in Section 5. The results up to Proposition 1 are presented for general finite field extensions of the form $\mathbb{F}_{q^d}/\mathbb{F}_q$, but we will later specialise to the quadratic extensions of prime fields, i.e., where $q = p$ and $d = 2$. The inversion-free GCD in Algorithm 1 is derived for an arbitrary polynomial ring $K[x]$, but we will only need to use it in $\mathbb{F}_p[x]$.

In this section, for a polynomial in $\mathbb{F}_{q^d}[X]$, we will reduce the the problem of detecting a root in $\mathbb{F}_q$ to computing the greatest common divisor of d related polynomials $g_1, ..., g_d$. In the case where $d > 2$, we will need to compute the GCD of more than two polynomials. This can be done by recursively computing the GCD of two polynomials and using the following identity:

$$\gcd(g_1, g_2, ..., g_d) = \gcd(g_1, \gcd(g_2, ..., g_d)). \tag{2}$$

We aim to minimise the number of $\mathbb{F}_q$ multiplications needed to compute the GCD and so we construct these polyomials so that they are defined over $\mathbb{F}_q$. To achieve this, we will will need two results. The first is a theorem by Lidl and Niederreiter [21, Theorem 2.24].

Theorem 1. *Let F be a finite extension of a finite field K, both considered as vector spaces over K. Then the linear transformations from F into K are exactly the mappings $L_\beta(\alpha)$, for $\beta \in F$, where $L_\beta(\alpha) = Tr_{F/K}(\beta\alpha)$ for all $\alpha \in F$. Furthermore, we have $L_\beta \neq L_\gamma$ whenever β, γ are distinct elements of F.*

The second result we will need is the following lemma.

Lemma 1. *For $n \in \mathbb{N}$, let $f_1, ..., f_n \in \mathbb{F}_{q^d}[X]$ be polynomials and $A \in \mathrm{GL}_n(\mathbb{F}_{q^d})$. Defining $(g_1, ..., g_n) := A \cdot (f_1, ..., f_n)$, we have*

$$\gcd(f_1, ..., f_n) = \gcd(g_1, ..., g_n).$$

Proof. If a polynomial $h \in \mathbb{F}_{q^d}[X]$ divides $f_1, ..., f_n$, then h divides any linear combination of the $f_1, ..., f_n$. Therefore, h divides $g_1, ..., g_n$. Since A is invertible, by swapping the roles of g_i and f_i we see that the converse holds. □

We are now ready to present the main result of this section.

Proposition 1. *For some $d \geq 2$, let π be the q-power Frobenius endomorphism in $\mathrm{Gal}(\mathbb{F}_{q^d}/\mathbb{F}_q)$ and consider a polynomial $f(X) = a_nX^n + ... + a_1X + a_0 \in \mathbb{F}_{q^d}[X]$. Let β be a primitive element of the extension $\mathbb{F}_{q^d}/\mathbb{F}_q$, in the sense that the field extension is generated by a single element β, i.e., $\mathbb{F}_q(\beta) = \mathbb{F}_{q^d}$. For $i = 1, .., d$, define the following polynomials over $\mathbb{F}_{q^d}$:*

$$g_i := \sum_{j=0}^{d-1} \pi^j(\beta^{i-1}f).$$

Then $g_i(X) \in \mathbb{F}_q[X]$, and $\gcd(g_1, ..., g_d)$ divides f. In particular, if $\gcd(g_1, ..., g_d)$ is of degree 1, then f has a root in $\mathbb{F}_q$. Furthermore, if $\gcd(g_1, ..., g_d) = 1$, then $f(X)$ does not have any roots in $\mathbb{F}_q$.

Proof. Using the notation in Theorem 1, we have

$$\begin{aligned} g_i(X) &= [(\beta^{i-1}a_n + \cdots + \pi^{d-1}(\beta^{i-1}a_n))X^n + \cdots + (\beta^{i-1}a_0 + \cdots + \pi^{d-1}(\beta^{i-1}a_0))] \\ &= \sum_{m=0}^{n} L_{\beta^{i-1}}(a_m)X^m. \end{aligned}$$

By Theorem 1, for all $i = 1, ..., d$ and $m = 0, \dots n$, we have $L_{\beta^{i-1}}(a_m) \in \mathbb{F}_q$, implying that $g_i(X) \in \mathbb{F}_q[X]$. Setting $(d \times d)$ matrix A to be

$$A = \begin{bmatrix} 1 & 1 & \dots & 1 \\ \beta & \pi(\beta) & \dots & \pi^{d-1}(\beta) \\ \vdots & \vdots & \ddots & \vdots \\ \beta^{d-1} & \pi(\beta^{d-1}) & \dots & \pi^{d-1}(\beta^{d-1}) \end{bmatrix} = \begin{bmatrix} 1 & 1 & \dots & 1 \\ \beta & \beta^q & \dots & \beta^{q^{d-1}} \\ \vdots & \vdots & \ddots & \vdots \\ \beta^{d-1} & (\beta^{d-1})^q & \dots & (\beta^{d-1})^{q^{d-1}} \end{bmatrix},$$

we have $(g_1, ..., g_d) := A \cdot (f, \pi(f) ..., \pi^{d-1}(f))$. As for Vandermonde matrices [18, §6.2], we find $\det(A) = \prod_{0 \leq i < j \leq d-1} (\beta^{q^j} - \beta^{q^i})$, which is non-zero for β a primitive element of the extension $\mathbb{F}_{q^d}/\mathbb{F}_q$ and so $A \in \mathrm{GL}_d(\mathbb{F}_{q^d})$. By Lemma 1, we have

$$\gcd(f, \pi(f), ..., \pi^{d-1}(f)) = \gcd(g_1, ..., g_d),$$

therefore $\gcd(g_1, ..., g_d) \mid f$. If $\gcd(g_1, ..., g_d)$ is of degree 1, then $(X - r) \mid f$ for some $r \in \mathbb{F}_q$, and so f has a root in $\mathbb{F}_q$.

We further note that $\gcd(f, \pi(f), ..., \pi^{d-1}(f))$, and therefore $\gcd(g_1, ..., g_d)$, is precisely the largest divisor of f that is defined over $\mathbb{F}_q$. As a result, if $\gcd(g_1, ..., g_d) = 1$, then $f(X)$ does not have any roots in $\mathbb{F}_q$. $\square$

Applying Proposition 1 to detect subfield nodes. The proof of Proposition 1 tells us that $\gcd(g_1, ..., g_d)$ is precisely the largest divisor of $f \in \mathbb{F}_{q^d}[X]$ that is defined over $\mathbb{F}_q[X]$. In our target application of searching for subfield nodes in large supersingular isogeny graphs, i.e., when $d = 2$ and $q = p$, we will most commonly encounter $\gcd(g_1, g_2) = 1$, which immediately rules out subfield neighbours in the ℓ-isogeny graph. Non-trivial GCD's will, with overwhelmingly high probability, be of degree 1 and reveal a single subfield node; this is why our implementation of Algorithm 1 below terminates and returns `true` when the degree of the GCD is 1.

For large supersingular isogeny graphs, the only way for the degree of $\gcd(g_1, g_2)$ to be larger than 1 is when a given j-invariant is ℓ-isogenous to multiple subfield nodes, or when a given j-invariant is ℓ-isogenous to conjugate j-invariants in $\mathbb{F}_{p^2}$.[7]

In our scenario where $d = 2$, we see that $\pi(\beta) + \beta = 0$, meaning that $\pi^k(\beta) = (-1)^k \beta$. As a result, to detect a subfield root, we compute $\gcd(g_1, \beta g_2)$ where $g_1 = f + \pi(f)$ and $g_2 = f - \pi(f)$. In this case we do not need to calculate any more powers of β and we only need to do one GCD computation.

Inversion-free polynomial GCD. To complete the detection of roots in a subfield, we must compute the GCD of polynomials in polynomial ring $K[X]$,

[7] A real-world attack should check any non-trivial GCD, since either of these scenarios are a win for the cryptanalyst; the latter case reveals information about the secret endomorphism ring of the target isomorphism class (see [20, §5.3]), and the former case gives multiple solutions to the subfield search problem.

where K is a field. In Algorithm 1, we modify Euclid's polynomial-adapted algorithm [26, §17.3] to compute the GCD of two polynomials $g, h \in K[X]$ while avoiding inversions in K. We use $\mathrm{LC}(f)$ to denote the leading coefficient of the polynomial f. Note that, for the purposes of incorporating it into our target application of subfield searching in the next section, the algorithm outputs the boolean `true` when the GCD has degree 1 in $K[X]$.

