

A String Theorist's Guide to Implementing MIKE

Giacomo Pope

5 October 2026

For a long time, I didn't know anything about cryptography beyond what I read about in spy books as a kid. At some point during my PhD in string theory I got interested in Capture the Flag competitions and through that, solving cryptography puzzles. As time went on, I started doing less maths with pen and paper and more using a computer and a few years ago I got totally hooked on isogeny-based cryptography research and I have had a lot of fun writing papers with proper mathematicians helping me. Generally, I have found that I have made myself most useful writing efficient code for algorithms discovered by coauthors and the story is no different for the recent paper *MIKE: A Fast and Post-Quantum NIKE*, which I had the pleasure of writing with ten seriously talented friends of mine.

MIKE is a non-interactive key exchange whose hardness assumption comes from isogeny-based cryptography. A description of MIKE was given back in 2024 by Damien Robert in [The module action for isogeny based cryptography](#) and our paper is a realisation of this description with a concrete implementation. At a very high level, what we have implemented is the same as what was originally described, but the work of our new paper was to solve several hard mathematical problems to produce an end result which was cryptographically reasonable.

The special thing about MIKE is that relative to other post-quantum NIKes we're either much more compact (80 byte public keys compared with [SWOOSH](#) which needs 220 Kb) or much faster (at 5ms means we're more than a magnitude faster than CSIDH). SWOOSH does have faster secret generation (about 2.5ms on the same machine, if you don't need public key validation), but much slower key generation (about 38ms).

p	$\log_2(p)$	e	Public Key (bytes)	Key Gen (ms)	Shared Secret (ms)
$633 \cdot 2^{308} - 1$	318	228	80	0.65	4.9
$593 \cdot 2^{474} - 1$	484	340	122	2.12	15.2
$317 \cdot 2^{628} - 1$	637	452	160	4.21	28.6
$117 \cdot 2^{374} - 1$	381	372	96	0.86	6.2
$77 \cdot 2^{566} - 1$	573	564	144	2.57	18.0
$41 \cdot 2^{758} - 1$	764	756	192	8.12	51.1

Parameter details and running times for MIKE key generation and shared secret generation. C implementation benchmarks recorded on an `AMD Ryzen 7 PRO 7840U CPU @ 3.3Ghz` with turboboost and multithreading disabled. Note that p is the prime characteristic of the finite field and e sets the degree of the 2^e -isogenies computed in dimension one (key generation) and dimension four (shared secret generation).

I think ultimately the work we have achieved went far beyond what I expected we could do. Although the paper itself is complex, with several key results being proved in lengthy appendices, the implementation itself is much easier thanks to several beautiful symmetries we were able to use in the simplification of many sub-algorithms of MIKE. After all the complications were properly understood, we were able to strip out a lot of mathematics to allow the resulting code to be relatively simple.

Generally I like to write a little at the beginning of my blog posts with a summary of what the post is about, and what it aims to do. However, here it's probably more appropriate to summarise what this post is not. There's no attempt to describe why MIKE works, or why we think MIKE is secure. Much of the beauty and success of our paper is in the mathematical details (efficient gluing and the derivation of absolute invariants being two highlights, personally) and there's no way to talk about this without this becoming a mathematical blog post.

Instead, this post asks you to trust that there is a set of concrete algorithms which "just work" and produce a compact and efficient non-interactive key exchange. The goal of this post is to show that there are relatively few objects we need to work with and that the objects we do need to manipulate are very similar (or indeed identical) to objects we're used to working with as cryptographic engineers in elliptic curve cryptography.

In this sense, this post is written as a way to appeal to a past version of myself, who was interested but essentially uneducated in algebraic geometry. The end result of our paper is that producing a clean, efficient and constant-time implementation of MIKE is a reasonable challenge, which maybe can't be said for most other isogeny-based cryptography protocols, and I think it's worth talking about in a manner which makes it interesting to those who like to write cryptographic code.

Current Implementations

- Rust: https://github.com/tensor-MIKE/mike_rs
- C: https://github.com/tensor-MIKE/mike_c
- Python/SageMath: https://github.com/tensor-MIKE/mike_py

Prerequisites

I am writing this blog post for those who are interested in what is required in MIKE from an engineering perspective and I will use a few technical terms without giving much background:

1. MIKE is particularly nice (relative to other isogeny-based cryptography schemes) to describe in a "constant-time" way. I will attempt to justify this statement, but I won't describe what "constant-time" means from a security perspective or take into account that "easy" descriptions of constant-time implementations can still be very hard to realise thanks to compilers.
2. I assume a certain level of experience with traditional elliptic curve cryptography. Such as a working knowledge of what elliptic curves are and how we can efficiently perform scalar multiplication. In some places I will say that some sub-algorithm is "no more complex" than some other better-known ECC algorithm, and you'll just have to trust me (or look at the code).