Algorithm 1 InvFreeGCD(): Inversion-free GCD

```
Input: Polynomials $g, h \in K[X]$, such that $\deg g \geq \deg h$
1: Initialise $r, s \leftarrow \mathrm{LC}(h) \cdot g, \mathrm{LC}(g) \cdot h$
2: while $\deg r \geq 1$ and $r \neq s$ do
3:     $r \leftarrow r - X^{\deg r - \deg s} \cdot s$
4:     $r, s \leftarrow \mathrm{LC}(s) \cdot r, \mathrm{LC}(r) \cdot s$
5:     if $\deg r \leq \deg s$ then
6:         $r, s \leftarrow s, r$
7: return $\neg(\deg r = 1$ and $r \neq s)$
```

Proposition 2. *Given input $g, h \in K[X]$ such that $\deg g \geq \deg h$, Algorithm 1 terminates using at most*

$$\frac{1}{2}(\deg g + \deg h + 2)(\deg g + \deg h + 3) - 6$$

multiplications in K.

Proof. Line 1 incurs at most $\deg g + \deg h + 2$ multiplications in K. Setting $r_0 := r, s_0 := s$, we define this to be loop 0. For $i \geq 1$, we denote by r_i, s_i (where $\deg s_i \geq \deg r_i$) the polynomials in loop i of Lines 2-6. Using this notation, we move to Line 7 when $\deg r_i \leq 1$ or $r_i = s_i$. Now, in loop $i \geq 1$ we replace r_i by $r_i - X^{\deg r_i - \deg s_i} s_i$, meaning $\deg r_{i-1} - \deg r_i \geq 1$, and compute $r_i \cdot LC(s_i)$ and $s_i \cdot LC(r_i)$. This requires $\deg r_i + \deg s_i + 2$ multiplications in K. In the worst case, we have $\deg r_{i-1} - \deg r_i = 1$ for $i \geq 1$, where the number of multiplications will decrease by exactly 1 after each loop. In the final loop we have $\deg r_i, \deg s_i = 1$, so we compute 4 multiplications in K. In summary, in the worst case we begin with $\deg g + \deg h + 2$ multiplications, decreasing by 1 until we get to 4. Therefore, the total number of multiplications is at most $\sum_{n=4}^{\deg g + \deg h + 2} n$, which is the bound above. $\square$

In summary, Proposition 1 shows that detecting subfield roots of $f \in \mathbb{F}_{q^d}[X]$ amounts to computing the GCD of d related polynomials in $\mathbb{F}_q[X]$. We showed that computing this GCD is simpler when $d = 2$. Proposition 2 gives an upper bound on the number of $\mathbb{F}_q$ multiplications required to compute such a GCD in $\mathbb{F}_q[X]$. In the next section we use these tools to build a faster algorithm for finding subfield nodes in supersingular isogeny graphs.

5 SuperSolver: optimised subfield searching with fast subfield root detection in $\mathcal{X}(\bar{\mathbb{F}}_p, \ell)$

SuperSolver is an algorithm which, given two j-invariants in S_{p^2} corresponding to two supersingular curves $E_1/\mathbb{F}_{p^2}$ and $E_2/\mathbb{F}_{p^2}$, will, on average, solve the supersingular isogeny problem with lower concrete complexity than the traditional Delfs–Galbraith Solver algorithm described in Section 3. As in the Delfs–Galbraith algorithm, SuperSolver takes non-backtracking walks in $\mathcal{X}(\bar{\mathbb{F}}_p, 2)$ until they hit a j-invariant in $\mathbb{F}_p$. However, at each step of the random walk, SuperSolver also inspects $\mathcal{X}(\bar{\mathbb{F}}_p, \ell)$, for carefully chosen $\ell > 2$, to efficiently detect whether j has any ℓ-isogenous neighbours in $\mathbb{F}_p$. Traditionally, inspecting $\mathcal{X}(\bar{\mathbb{F}}_p, \ell)$ for a subfield neighbour requires fully factoring a degree-N_ℓ polynomial and determining whether any of the roots lie in $\mathbb{F}_p$. Performing this for each ℓ would require $O(\ell^3 + 2\ell^2 \log p)$ operations in $\mathbb{F}_{p^2}$ using the modified Cantor-Zassenhaus algorithm (see Section 2), which is prohibitively costly. Following the results from Section 4, however, SuperSolver conducts the inspection of $\mathcal{X}(\bar{\mathbb{F}}_p, \ell)$ with $O(\ell^2)$ multiplications in $\mathbb{F}_p$. We make this count precise later in this section. Crucially, the number of $\mathbb{F}_p$ operations is no longer dependent on the size of p, and this means that as p grows large, the set of ℓ's that are optimal to use also grows, and the more profitable (relatively speaking) SuperSolver becomes. We reiterate that, although both Solver and SuperSolver return the full isogeny between $E_1/\mathbb{F}_{p^2}$ and $E_2/\mathbb{F}_{p^2}$, our discussion focusses on the bottleneck problem of finding an isogeny from $E_1/\mathbb{F}_{p^2}$ (resp. $E_2/\mathbb{F}_{p^2}$) to $E_1'/\mathbb{F}_p$ (resp. $E_2/\mathbb{F}_p$). If, at some node j, we detect an ℓ-isogenous neighbour in $\mathbb{F}_p$, SuperSolver will then factorise the degree-N_ℓ polynomial $\Phi_{\ell,p}(X, j)$ to determine the subfield j-invariant. We view this as a post-computation step, since we are only interested in the concrete complexity of the average step taken in the walk (which we assume does not find a subfield node). Note that the paths between $E_1/\mathbb{F}_{p^2}$ and $E_2/\mathbb{F}_{p^2}$ returned by both Solver and SuperSolver both look the same: in general, both start and finish with a chain of 2-isogenies that is connected in the middle by a chain of different prime-degree isogenies. The main difference, as the results in Section 7 illustrate, is that 2-isogeny chains at each end are *much* shorter. Recall that in the original Delfs–Galbraith algorithm, each step consists of finding the roots of a quadratic equation in $\mathbb{F}_{p^2}[X]$, which reveals two neighbouring nodes in $\mathcal{X}(\bar{\mathbb{F}}_p, 2)$. In SuperSolver, after forming a list of carefully chosen $\ell > 2$, each step will also include the rapid inspection of $\mathcal{X}(\bar{\mathbb{F}}_p, \ell)$ for every ℓ in this list. Though the inspection of the neighbours in $\mathcal{X}(\bar{\mathbb{F}}_p, \ell)$ increases the total number of $\mathbb{F}_p$ multiplications at each step, more nodes are checked. We first describe the process of taking a step in SuperSolver, and then move to describing how to choose the list of $\ell > 2$ in order to minimise the number of $\mathbb{F}_p$ multiplications per node inspected.

Remark 4 (Odd ℓ only). With the exception of the leaf nodes in the last level of the binary tree, it is redundant to perform rapid node inspections in $\mathcal{X}(\bar{\mathbb{F}}_p, 2\ell)$ if rapid inspections in $\mathcal{X}(\bar{\mathbb{F}}_p, \ell)$ are also part of the routine, since the latter inspections will detect (or exclude) subfield nodes at the next level of the walk down the tree. We therefore find it optimal to only include odd ℓ_i in the lists constructed at the end of this section. Note that there is no redundancy in including odd composite ℓ_i's in our lists, even if they have proper divisors that are also in the list.

Rapid inspection of the ℓ-isogenous neighbours. Here we describe Algorithm 2: NeighbourInFp. On input of ℓ, $j \in \mathbb{F}_{p^2}$ and p, it outputs `true` if j is ℓ-isogenous to a $j' \in \mathbb{F}_p$, and `false` otherwise. Recall from Equation (1) that the degree of $\Phi_{\ell,p}$ in X and Y is N_ℓ. The first subroutine of NeighbourInFp is EvalModPolyj(ℓ, j, p): it evaluates $\Phi_{\ell,p}(X,Y)$ at $Y = j$ by computing $j^2, ..., j^{N_\ell}$, and then multiplying these by the corresponding coefficients of $\Phi_{\ell,p}$, returning the coefficients $a_{N_\ell}, ..., a_0$ of X in $\Phi_{\ell,p}(X,j)$. Note that, since we typically have a list of multiple ℓ, i.e., $\ell_1 < \cdots < \ell_t$, the powers of j (up to N_{ℓ_t}) are computed once-and-for-all at every j, and recycled among the $\ell_i < \ell_t$. We follow Section 4 to detect whether $\Phi_{\ell,p}(X,j) \in \mathbb{F}_{p^2}[X]$ has a root in $\mathbb{F}_p$. Letting $\beta \in \mathbb{F}_{p^2}$ be such that $\mathbb{F}_{p^2} = \mathbb{F}_p(\beta)$, we first compute the related polynomials

$$g_1 := (1/2) \cdot [\Phi_{\ell,p}(X,j) + \pi(\Phi_{\ell,p}(X,j))] \qquad \text{and}$$
$$g_2 := (-\beta/2) \cdot [\Phi_{\ell,p}(X,j) - \pi(\Phi_{\ell,p}(X,j))],$$

where $\pi \in \mathrm{Gal}(\mathbb{F}_{p^2}/\mathbb{F}_p)$ is the Frobenius endomorphism. By Proposition 1, we have $g_1, g_2 \in \mathbb{F}_p[X]$ and

$$\deg(\gcd(g_1, g_2)) = 1 \implies \Phi_{\ell,p}(X,j) \text{ has a root in } \mathbb{F}_p.$$

We then complete the inspection of $\mathcal{X}(\bar{\mathbb{F}}_p, \ell)$ by using Algorithm 1 to calculate $\gcd(g_1, g_2)$. If $\gcd(g_1, g_2) \neq 1$, then (for large enough p) it is overwhelmingly likely that $\deg(\gcd(g_1, g_2)) = 1$, which is why our implementation uses the degree of the GCD as the criterion for terminating the subfield search. Another possibility is to terminate whenever $\gcd(g_1, g_2)$ is non-constant, and then to inspect the higher degree GCD according to the two possible scenarios discussed in Section 4.

Note that if we have a polynomial $f(X) = a_n X^n + a_{n-1}X^{n-1} + ... + a_1 X + a_0 \in \mathbb{F}_{p^2}[X]$ then

$$\frac{1}{2}[f + \pi(f)] = \mathrm{Re}(a_n)X^n + \mathrm{Re}(a_{n-1})X^{n-1} + ... + \mathrm{Re}(a_1)X + \mathrm{Re}(a_0) \in \mathbb{F}_p[X],$$
$$\frac{-\beta}{2}[f - \pi(f)] = \mathrm{Im}(a_n)X^n + \mathrm{Im}(a_{n-1})X^{n-1} + ... + \mathrm{Im}(a_1)X + \mathrm{Im}(a_0) \in \mathbb{F}_p[X],$$

where, for $a + b\beta \in \mathbb{F}_{p^2}$, $\mathrm{Re}(a + b\beta) = a$ and $\mathrm{Im}(a + b\beta) = b$, in analogy with the notation used for complex numbers. As a result, we can obtain g and h directly from $f = \Phi_{\ell,p}$ by computing

$$g_1 = X^{N_\ell} + ... + \mathrm{Re}(a_0), \quad \text{and} \quad g_2 = \mathrm{Im}(a_{N_\ell - 1})X^{N_\ell - 1} + ... + \mathrm{Im}(a_0).$$

This avoids having to compute any $\mathbb{F}_{p^2}$ multiplications to calculate the related polynomials g_1, g_2.

Algorithm 2 NeighbourInFp(): Detect whether $j \in \mathbb{F}_{p^2}$ is ℓ-isogenous to a $j' \in \mathbb{F}_p$

```
    Input: $\ell, j, p$
 1: $a_{N_\ell}, ..., a_0 \leftarrow$ EvalModPolyj$(\ell, j, p)$
 2: $g_1 \leftarrow X^{N_\ell} + ... + \mathrm{Re}(a_0)$
 3: $g_2 \leftarrow \mathrm{Im}(a_{N_\ell - 1})X^{N_\ell - 1} + ... + \mathrm{Im}(a_0)$
 4: return InvFreeGCD$(g_1, g_2)$
```

Cost of Inspecting the ℓ-isogeny Graph. Evaluating $\Phi_{\ell,p}(X, Y)$ at $Y = j$ with EvalModPolyj requires at most $9N_\ell(N_\ell - 1)$ multiplications in $\mathbb{F}_p$, noting that one $\mathbb{F}_{p^2}$ multiplication is equivalent to 3 $\mathbb{F}_p$ multiplications. By Proposition 2, we compute InvFreeGCD(g_1, g_2) with at most $(2N_\ell + 1)(N_\ell + 1) - 6$ $\mathbb{F}_p$ multiplications. Therefore, for a fixed ℓ, the cost of inspecting $\mathcal{X}(\overline{\mathbb{F}}_p, \ell)$ is

$$\begin{aligned} \mathrm{cost}_\ell &= \frac{1}{N_\ell}[\#\mathbb{F}_p \text{ multiplications needed to inspect } \ell\text{-isogenous neighbours}] \\ &\leq \frac{1}{N_\ell}[11N_\ell^2 - 6N_\ell - 5], \end{aligned}$$

which depends only on ℓ. This means that, for each ℓ, cost_ℓ can be computed once for all primes. In Table 3 we present the ℓ with the lowest cost, ordering them by increasing cost_ℓ from left to right.

ℓ	3	5	7	9	11	13	17	19	15	23
N_ℓ	4	6	8	12	12	14	18	20	24	24
$\mathbb{F}_p$ muls per node	16.3	24.5	32.6	48.8	48.8	56.8	72.8	80.9	96.9	96.9
ℓ	25	29	21	31	27	37	41	43	33	35
N_ℓ	30	30	32	32	36	38	42	44	48	48
$\mathbb{F}_p$ muls per node	120.9	120.9	128.9	128.9	144.9	152.9	168.9	177.0	192.9	192.9

Table 3. The cost of inspecting ℓ-isogenous neighbours, cost_ℓ, for ℓ ordered by increasing cost from left to right.

The important takeaway from Table 3 is that the number of $\mathbb{F}_p$ multiplications incurred by our algorithm does not grow with p. This count is fixed and depends only on ℓ. Looking back at the root solving algorithms in Section 2, we see a stark difference in expected performance. Those algorithms have many constants hidden by the big-O, have a leading ℓ^3 term (compared to our ℓ^2 term), and, importantly, the number of field operations they incur grows as the field grows due to their implicit dependency on $\log p$. Moreover, as mentioned in Section 2, the complexities cited are for *probabilistic* root finding algorithms. Their deterministic variants have even worse complexities [26, §20.6].

Choosing the ℓ_i to minimise the cost of a step. We consider the cost of each step in SuperSolver, which we denote by the ratio

$$\text{cost} = \frac{\text{total \# of } \mathbb{F}_p \text{ multiplications}}{\text{total \# of nodes revealed}}. \tag{3}$$

The aim of this section is to describe how to construct a list of ℓ_i that minimises the cost. Recall from in Table 3 that the ℓ's that give the cheapest cost per node inspected are (from left to right)

$$[3, 5, 7, 11, 13, 9, 17, 19 \ldots]. \tag{4}$$

We will use L_b to denote each list of ℓ_i and cost_{L_b} to denote the corresponding cost, where the bit representation of b specifies the set of ℓ's from Equation (4); the least significant bit of b determines if 3 is included, the second least significant bit of b determines if 5 is included, and so on. For example, $L_0 = \{\}$, $L_2 = \{5\}$, and and $L_{11} = \{3, 5, 7\}$. Each step will always include revealing 2 neighbours in $\mathcal{X}(\bar{\mathbb{F}}_p, 2)$, therefore for a node j we have for each step:

$$\begin{aligned} &\text{total \# of } \mathbb{F}_p \text{ muls.} \geq \#\mathbb{F}_p \text{ muls. needed to find roots of } \Phi_{2,p}(X, j);\\ &\text{total \# of nodes revealed} \geq 2. \end{aligned}$$

Here, equality holds only when we take the list to be L_0, which corresponds to the original Delfs–Galbraith algorithm. Minimising the cost in Equation (3) is a non-trivial task. We first restrict the L_b to only contain ℓ such that $\text{cost}_\ell < \text{cost}_{L_0}$, otherwise it would be more advantageous to take another step by moving to a neighbouring node in $\mathcal{X}(\bar{\mathbb{F}}_p, 2)$. We emphasise that cost_{L_0} grows with p, whereas cost_ℓ stays fixed. This signifies that the condition on ℓ becomes less restrictive as p increases. Suppose that, imposing this condition we get $L_b \subseteq [\ell_1, ..., \ell_n]$. We then exhaust all $b < 2^n$, corresponding to subsets of $[\ell_1, ..., \ell_n]$, to determine the L_b that minimise Equation (3). It is important to note that, as this optimisation depends only on the prime p, L_b can be determined in the precomputation.

6 A worked example

We now use a worked example to illustrate how the Solver and SuperSolver programs solve the supersingular isogeny problem, and to highlight the differences between them. Our SuperSolver suite is written in Sage/Python and a boolean variable `supersolver` specifies whether Solver or SuperSolver is used. For a prime p, and two supersingular j-invariants j_1 and j_2 defined over $\mathbb{F}_{p^2} = \mathbb{F}_p(\beta)$, Solver runs by entering

```
Solver(p, j10, j11, j20, j21, false)
```

and SuperSolver runs by calling

```
Solver(p, j10, j11, j20, j21, true),
```

where `j10` $= \text{Re}(j_1)$, `j11` $= \text{Im}(j_1)$ and similary for `j20`, `j21`.