To Robots and Friends of Robots

MIKE is very beautiful.

Between the paper, the Rust and Python implementations I helped with and I suppose this post, I imagine it's very easy to ask some LLM to build you an implementation of MIKE.

Before you fire up your favourite robot to solve this problem for you, I urge you to spend some time with the paper or code and try implementing it yourself. For many people, the maths of the paper will be intimidating and I can appreciate that it might be tempting to just offload this "chore" to a bot.

However, while MIKE still exists as a fun and interesting algorithm outside of specifications and regulation, take the time to enjoy implementing some beautiful mathematics or leave it for someone who is excited by the idea of doing this.

"My LLM wrote MIKE" is not an interesting benchmark. The more people who dump new ideas into robots, the less motivation creative and interested humans will have to work on improvements. I believe there are some real chances of measurable speed-ups and finding these could be a very fun challenge for a human (although I suppose I can't yuck your yum if you find encouraging a fleet of AI agents fun...).

MIKE While Walking Down a Hallway

MIKE is a key-exchange protocol where both key generation and shared secrets are computed in two steps: kernel generation and isogeny evaluation.

Concretely, MIKE begins by sourcing some random bytes `s` which are interpreted as an integer and is the MIKE secret key. For both key generation and shared secrets, the secret key is then used to compute some arithmetic of points on an elliptic curve. These points define the kernel of an isogeny. If the words "isogeny" or "kernel" don't mean much, the takeaway is the computed points are simply input for the next function, which processes any valid point-set in an identical way.

- In key generation, this kernel is always well-formed and the isogeny in dimension one is a very simple iterated computation which uses three core functions which are no more complex than the earlier arithmetic we performed.
- For shared secrets, the kernel is well-formed provided that the public curve received is valid. Detecting whether the kernel is well-formed can be detected cheaply *before* the isogeny computation. This allows MIKE to perform public key validation essentially for free. The dimension four isogeny is more complex than in dimension one, but requires the same kind of simple arithmetic, just more of it!

For key generation, the output of this isogeny computation is another elliptic curve, which is the MIKE public key.

For secret generation, the isogeny computation produces a dimension four abelian variety, which maybe sounds more scary, but is some object defined by coefficients in some finite field, just like an elliptic curve that we're used to. Finally, Alice and Bob will only agree on this isogeny output up to some isomorphism, so to get a valid shared secret, they have to compute some invariants of these objects which can be done efficiently with only a few multiplications and squares.

If you prefer pseudocode you have something like:

```
def keygen():
    # A single call to randomness to generate an integer
    sA = generate_secret()
    #  $P_0 + [sA]Q_0$ , with  $P_0, Q_0$  deterministically computed from  $E_0$ 
    kernel_data = dim_one_kernel( $E_0$ , sA)
    #  $\phi: E_0 \rightarrow EA = E_0 / \langle P_0 + [sA]Q_0 \rangle$  of degree  $2^e$ 
    EA = dim_one_isogeny( $E_0$ , kernel_data)
    return (EA, sA)

def shared_secret(sA, EB):
    #  $\{P, Q, [sA]P, [sA]Q\}$  with  $P$  and  $Q$  deterministically computed from  $EB$ 
    kernel_data = dim_four_kernel(EB, sA)
    if kernel_is_bad(kernel_data):
        raise Error("invalid public key")
    #  $XAB$  is an abelian four-fold, the codomain of the  $2^e$ -isogeny from  $EB$ 
    XAB = dim_four_isogeny(EB, kernel_data)
    # Alice and Bob only agree up to isomorphism of  $XAB$  and  $XBA$ , so we need
    # to compute data which is identical for all isomorphic four-folds.
    jAB = invariants(XAB)
    return jAB
```

Isogeny-Based Comparisons

The above summary probably raises more questions than it answers but the core component which I am trying to communicate is how cleanly we can separate "secret dependent" computations, entirely within scalar multiplications which are very well understood, and the more complex but fixed isogeny algorithms which produce some output given any valid kernel in a fixed way. This is a dream from an engineering perspective, especially compared to what we're used to in isogeny based cryptography...

Comparison to SQIsign: for those who know SQIsign, the "hardest" part of a cryptographic implementation is in the implementation of the quaternion algebra and how it integrates into signing. This requires constant-time big integer arithmetic, which is hard to make both efficient and safe. For SQIsign, the secret key is fed into these quaternion computations and because of this, it is very hard to write a constant-time signing algorithm or even track how the secret key could be leaked during the computation.