We picked

$$p = 2^{20} - 3,$$

the smallest of the primes from Table 4 (of Section 7), and generated two pseudo-random[8] j-invariants in $S_{p^2} \setminus S_p$:

$$j_1 = 129007\beta + 818380 \qquad \text{and} \qquad j_2 = 97589\beta + 660383.$$

Preprocessing. The preprocessing phase of both programs starts by constructing the extension field $\mathbb{F}_{p^2} = \mathbb{F}_p(\beta)$, where β^2 is the first non-square in the sequence $-1,-2,2,-3,3,\ldots$. It then computes a list of constants for the Tonelli-Shanks subroutine, most notably the exponent $(p-2^e-1)/2^{e+1}$, where e is the maximum integer such that $2^e \mid (p-1)$. This exponent is Scott's 'progenitor' [25, p. 3], which essentially determines the complexity of $\mathbb{F}_p$ square roots, and therefore of $\mathbb{F}_{p^2}$ square roots. As a result, it determines the cost of taking a step in $\mathcal{X}(\bar{\mathbb{F}}_p, 2)$ – see Section 3. The preprocessing phase then computes a set of integers $\ell \geq 3$ (according to the optimisations in Section 5 and the relevant heuristics in [13]), fetches the associated files (originally from Sutherland's database [28]) containing $\Phi_\ell(X,Y) \in \mathbb{Z}[X,Y]$ and reduces all of the coefficients to store a set of new, more compact files containing elements of $\mathbb{F}_p$ that define each of the $\Phi_{\ell,p}(X,Y) \in \mathbb{F}_p[X,Y]$. Note that this is done for both Solver and SuperSolver, since both of these programs use the original Delfs–Galbraith subfield path algorithm [13, Algorithm 1] after the searches for subfield nodes is complete. It is important to note, especially in the cryptanalytic context, that all of these preprocessing steps only depend on p and can therefore be done without knowledge of j_1 and j_2.

Solver. The optimised walk in $\mathcal{X}(\bar{\mathbb{F}}_p, 2)$ proceeds exactly as described in Section 3, i.e., using the depth first search through the binary trees rooted at j_1 and j_2, until both searches find the subfield nodes $j_1' \in \mathbb{F}_p$ and $j_2' \in \mathbb{F}_p$. In the case of our example, paths were found to $j_1' = 760776$ and $j_2' = 35387$, depicted in Figure 2 and Figure 3. They correspond to $\phi_1 \colon E_1 \to E_1'$ and $\phi_2 \colon E_2 \to E_2'$, where $j(E_1) = j_1$, $j(E_1') = j_1'$, $j(E_2) = j_2$, and $j(E_2') = j_2'$.

Solver then computes a connecting path between the subfield nodes following Delfs–Galbraith [13, Algorithm 1]. This is depicted in Figure 4. Solver simply reverses the steps in ϕ_2 to obtain its dual, $\hat{\phi}_2$, and outputs the full path as $\phi \colon E_1 \to E_2$ as $\phi = \hat{\phi}_2 \circ \phi' \circ \phi_1$.

SuperSolver. With $p = 2^{20} - 3$, the preprocessing phase determined that SuperSolver is optimal with $L_3 = \{3, 5\}$ (see also Table 4 in the next section). Before departing the starting node $j_1 = 129007\beta + 818380$, SuperSolver performs the rapid inspection of its 3- and 5-isogenous neighbours as described in Section 5. It then takes steps in $\mathcal{X}(\bar{\mathbb{F}}_p, 2)$ as in Section 3, but at each new node it performs

[8] We do this by taking long walks in $\mathcal{X}(\bar{\mathbb{F}}_p, 3)$ away from a known subfield curve.

$$\begin{tikzcd}
\phi_{\mathbf{1}} : j_1 \arrow[r, "2"] & 219247\beta + 863507 \arrow[r, "2"] & 489342\beta + 132142 \arrow[d, "2"] \\
174188\beta + 794346 \arrow[d, "2"] & 291380\beta + 146098 \arrow[l, "2"] & 148602\beta + 24450 \arrow[l, "2"] \\
263095\beta + 184707 \arrow[r, "2"] & 37438\beta + 90559 \arrow[r, "2"] & 1027930\beta + 498080 \arrow[d, "2"] \\
612554\beta + 208821 \arrow[d, "2"] & 994015\beta + 681197 \arrow[l, "2"] & 206051\beta + 982009 \arrow[l, "2"] \\
649416\beta + 751358 \arrow[r, "2"] & 203489\beta + 43055 \arrow[r, "2"] & 393773\beta + 1028490 \arrow[d, "2"] \\
318158\beta + 140927 \arrow[d, "2"] & 175225\beta + 937858 \arrow[l, "2"] & 971263\beta + 725197 \arrow[l, "2"] \\
348684\beta + 935077 \arrow[r, "2"] & 341898\beta + 405481 \arrow[r, "2"] & 274229\beta + 367729 \arrow[d, "2"] \\
 & & j_1' = 760776
\end{tikzcd}$$

Fig. 2. A walk through $\mathcal{X}(\bar{\mathbb{F}}_p, 2)$ for $p = 2^{20} - 3$ during Solver. The walk starts at $j_1 = 129007\beta + 818380 \in S_{p^2}$ and finds the subfield node $j_1' = 760776 \in S_p$ after 21 steps.

$$\begin{tikzcd}
\phi_{\mathbf{2}} : j_2 \arrow[r, "2"] & 867493\beta + 220256 \arrow[r, "2"] & 252807\beta + 1011175 \arrow[d, "2"] \\
657423\beta + 286117 \arrow[d, "2"] & 440840\beta + 706619 \arrow[l, "2"'] & 953362\beta + 11601 \arrow[l, "2"'] \\
734841\beta + 660440 \arrow[r, "2"] & 919529\beta + 442520 \arrow[r, "2"] & 219960\beta + 646080 \arrow[d, "2"] \\
638727\beta + 940073 \arrow[d, "2"] & 219719\beta + 594710 \arrow[l, "2"'] & 619876\beta + 961666 \arrow[l, "2"'] \\
407014\beta + 868179 \arrow[r, "2"] & 535787\beta + 1046047 \arrow[r, "2"] & 138865\beta + 8726 \arrow[d, "2"] \\
1016378\beta + 696447 \arrow[d, "2"] & 289439\beta + 170877 \arrow[l, "2"'] & 665078\beta + 700037 \arrow[l, "2"'] \\
895198\beta + 793471 \arrow[r, "2"] & 562302\beta + 547814 \arrow[r, "2"] & 68076\beta + 946405 \arrow[d, "2"] \\
 & & j_2' = 35387
\end{tikzcd}$$

Fig. 3. A walk through $\mathcal{X}(\bar{\mathbb{F}}_p, 2)$ for $p = 2^{20} - 3$ during Solver. The walk starts at $j_2 = 97589\beta + 660383 \in S_{p^2}$ and finds the subfield node $j_2' = 35387 \in S_p$ after 21 steps.

$$\begin{tikzcd}
\phi' : j_1' \arrow[r, "31"] & 815910 \arrow[r, "17"] & 848568 \arrow[r, "31"] & 157399 \arrow[r, "29"] & 451011 \arrow[r, "31"] & 820763 \arrow[d, "31"] \\
 & & & j_2' & 286978 \arrow[l, "17"'] & 76159 \arrow[l, "37"']
\end{tikzcd}$$

Fig. 4. A path connecting two subfield j-invariants by taking steps in $\mathcal{X}(\bar{\mathbb{F}}_p, \ell)$ with $\ell \in \{17, 29, 31, 37\}$. The walk starts at $j_1' = 760776 \in S_p$ and connects to $j_2' = 35387 \in S_p$ after 8 steps.

the rapid inspection of the 3- and 5-isogenous neighbours. In our example, both walks found a subfield node after 2 steps in $\mathcal{X}(\bar{\mathbb{F}}_p, 2)$. The walk from j_1 found a 3-isogenous neighbour and the walk from j_2 found a 5-isogenous neighbour. The final step that finds ϕ' is implemented in SuperSolver exactly as it was for Solver. The three isogenies ϕ_1, ϕ_2, and ϕ', comprising the full isogeny $\phi = \hat{\phi}_2 \circ \phi' \circ \phi_1$, are depicted in Figure 5.

$$\begin{tikzcd}
\boldsymbol{\phi_1} : \ j_1 \arrow[r, "2"] & 219247\beta + 863507 \arrow[r, "2"] & 489342\beta + 132142 \arrow[r, "3"] & j_1' = 35387 \\
\boldsymbol{\phi_2} : \ j_2 \arrow[r, "2"] & 867493\beta + 220256 \arrow[r, "2"] & 252807\beta + 1011175 \arrow[r, "5"] & j_2' = 292917
\end{tikzcd}$$

$$\begin{tikzcd}
\phi' : \ j_1' \arrow[r, "17"] & 658300 \arrow[r, "29"] & 343840 \arrow[r, "31"] & 560315 \arrow[d, "17"] \\
 & & j_2' & 439276 \arrow[l, "37"']
\end{tikzcd}$$

Fig. 5. The three paths found comprising an isogeny from E_1 to E_2 as found by SuperSolver.

To illustrate the core idea in this paper, we focus on the isogeny ϕ_1 depicted at the top of Figure 5 and walk through the steps of the NeighbourInFp algorithm. Evaluating the third modular polynomial at the intermediate j-invariants (Step 1 of Algorithm 2) yields

$$\begin{aligned}
\Phi_{3,p}(X,\, 219247\beta + 863507) &= X^4 + (212814\beta + 479338)X^3 + (408250\beta + 920025)X^2 \\
&\quad + (811739\beta + 93038)X + 942336\beta + 847782; \\
\Phi_{3,p}(X,\, 489342\beta + 132142) &= X^4 + (872004\beta + 13960)X^3 + (1031755\beta + 822066)X^2 \\
&\quad + (969683\beta + 747785)X + 813010\beta + 255391.
\end{aligned}$$

Though the theory tells us that these two polynomials split over $\mathbb{F}_{p^2}[X]$, to the naked eye there is no way to distinguish which (if any) of these polynomials has a root in $\mathbb{F}_p$. In both cases, setting $g_1 = 1/2 \cdot (\Phi_{3,p} + \pi(\Phi_{3,p}))$ (Step 2 of Algorithm 2) and $g_2 = -\beta/2 \cdot (\Phi_{3,p} - \pi(\Phi_{3,p}))$ (Step 3 of Algorithm 2) respectively yields

$$\begin{aligned}
g_1 &= X^4 + 479338X^3 + 920025X^2 + 93038X + 847782; \\
g_2 &= 425628X^3 + 816500X^2 + 574905X + 836099,
\end{aligned}$$

and

$$\begin{aligned}
g_1 &= X^4 + 13960X^3 + 822066X^2 + 747785X + 255391; \\
g_2 &= 695435X^3 + 1014937X^2 + 890793X + 577447.
\end{aligned}$$

In the first case, Step 4 of Algorithm 2 outputs $\gcd(g_1, g_2) = 1$, meaning that $\Phi_{3,p}(X,\, 219247\beta + 863507)$ has no subfield roots. In the second case, we see $\gcd(g_1, g_2) = X + 1013186$, meaning that $-1013186 = 35387$ is a subfield root. In our example, we note that the total number of steps between j_1 and j_2 returned by SuperSolver is 10, which is much shorter than the 50 steps taken by Solver. Since the middle subfield path finding algorithm is the same in both routines, there is no guarantee that the total path will always be smaller for SuperSolver. It is worth pointing out, however, that the two *outer* paths from elements in $S_{p^2} \setminus S_p$ to S_p (i.e., ϕ_1 and ϕ_2) returned by SuperSolver will never be longer than those returned by Solver. Indeed, Solver can be viewed as a special case of SuperSolver where the list of ℓ's is chosen to be L_0. Finally, we note that both Solver and SuperSolver always conclude by checking the correctness of the full path from j_1 to j_2.

7 Implementation results

In this section we present some experimental results highlighting the efficacy of SuperSolver. The experiments focus solely on the search for subfield nodes (i.e., the bottleneck step of Delfs–Galbraith) and come in two flavours: many j-invariants over small primes, and one j-invariant over a large, cryptographic prime.

Small primes and many walks. Table 4 and Table 5 report experiments that were run on the largest primes of the 30 bitlengths from 20 to 49. We started at 5000 pseudo-random[9] supersingular j-invariants in $S_{p^2} \setminus S_p$ for the primes of bitlengths 20-24, at 1000 j's for the primes of bitlengths 25-29, at 500 j's for the primes of bitlengths 30-34, at 100 j's for the primes of bitlengths 35-39, at 50 j's for the primes of bitlengths 40-44, and at 10 j's for the primes of bitlengths 45-49. For every j, we ran both Solver and SuperSolver (with the five sets of ℓ's that were predicted to perform best during preprocessing) until all walks hit a subfield j-invariant. Throughout, we will denote these fast sets of ℓ's by L_b, as in Section 5. In all cases we counted the exact number of $\mathbb{F}_p$ multiplications, squarings and additions required to find the subfield node. Following our metric in Section 2, Table 5 reports the average number of $\mathbb{F}_p$ multiplications by counting squarings as multiplications, and highlights in **bold** which of the five predicted sets of ℓ's performed best on average.

Table 4 reports the average number of nodes visited in each of the walks, along with $\lceil \#S_{p^2}/\#S_p \rceil$, the expected number of random elements in S_{p^2} that would need to be sampled to find a subfield element in S_p. Here, the primes are small enough that S_p can be computed precisely (see Section 2). For each prime, Table 4 highlights in **bold** the column that matches up with the least multiplications reported in Table 5. Note that, for SuperSolver, the number of

[9] Just as in Section 6, we used long walks in $\mathcal{X}(\bar{\mathbb{F}}_p, 3)$ away from a known starting curve to achieve uniformity in S_{p^2}.

nodes visited is the number of nodes that are actually walked onto in $\mathcal{X}(\bar{\mathbb{F}}_p, 2)$, not the number of nodes inspected using our fast subfield detection algorithm. Thus, in general, the lowest average number of nodes visited does not correspond to the lowest average number of multiplications. Indeed, the walks with fewer ℓ's spend less compute time inspecting ℓ-isogenous neighbours and therefore move onto new nodes faster, but do not cover as much of the supersingular set during the fast inspection.

					Solver	SuperSolver				
Prime p	p mod8	$\left\lceil \frac{\#S_{p^2}}{\#S_p} \right\rceil$	$\mathbb{F}_p$-mults. per step	Fastest L_j's $[L_{(i)} \dots, L_{(v)}]$	L_0	$L_{(i)}$	$L_{(ii)}$	$L_{(iii)}$	$L_{(iv)}$	$L_{(v)}$
$2^{20}-3$	5	530	54	$[L_3, L_1, L_7, L_5, L_2]$	812	**127**	257	76	107	193
$2^{21}-9$	7	156	53	$[L_3, L_1, L_7, L_5, L_2]$	459	**86**	218	53	87	111
$2^{22}-3$	5	584	60	$[L_3, L_7, L_1, L_5, L_6]$	885	170	108	**288**	146	145
$2^{23}-15$	1	583	71	$[L_3, L_7, L_5, L_6, L_1]$	838	**172**	106	169	121	430
$2^{24}-3$	5	1277	64	$[L_3, L_7, L_5, L_1, L_6]$	1897	**318**	209	311	618	273
$2^{25}-39$	1	1231	71	$[L_3, L_7, L_5, L_1, L_6]$	1873	**360**	223	359	933	259
$2^{26}-5$	3	732	62	$[L_3, L_7, L_1, L_5, L_6]$	1362	352	**194**	691	271	233
$2^{27}-39$	1	2348	73	$[L_3, L_7, L_5, L_6, L_1]$	3455	917	**438**	579	497	1766
$2^{28}-57$	7	2965	64	$[L_3, L_7, L_1, L_5, L_6]$	9748	1788	1022	**3065**	1314	1306
$2^{29}-3$	5	2953	74	$[L_3, L_7, L_5, L_6, L_1]$	4384	1053	**526**	712	603	2161
$2^{30}-35$	5	3965	75	$[L_3, L_7, L_5, L_6, L_1]$	5555	1443	749	**961**	849	2825
$2^{31}-1$	7	9009	75	$[L_3, L_7, L_5, L_6, L_1]$	27103	4501	**2602**	3755	3136	8794
$2^{32}-5$	3	5142	75	$[L_3, L_7, L_5, L_6, L_1]$	10149	2520	**1445**	2108	1702	5335
$2^{33}-9$	7	6638	77	$[L_3, L_7, L_5, L_6, L_1]$	20387	**3832**	2342	3756	2676	10562
$2^{34}-41$	7	10526	78	$[L_3, L_7, L_5, L_6, L_1]$	32640	6443	3790	6094	**4531**	16320
$2^{35}-31$	1	117571	99	$[L_7, L_3, L_5, L_6, L_{23}]$	150101	**14893**	27873	23076	20921	9850
$2^{36}-5$	3	29040	83	$[L_3, L_7, L_5, L_6, L_{23}]$	63384	15929	**9127**	11974	10807	5249
$2^{37}-25$	7	70328	84	$[L_3, L_7, L_5, L_6, L_{23}]$	218775	**26241**	16098	29226	24153	10405
$2^{38}-45$	3	100268	86	$[L_3, L_7, L_5, L_6, L_{23}]$	217145	43595	21343	**27187**	26982	14897
$2^{39}-7$	1	174817	96	$[L_7, L_3, L_5, L_6, L_{23}]$	230235	28802	48488	**36770**	38318	19677
$2^{40}-87$	1	266662	95	$[L_7, L_3, L_5, L_6, L_{23}]$	394908	49855	**80764**	66646	56901	28016
$2^{41}-21$	3	205227	92	$[L_7, L_3, L_5, L_6, L_{23}]$	448887	52656	105639	69940	62212	**27395**
$2^{42}-11$	5	557046	99	$[L_7, L_3, L_5, L_6, L_{23}]$	720206	93920	189498	147651	**102116**	64309
$2^{43}-57$	7	198777	95	$[L_7, L_3, L_5, L_6, L_{23}]$	705224	**69021**	153095	95778	81922	44112
$2^{44}-17$	7	307870	98	$[L_7, L_3, L_5, L_6, L_{23}]$	808057	131220	285136	**145263**	142750	72964
$2^{45}-55$	1	3120225	108	$[L_7, L_3, L_5, L_6, L_{23}]$	2298828	301730	**410169**	579449	404520	226542
$2^{46}-21$	3	2759728	102	$[L_7, L_3, L_5, L_6, L_{23}]$	9075335	516826	**788898**	957832	730020	382101
$2^{47}-115$	5	4234340	108	$[L_7, L_3, L_5, L_6, L_{23}]$	5182631	650377	866413	**650377**	801837	781907
$2^{48}-59$	5	2706129	111	$[L_7, L_3, L_5, L_6, L_{23}]$	6739857	**546014**	899553	756358	651990	491312
$2^{49}-81$	7	1239417	107	$[L_7, L_3, L_5, L_6, L_{23}]$	3582205	288124	660449	**326050**	319641	252270