Comparison to CSIDH: for those who know CSIDH, the "hardest" part of a cryptographic implementation is in computing the variable-degree isogeny in constant time in a safe and efficient way. Instead of a CSIDH secret being a scalar, the secret is the isogeny degree itself, forcing complex secret dependent computations within the much more expensive and complex isogeny formula.

This is not to say a constant-time implementation of SQIsign or CSIDH is impossible, or prohibitively expensive, but MIKE arrives "constant time" out of the box, meaning we can put forward a simple description of how it works which is also naturally constant time.

The Building Blocks of MIKE

Before discussing some more of the technical parts of a MIKE implementation, I wanted to try and argue that the objects/types required within MIKE are pretty simple, so here's a little summary. This is not comprehensive (some details are glossed over), but any small omissions are similar in shape to those described here.

Finite Fields

The foundations of MIKE are in the implementations of finite fields. In particular we require working with two different fields at different points of the key exchange.

The Base Field

For the majority of the shared secret generation, operations are in the field $\mathbb{F}_p$. For some prime p , all arithmetic ($+$, $-$, $\times$, $/$) is performed modulo the prime p . Up to the choice of the prime, this is exactly what we're used to working with in elliptic curve cryptography, whose coordinates are in some finite field.

In a certain sense, working on efficient and constant-time $\mathbb{F}_p$ arithmetic is the hardest part of MIKE. Once the arithmetic and a few helper functions (such as constant-time conditional swapping) are implemented, almost all the hard work has been done (from an engineering perspective at least).

The Extension Field

For key generation, and the beginning of the shared secret generation, we have to work in a degree-two extension of $\mathbb{F}_p$ which we denote $\mathbb{F}_{p^2}$. At the level of the code, an element of $\mathbb{F}_{p^2}$ is represented by two elements x_0 and x_1 of $\mathbb{F}_p$ such that $x = x_0 + ix_1$. In our case, where $p \equiv 3 \pmod{4}$, we use the generator i such that $i^2 = -1$ and $\mathbb{F}_{p^2} = \mathbb{F}_p[i]/\langle i^2 + 1 \rangle$.

Working with these fields is less common in cryptography (although extension fields are used in pairing-based cryptography), but it is a very familiar field within isogeny-based research (it is for example the field required for SIKE [RIP] and SQIsign).

For the Rust implementation, the finite field arithmetic was implemented by importing my [fp2](#) crate, which is a reasonably fast and constant time implementation which I carry around with me through various projects.

Projective Points

The core objects of MIKE are projective points. Generally, a projective point is a tuple of n finite field elements ($X_1 : \dots : X_n$). In elliptic curve cryptography we are used to these points as projective points ($X : Y : Z$) on the elliptic curve.

Aside from a few tricks and steps within something called the "gluing isogeny" there are only three kinds of projective points we really need:

1. Points of the form $x(P) = (X : Z)$, which are points on the Kummer line $E/\langle -1 \rangle$ of the Montgomery curve. These are usually known as "x-only" points, and are used all the time in elliptic curve cryptography (for example when computing the scalar multiple of a point in X25519 key exchange).
2. Full projective points $P = (X : Y : Z)$, which are the points one is first shown when working with elliptic

curves. Note that as we have $-P = (X : -Y : Z)$, then on the Kummer line we have no way of discerning between P and $-P$ (as we have forgotten the Y -coordinate).

3. Theta points $P = (x_0 : \dots : x_{15})$. For MIKE, we're interested in level-two theta points in dimension four which means the point is represented by sixteen coordinates. These are more like a generalisation of the x -only points than the full projective ones.

"Theta points" may sound scary due to being unfamiliar, but from the perspective of writing code, they're just another collection of coefficients for which we perform coordinate-wise manipulations.

Elliptic Curves

For all cases in MIKE we work with elliptic curves in the Montgomery form $E : y^2 = x^3 + Ax^2 + x$ over $\mathbb{F}_{p^2}$. This means we can represent the curve with a single finite field element A (although for optimisations we often also keep track of $A_{24} = (A + 2)/4$). The use of the curve (rather than the point alone) appears in the code when we want to perform arithmetic. For example, computing $[2]P = P + P$ requires knowledge of both $x(P)$ as well as A .

Note that this curve type is suitable for working with both the projective and x -only curve points.

Theta Structures

Just as elliptic curves are the parent of the projective point, theta structures are the parent of the theta points. In dimension one, a level-two theta structure can be understood as an alternative model to the more familiar Kummer line of the Montgomery model. The theta structure can be represented by a single theta point, known as the null theta point, which is the image of the identity element.