Table 4. The average number of nodes visited in the search for subfield j-invariants in Solver and SuperSolver. Further explanation in text.

The key trend to highlight is that, relatively speaking, SuperSolver gains more advantage over Solver as the primes get larger. This is not as evident for the small primes in Tables 4 and 5 as it is for the larger primes below.

Remark 5 ($\mathcal{X}(\mathbb{F}_p, 2)$ clusters in $\mathcal{X}(\bar{\mathbb{F}}_p, 2)$). An interesting trend to highlight in Table 4 is that the average number of nodes visited in the optimised Delfs–Galbraith walk through $\mathcal{X}(\mathbb{F}_p, 2)$ is significantly more than the expected number of elements one would need to select randomly from S_{p^2} in order to find an element of S_p. The reason for this is that components of $\mathcal{X}(\mathbb{F}_p, 2)$ *cluster together*

				Solver	SuperSolver				
Prime p	$\left\lceil \frac{\#S_{p^2}}{\#S_p} \right\rceil$	$\mathbb{F}_p$-mults. per step	Fastest L_j's $[L_{(i)} \cdots, L_{(v)}]$	L_0	$L_{(i)}$	$L_{(ii)}$	$L_{(iii)}$	$L_{(iv)}$	$L_{(v)}$
$2^{20}-3$	530	54	$[L_3, L_1, L_7, L_5, L_2]$	44848	**20601**	22585	22235	23459	24951
$2^{21}-9$	156	53	$[L_3, L_1, L_7, L_5, L_2]$	24187	**13648**	18578	15453	18770	14064
$2^{22}-3$	584	60	$[L_3, L_7, L_1, L_5, L_6]$	52385	28062	31962	**26410**	32555	38348
$2^{23}-15$	583	71	$[L_3, L_7, L_5, L_6, L_1]$	59691	**30508**	32883	39703	33370	44556
$2^{24}-3$	1277	64	$[L_3, L_7, L_5, L_1, L_6]$	112878	**53900**	62725	70482	59206	73117
$2^{25}-39$	1231	71	$[L_3, L_7, L_5, L_1, L_6]$	128703	**63021**	68210	83333	94434	70878
$2^{26}-5$	732	62	$[L_3, L_7, L_1, L_5, L_6]$	85437	59484	**58286**	65813	61261	62216
$2^{27}-39$	2348	73	$[L_3, L_7, L_5, L_6, L_1]$	251304	164036	**135672**	136633	137780	185819
$2^{28}-57$	2965	64	$[L_3, L_7, L_1, L_5, L_6]$	631157	305345	308003	**298049**	299314	351102
$2^{29}-3$	2953	74	$[L_3, L_7, L_5, L_6, L_1]$	326888	199985	**171489**	173335	177986	235902
$2^{30}-35$	3965	75	$[L_3, L_7, L_5, L_6, L_1]$	412457	260188	232753	**228089**	236360	301541
$2^{31}-1$	9009	75	$[L_3, L_7, L_5, L_6, L_1]$	1998840	809040	**807306**	889210	871068	934319
$2^{32}-5$	5142	75	$[L_3, L_7, L_5, L_6, L_1]$	758637	455571	**449889**	501335	474549	572203
$2^{33}-9$	6638	77	$[L_3, L_7, L_5, L_6, L_1]$	1564701	**700390**	733705	900515	751310	1153911
$2^{34}-41$	10526	78	$[L_3, L_7, L_5, L_6, L_1]$	2537688	1184024	1191084	1467113	**1276654**	1799292
$2^{35}-31$	117571	99	$[L_7, L_3, L_5, L_6, L_{23}]$	15272705	**5037679**	5790782	6109529	6396752	6213090
$2^{36}-5$	29040	83	$[L_3, L_7, L_5, L_6, L_{23}]$	5244914	3006618	**2913909**	2942626	3099020	3211580
$2^{37}-25$	70328	84	$[L_3, L_7, L_5, L_6, L_{23}]$	18322417	**4979176**	5155517	7211517	6950196	6375918
$2^{38}-45$	100268	86	$[L_3, L_7, L_5, L_6, L_{23}]$	18402937	8315681	6856578	**6735588**	7791309	9143526
$2^{39}-7$	174817	96	$[L_7, L_3, L_5, L_6, L_{23}]$	22505327	9627241	9879376	**9587856**	11562406	12332858
$2^{40}-87$	266662	95	$[L_7, L_3, L_5, L_6, L_{23}]$	38602102	16664021	**16455546**	17377853	17169885	17559520
$2^{41}-21$	205227	92	$[L_7, L_3, L_5, L_6, L_{23}]$	41185437	17284297	20890068	17817383	18399209	**17006068**
$2^{42}-11$	557046	99	$[L_7, L_3, L_5, L_6, L_{23}]$	70760036	31439715	38704883	38573868	**30864574**	40337712
$2^{43}-57$	198777	95	$[L_7, L_3, L_5, L_6, L_{23}]$	66820000	**22863425**	30733754	24686759	24474388	27514948
$2^{44}-17$	307870	98	$[L_7, L_3, L_5, L_6, L_{23}]$	79795521	43991657	58381667	**38022655**	43217829	45803624
$2^{45}-55$	3120225	108	$[L_7, L_3, L_5, L_6, L_{23}]$	247697962	103871110	**87674099**	156886333	126109617	144250917
$2^{46}-21$	2759728	102	$[L_7, L_3, L_5, L_6, L_{23}]$	923415913	174816651	**163893709**	198006728	205399202	220786555
$2^{47}-115$	4234340	108	$[L_7, L_3, L_5, L_6, L_{23}]$	550653552	222915969	183895536	**175113230**	248769348	272706341
$2^{48}-59$	2706129	111	$[L_7, L_3, L_5, L_6, L_{23}]$	729589278	**188237971**	192728454	205161765	251091015	310387211
$2^{49}-81$	1239417	107	$[L_7, L_3, L_5, L_6, L_{23}]$	385982057	99186957	141171527	**88278361**	99648273	160633132

Table 5. The average number of $\mathbb{F}_p$ multiplications used to search for subfield j-invariants in Solver and SuperSolver. Further explanation in text.

in $\mathcal{X}(\bar{\mathbb{F}}_p, 2)$. Thus, with respect to finding subfield nodes, walks in $\mathcal{X}(\bar{\mathbb{F}}_p, 2)$ are significantly different from selecting nodes at random from S_{p^2}. The types of clusterings in $\mathcal{X}(\bar{\mathbb{F}}_p, 2)$ depend on the value of $p \bmod 8$ [13, Theorem 2.7], which is why this value is given alongside p in each row. Write N for the ratio between the number of nodes we visited on average (i.e., the bold column) and the number of elements we would expect to draw at random from S_{p^2} before finding one in S_p (i.e., $\#S_{p^2}/\#S_p$). Table 4 shows that (i) when $p \equiv 1 \bmod 4$, we typically see $1 \leq N \leq 2$; (ii) when $p \equiv 3 \bmod 8$, we typically see $2 \leq N \leq 3$; and (iii) when $p \equiv 7 \bmod 8$, we often see $N > 3$. For more experimental data illustrating this phenomenon, see [3, §4.3]. In practice, we do not see the $N > 1$ as enough of a reason to incur the significant overhead of walking in $\mathcal{X}(\bar{\mathbb{F}}_p, \ell)$ for $\ell > 2$ instead. In any case, the method of fast subfield root detection proposed in this paper will work regardless of the ℓ-isogenies that are used to take steps in a given walk. In fact, if walking in $\mathcal{X}(\bar{\mathbb{F}}_p, \ell)$ for $\ell > 2$ results in better concrete performance than for $\ell = 2$, the greater cost of taking a step in $\mathcal{X}(\bar{\mathbb{F}}_p, \ell)$ is likely to increase the size of the set of "fast ℓ's" and the relative efficacy of invoking subfield root detection.

Large primes and optimal node coverage. Table 6 illustrates the increased efficacy of SuperSolver over Solver as the supersingular isogeny graphs get larger. Recall that we reported some of the results from this table up front in Section 1, namely from the experiments using primes from the isogeny literature. We chose the largest prime below 2^k for $k \in \{50, 100, \ldots 800\}$, and started from a pseudo-random j-invariant in $S_{p^2} \setminus S_p$ as usual. Since these instances are too large to actually run the full subfield search until it terminates, in each case we ran both Solver and SuperSolver (for the three sets of ℓ's that were predicted to perform best during preprocessing) until the number of $\mathbb{F}_p$ multiplications used exceeded 10^8, and then immediately stopped. The numbers reported in bold in Table 6 are the total number of nodes covered (i.e., both walked onto *and* inspected) during these walks. For the smallest prime $p = 2^{50} - 27$, SuperSolver covers between 3 and 4 times the number of nodes that Solver does; for the largest prime $p = 2^{800} - 105$, SuperSolver covers between 18 and 19 times the number of nodes. Though primes beyond this size are unlikely to be of cryptographic interest, it is worth pointing out that this trend continues: the larger p grows, the more profitable it becomes to keep adding ℓ's in the fast subfield inspection algorithm.

Storing and accessing the reduced modular polynomials. The unreduced modular polynomials $\Phi_\ell(X, Y) \in \mathbb{Z}[X, Y]$ require a significant amount of storage, but recall that the preprocessing phase immediately reduces all of the coefficients into $\mathbb{F}_p$ to produce $\Phi_{\ell,p}(X, Y) \in \mathbb{F}_p[X, Y]$. This can be done once-and-for-all for a specific prime, and this makes the storage and access of the $\Phi_{\ell,p}(X, Y)$ a non-issue. Storing $\Phi_{\ell,p}(X, Y)$ requires at most $(N_\ell^2/2) \cdot \log_2(p)$ bits. For example, the largest $\Phi_{\ell,p}(X, Y)$ for the 250-bit prime above is $\Phi_{13,p}(X, Y)$, which requires the storage of at most $N_{13}^2/2 = 14^2/2 = 98$ elements of $\mathbb{F}_p$, around 3KB. The largest $\Phi_{\ell,p}(X, Y)$ for the 800-bit prime above requires the storage of at most $N_{19}^2/2 = 20^2/2 = 200$ elements of $\mathbb{F}_p$, around 20KB. Any of these would comfortably fit into the L1 cache on a modern CPU.

Concrete security of the supersingular isogeny problem. Our SuperSolver suite makes it straightforward to obtain precise estimates on the concrete classical security offered by the general supersingular isogeny problem in S_{p^2}, for any prime p. Combining a small experiment (like those reported in Table 6) with the expected number of nodes one must cover before reaching a subfield node allows us to obtain accurate counts on the expected number of $\mathbb{F}_p$ multiplications, squarings and additions that must be carried out during a full cryptanalytic attack. It is then a matter of costing these $\mathbb{F}_p$ operations with respect to the appropriate metric, whether that be bit operations, cycle counts, gate counts, or circuit depth.

Take, for example, the 256-bit prime

$$p = 73743043621499797449074820543863456997944695372324032511999999999999999999999$$

underlying SQISign [15] to illustrate how our software can be used to obtain precise security estimates. The precomputation phase of SuperSolver (which takes a few seconds on input of p) reveals that taking an optimised step in $\mathcal{X}(\bar{\mathbb{F}}_p, 2)$ costs 407 multiplications in $\mathbb{F}_p$. Based on this cost, the precomputation further determines that the fastest set of ℓ's to proceed with are

$$\ell \in \{3, 5, 7, 11, 13\}.$$

On average, the combination of this set of ℓ's and Algorithm 2 reduces the cost of the subfield search from 407 multiplications in $\mathbb{F}_p$ per node to 58.0 multiplications in $\mathbb{F}_p$ per node (see Table 1). Thus, on average, solving the supersingular isogeny problem costs

$$58.0 \times \left(\frac{\#S_{p^2}}{\#S_p}\right) \ \mathbb{F}_p \text{ multiplications.}$$

Since $p \equiv 7 \bmod 12$, we have $\#S_{p^2} = \lfloor p/12 \rfloor + 1$ [27, Theorem V.4.1(c)], and since $p \equiv 7 \bmod 8$, $\#S_p$ is exactly the class number of the imaginary quadratic field $\mathbb{Q}(\sqrt{-p})$ [13, Equation 1]. We suppose this class number is N, i.e., $\#S_p = N$. Writing $N = 2^k$, where k is correct to 3 decimal places, we would obtain that the average cost of breaking this instance of SQISign is $2^{257.622-k}$ multiplications in $\mathbb{F}_p$.[10]

In Table 7 we give average counts for the cost of breaking the supersingular isogeny problem using SuperSolver for a number of primes underlying either B-SIDH or SQISign.

Acknowledgements. Thanks to Sam Frengley, Michael Naehrig, Krijn Reijnders, Benjamin Smith, Greg Zaverucha, and the CRYPTO2022 reviewers for their valuable comments on an earlier version of this paper. We also thank Drew Sutherland for answering our questions about alternative modular functions.

References

1. G. Adj, D. Cervantes-Vázquez, J. Chi-Domínguez, A. Menezes, and F. Rodríguez-Henríquez. On the cost of computing isogenies between supersingular elliptic curves. In *International Conference on Selected Areas in Cryptography*, pages 322–343. Springer, 2018.
2. G. Adj and F. Rodríguez-Henríquez. Square root computation over even extension fields. *IEEE Transactions on Computers*, 63(11):2829–2841, 2013.
3. S. Arpin, C. Camacho-Navarro, K. Lauter, J. Lim, K. Nelson, T. Scholl, and J. Sotáková. Adventures in supersingularland. *Experimental Mathematics*, pages 1–28, 2021.
4. E. R. Berlekamp. Factoring polynomials over large finite fields. *Mathematics of computation*, 24(111):713–735, 1970.
5. W. Beullens, T. Kleinjung, and F. Vercauteren. CSI-FiSh: Efficient isogeny based signatures through class group computations. In S. D. Galbraith and S. Moriai, editors, *ASIACRYPT 2019*, volume 11921 of *Lecture Notes in Computer Science*, pages 227–247. Springer, 2019.

[10]For large, cryptographic sized primes p, computing class numbers is very computationally expensive. Indeed, a recent class group computation for a 512-bit prime terminated in ≈ 52 core years.

6. J. H. Bruinier, K. Ono, and A. V. Sutherland. Class polynomials for nonholomorphic modular functions. *Journal of Number Theory*, 161:204–229, 2016.
7. J. Burdges and L. De Feo. Delay encryption. In A. Canteaut and F. Standaert, editors, *Advances in Cryptology - EUROCRYPT 2021 - 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17-21, 2021, Proceedings, Part I*, volume 12696 of *Lecture Notes in Computer Science*, pages 302–326. Springer, 2021.
8. D. G. Cantor and H. Zassenhaus. A new algorithm for factoring polynomials over finite fields. *Mathematics of Computation*, pages 587–592, 1981.
9. W. Castryck and T. Decru. CSIDH on the surface. In J. Ding and J. Tillich, editors, *Post-Quantum Cryptography - 11th International Conference, PQCrypto 2020, Paris, France, April 15-17, 2020, Proceedings*, volume 12100 of *Lecture Notes in Computer Science*, pages 111–129. Springer, 2020.
10. W. Castryck, T. Decru, and F. Vercauteren. Radical isogenies. In S. Moriai and H. Wang, editors, *Advances in Cryptology - ASIACRYPT 2020 - 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7-11, 2020, Proceedings, Part II*, volume 12492 of *Lecture Notes in Computer Science*, pages 493–519. Springer, 2020.
11. C. Costello. B-SIDH: supersingular isogeny Diffie-Hellman using twisted torsion. In *ASIACRYPT*, pages 440–463. Springer, 2020.
12. C. Costello, M. Meyer, and M. Naehrig. Sieving for twin smooth integers with solutions to the Prouhet-Tarry-Escott problem. In *EUROCRYPT*, volume 12696, pages 272–301. Springer, 2021.
13. C. Delfs and S. D. Galbraith. Computing isogenies between supersingular elliptic curves over $\mathbb{F}_p$. *Designs, Codes and Cryptography*, 78(2):425–440, 2016.
14. L. De Feo, D. Jao, and J. Plût. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. *J. Math. Cryptol.*, 8(3):209–247, 2014.
15. L. De Feo, D. Kohel, A. Leroux, C. Petit, and B. Wesolowski. SQISign: compact post-quantum signatures from quaternions and isogenies. In *ASIACRYPT*, pages 64–93. Springer, 2020.
16. S. D. Galbraith, C. Petit, and J. Silva. Identification protocols and signature schemes based on supersingular isogeny problems. *J. Cryptol.*, 33(1):130–175, 2020.
17. M. Hamburg. Fast and compact elliptic-curve cryptography. Cryptology ePrint Archive, Report 2012/309, 2012. `https://ia.cr/2012/309`.
18. R. A. Horn and C. R. Johnson. *Topics in matrix analysis.* Cambridge university press, 1994.
19. D. Jao, R. Azarderakhsh, M. Campagna, C. Costello, L. De Feo, B. Hess, A. Jalali, B. Koziel, B. LaMacchia, P. Longa, M. Naehrig, J. Renes, V. Soukharev, and D. Urbanik. SIKE: Supersingular Isogeny Key Encapsulation. Manuscript available at `sike.org/`, 2017.
20. C. Leonardi. *Security Analysis of Isogeny-Based Cryptosystems.* PhD thesis, University of Waterloo, Ontario, Canada, 2020.
21. R. Lidl and H. Niederreiter. *Introduction to finite fields and their applications.* Cambridge University Press, 1994.
22. A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of applied cryptography.* CRC press, 2018.
23. J-F. Mestre. La méthode des graphes. exemples et applications. In *Proceedings of the international conference on class numbers and fundamental units of algebraic number fields (Katata)*, pages 217–242. Citeseer, 1986.
24. A. K. Pizer. Ramanujan graphs and Hecke operators. *Bulletin of the American Mathematical Society*, 23(1):127–137, 1990.