Summary of Types

All together, you can get through all of MIKE using the following objects:

- `fp` an element of the base field $\mathbb{F}_p$
- `fp2: (fp, fp)` an element of the extension field $\mathbb{F}_{p^2}$
- `point_x: (fp2, fp2)` the x -only point $x(P)$ on the Kummer line of an elliptic curve
- `point: (fp2, fp2, fp2)` the full projective point P on an elliptic curve
- `curve: fp2` an elliptic curve in Montgomery form represented by the coefficient A .
- `theta: (fp; 16)` a level-two theta point in dimension four, which has sixteen coefficients in $\mathbb{F}_p$.
- `theta_structure: theta` a level-two theta structure is simply represented by the null point (a theta point).

Theta Arithmetic is Beautiful

Another detour! Before talking more about MIKE, I want to quickly discuss what arithmetic with theta points actually looks like, which I think is a good example of how things ultimately are very nice to work with even if everything feels unfamiliar.

Almost everything we do with a level-two theta point in dimension four is built from three operations on its sixteen coordinates. The first is the Hadamard transform, which is the map `hadamard: (a, b) -> (a + b, a - b)` applied across pairs of coordinates in four layers, and costs 64 additions and subtractions. The other two functions are even simpler: `multiply(P, Q)` is the coordinate-wise multiplication of all 16 coordinates between two points and `square(P)` is the special case `multiply(P, P)`.

With only these three functions, and given the precomputation of two specific points `C1` and `C2`, doubling a theta point is then given by the following pseudo-code:

```
def double(P: theta, C1: theta, C2: theta) -> theta:
    Q = square(P)
    Q = hadamard(Q)
    Q = square(Q)
    Q = multiply(Q, C1)
    Q = hadamard(Q)
    Q = multiply(Q, C2)
    return Q
```

The points `C1` and `C2` depend only on the theta structure (derived from the theta null point), so they're computed once and reused for every doubling. All together, doubling in dimension four costs 32 squares, 32 multiplications and 128 additions in the base finite field. Additionally, as `square` and `multiply` act on each coordinate independently and the Hadamard transform is a fixed pattern of additions, theta arithmetic is a natural candidate for optimised vectorised implementations.

For those who are used to implementing `xDBL` on elliptic curves, starting with a Hadamard followed by squaring should feel familiar, and rightly so. The level-two theta model in dimension one is related to the Montgomery model by a linear change of variables.

This simplicity extends to the isogeny computations themselves. The codomain (slightly abstracted for brevity) is computed from two points K_1 and K_2 which are points of order 8 above the kernel ($\ker(f) = \langle [4]K_1, [4]K_2 \rangle$).

```
def codomain(K1: theta, K2: theta) -> (theta_structure, theta):
    P = square(K1)
    P = hadamard(P)

    Q = square(K2)
    Q = hadamard(Q)

    # Cost: 25 multiplications
    C_img = inverse_dual_null_point(P, Q)

    # Cost: 27 multiplications + 34 additions
    codomain = ThetaStructure.from_inverse_dual(C_img)

    return (codomain, C_img)
```

and given the data from this codomain computation, an image of any point through an isogeny is also very simple:

```
def evaluate(P: theta, C_img: theta) -> theta:
    Q = square(P)
    Q = hadamard(Q)
    Q = multiply(Q, C_img)
    Q = hadamard(Q)
    return Q
```

With these three functions, the core of an isogeny chain in dimension four is complete. A working, but very slow implementation is as simple as:

```
def isogeny_chain(domain: theta_structure, K1: theta, K2: theta, e: int) -> theta_structure:
    A = domain
    for i in range(e):
        P, Q = K1, K2
        C1, C2 = arithmetic_precomputation(A)
        for _ in range(e - i - 1):
            P = double(P, C1, C2)
            Q = double(Q, C1, C2)
        A, C_img = codomain(P, Q)
        K1 = evaluate(K1, C_img)
        K2 = evaluate(K2, C_img)
    return A
```

Performance note: This would be wildly inefficient (due to the e^2 doublings computed) and the right thing to do is to use a balanced strategy to push intermediate multiples of K_1 and K_2 through each step. This results in only a quasi-linear number of evaluations and doubles, instead of the quadratic number of doubles shown here. This is discussed a little more in the following "Memory consideration" section below.

MIKE Key Generation and Shared Secret

The idea of this final section is to use a little more maths notation and proper function names to expand on the hallway section, but in reality there's still a lot of information missing. The main purpose of this is to make concrete some of the very vague abstractions earlier in the post, but for more detail, I think my recommendation would be either the paper (for maths) or our Python/Rust implementations (for implementation.)