25. M. Scott. A note on the calculation of some functions in finite fields: Tricks of the trade. *IACR Cryptol. ePrint Arch.*, page 1497, 2020.
26. V. Shoup. *A computational introduction to number theory and algebra.* Cambridge university press, 2009.
27. J. H. Silverman. *The arithmetic of elliptic curves*, volume 106. Springer, 2009.
28. A. V. Sutherland. Modular Polynomials. `https://math.mit.edu/~drew/ClassicalModPolys.html`. Online; accessed 30 September 2021.
29. A. V. Sutherland. On the evaluation of modular polynomials. *The Open Book Series*, 1(1):531–555, 2013.
30. The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 9.2)*, 2021. `https://www.sagemath.org`.
31. J. Vélu. Isogénies entre courbes elliptiques. *CR Acad. Sci. Paris Sér. AB*, 273(A238-A241):5, 1971.

	Solver		SuperSolver			
Prime p	Nodes inspected	$\mathbb{F}_p$-mults. per node	Fastest Sets	Nodes inspected	$\mathbb{F}_p$-mults. per node	**Improv. Factor**
			{3,5,7}	2,859,221	35.0	
$2^{50} - 27$	883,007	113	{3,5}	2,736,601	36.5	**2.8 - 3.2x**
			{3,7}	2,533,945	39.4	
			{3,5,7}	2,165,681	46.0	
$2^{100} - 15$	443,951	223	{3,5,7,11}	2,121,313	47.1	**4.5 - 4.9x**
			{3,5,7,9}	1,988,731	47.1	
			{3,5,7,9,11}	1,847,413	51.8	
$2^{150} - 3$	317,215	315	{3,5,7,11}	1,895,201	52.9	**5.6 - 6.0x**
			{3,5,7,9}	1,776,751	52.9	
			{3,5,7,9,11}	1,700,749	56.2	
$2^{200} - 75$	241,987	415	{3,5,7,9,11,13}	1,715,449	56.3	**7.0 - 7.1x**
			{3,5,7,11,13}	1,716,767	58.3	
			{3,5,7,9,11,13}	1,607,145	60.1	
$2^{250} - 207$	191,115	521	{3,5,7,9,11}	1,561,645	61.1	**8.2 - 8.4x**
			{3,5,7,11,13}	1,586,495	63.0	
			{3,5,7,9,11,13}	1,531,993	63.0	
$2^{300} - 153$	164,273	610	{3,5,7,9,11}	1,468,279	65.0	**8.0 - 8.4x**
			{3,5,7,9,11,13,17}	1,489,991	65.3	
			{3,5,7,9,11,13}	1,452,529	66.5	
$2^{350} - 113$	141,097	709	{3,5,7,9,11,13,17}	1,432,345	68.0	**9.7 - 10.3x**
			{3,5,7,9,11}	1,372,351	70.0	
			{3,5,7,9,11,13}	1,380,849	70.0	
$2^{400} - 593$	123,649	809	{3,5,7,9,11,13,17}	1,378,991	70.6	**10.8 - 11.2x**
			{3,5,7,9,11,13,19}	1,339,805	72.8	
			{3,5,7,11,13,9,17}	1,330,891	73.2	
$2^{450} - 501$	110,407	906	{3,5,7,11,13,9}	1,317,849	73.3	**11.9 - 12.1x**
			{3,5,7,11,13,9,17,19}	1,309,703	74.8	
			{3,5,7,9,11,13,17}	1,274,503	76.4	
$2^{500} - 863$	97,209	1032	{3,5,7,9,11,13,17,19}	1,266,275	77.3	**12.8 - 13.1x**
			{3,5,7,9,11,13}	1,245,721	77.5	
			{3,5,7,9,11,13,17}	1,239,501	78.6	
$2^{550} - 5$	90,031	1111	{3,5,7,9,11,13,17,19}	1,238,921	79.0	**13.3 - 13.8x**
			{3,5,7,9,11,13}	1,201,873	80.3	
			{3,5,7,9,11,13,17,19}	1,200,945	81.5	
$2^{600} - 95$	81,253	1230	{3,5,7,9,11,13,17}	1,191,549	81.7	**14.4 - 14.8x**
			{3,5,7,9,11,13,19}	1,166,297	83.6	
			{3,5,7,9,11,13,17,19}	1,176,411	83.3	
$2^{650} - 611$	76,207	1314	{3,5,7,9,11,13,17}	1,161,061	83.9	**14.9 - 15.4x**
			{3,5,7,9,11,13,19}	1,137,873	85.7	
			{3,5,7,9,11,13,17,19}	1,148,963	85.2	
$2^{700} - 1113$	71,037	1408	{3,5,7,9,11,13,17}	1,127,317	86.4	**15.8 - 16.2x**
			{3,5,7,9,11,13,17,19,23}	1,123,125	87.5	
			{3,5,7,9,11,13,17,19}	1,121,045	87.4	
$2^{750} - 161$	66,237	1510	{3,5,7,9,11,13,17}	1,093,351	89.0	**16.5 - 16.9x**
			{3,5,7,9,11,13,17,19,23}	1,101,767	89.3	
			{3,5,7,9,11,13,17,19}	1,095,195	89.4	
$2^{800} - 105$	62,163	1610	{3,5,7,9,11,13,17,19,23}	1,081,825	90.9	**16.2 - 17.6x**
			{3,5,7,9,11,13,17,19,15}	1,008,481	90.9	

Table 6. The number of nodes inspected per 10^8 field multiplications for the largest primes of various bitlengths. The Solver column corresponds to optimised Delfs–Galbraith walks in $\mathcal{X}(\bar{\mathbb{F}}_p, 2)$ – see Section 3. The SuperSolver columns correspond to enabling our fast subfield root detection algorithm with the three fastest sets of ℓ's (left to right) – see Section 5. Numbers in round brackets are the approximate number of $\mathbb{F}_p$ multiplications per node inspected, as computed during the precomputation phase that determines which sets of ℓ's will perform fastest.

.

Prime p	p mod 8	Average number of $\mathbb{F}_p$-mults. per node	$\#S_{p^2}$	$\#S_p$	Average cost of SuperSolver
B-SIDH-p247 [11]	7	55.6	$2^{242.559}$	2^{k_1}	$2^{248.356-k_1}$
TwinSmooth-p250 [12]	1	59.1	$2^{246.220}$	2^{k_2-1}	$2^{252.105-k_2}$
SQISign-p256 [15]	7	55.7	$2^{251.764}$	2^{k_3}	$2^{257.564-k_3}$
TwinSmooth-p384 [12]	1	63.1	$2^{379.735}$	2^{k_4-1}	$2^{385.715-k_4}$
TwinSmooth-p512 [12]	5	69.1	$2^{507.896}$	2^{k_5-1}	$2^{514.007-k_5}$

Table 7. The average number of $\mathbb{F}_p$ multiplications required to solve the supersingular isogeny problem using SuperSolver. When $p \equiv 1 \bmod 4$, we assume that $N = 2^k$ is the class number of $\mathbb{Q}(\sqrt{-4p})$, where k correct to 3 decimal places. Otherwise, it is the class number of $\mathbb{Q}(\sqrt{-p})$. As N varies for each prime, we will index N and k by the row in the table, i.e., $N_i = 2^{k_i}$ will be the class number of the i-th prime in the table. The number of $\mathbb{F}_p$ multiplications per node using SuperSolver is taken from Table 1.