For a given security level, a MIKE parameter set is a prime $p = c \cdot 2^f - 1$, an integer $e \leq f - 2$, a starting curve E_0 and two points P_0 and Q_0 .

Nerdy comment: These two points generate what is known as a torsion subgroup (for MIKE, this is $E_0[2^e] = \mathbb{Z}/2^e \times \mathbb{Z}/2^e = \langle P_0, Q_0 \rangle$) and for efficiency reasons, instead of precomputing P_0 and Q_0 as full projective points, we store $x(P_0)$, $x(Q_0)$ and $x(P_0 - Q_0)$ which requires three `fp2` elements.

Even nerdier comment: we actually compute $E_0[2^{e+2}] = \mathbb{Z}/2^{e+2} \times \mathbb{Z}/2^{e+2}$ to have additional torsion data above the kernel and compute K such that $[4]K = \ker \phi$ to speed up some parts of how the public key is normalised. Now, this normalisation is totally missing from this post, but the details of this are explicit in the paper. So, for the rest of the post assume P_0 and Q_0 have order 2^e ...

Key Generation

Key generation starts by sampling a secret key, which is an integer s in the range $[0, 2^e)$, but because this is a "big number" we instead sample a bunch of bytes and mask bits to ensure the resulting integer is smaller than 2^e . Also, an implementation detail of how we compute the gluing isogeny is that this secret s must be equal to 3 (mod 8), so the bottom three bits of s are known to be `0b011`. This doesn't require rejection sampling as we can simply clamp these bits after generating the integer.

Next, we take the starting curve E_0 together with the `point_x` points $x(P_0)$, $x(Q_0)$ and $x(P_0 - Q_0)$ and compute $K = P_0 + [s]Q_0$ using `three_point_ladder`. This function requires only `x_double_and_add` and conditional swaps of finite field elements, something we have a good understanding of from elliptic curve cryptography.

Once we have the kernel, we compute the 2^e -isogeny itself which requires three simple sub-algorithms `x_double`, `four_isogeny_codomain` and `four_isogeny_eval`. We use the efficient four-isogeny formula so the chain has length $e/2$, which is why our parameters pick even e (odd e can be made to work by using a single two-isogeny at the end of the chain, but we don't need to do this, so we don't). This takes as input the kernel K which is considered secret, but requires no special case handling or branching and the codomain curve which is the output is the MIKE public key.

It's worth stating here that key generation requires computing the same isogeny we saw in SIDH/SIKE (for Alice) and so we can lean on previous literature for all forms of optimisations and clever tricks to make this as fast as possible. We are **not vulnerable** to the SIDH attacks though as we output only the codomain of the isogeny and not the images of points through the secret isogeny.

Memory consideration

One implementation consideration here is how to efficiently compute the isogeny chain. Typically, the cost of a chain of length e is made quasi-linear by storing $O(\log(e))$ points as intermediates through the chain, which increases the number of `four_isogeny_eval` calls while reducing the number of `x_double` calls. For systems where memory constraints take priority, the strategy can be changed to require less memory but more time. The extreme of this is keeping track of only the kernel point and no intermediates, at the cost of $O(e^2)$ calls to `x_double`.

Shared Secret

To compute the shared secret, the first step is to take Bob's public key E_B and compute a deterministic set of points P_B, Q_B . However, unlike key-generation we require the full projective $(X : Y : Z)$ representation of these points.

Nerdy comment (again): these points are again a torsion basis $E_B[2^{e+2}] = \mathbb{Z}/2^{e+2} \times \mathbb{Z}/2^{e+2} = \langle P_B, Q_B \rangle$.

Once we have P_B, Q_B we then compute scalar multiples $[s_A]P_B, [s_A]Q_B$ of these points, and then compute a fairly complex set of points which are used as the kernel of the isogeny chain. Computationally, this is not so expensive, it requires a few elliptic curve point additions and many doublings to get points of order 8, but it is the requirement of these additions which forces us to use full projective points.

The scalar multiplication of these points can be made cheaper by using x-only arithmetic to compute $x([n]P)$ and $x([n+1]P)$ which is enough data to recover $[n]P$ without square-roots thanks to the formula of Okeya and Sakurai.

Once the kernel has been computed the main work is to compute a 2^e isogeny. This is done in two steps. Firstly a gluing isogeny is responsible for mapping one-dimensional data (projective points on E_B) to theta points in dimension four (via an intermediate step in dimension two). In the process of gluing, we move from elliptic curve points with coefficients in $\mathbb{F}_{p^2}$ to theta points with coefficients in $\mathbb{F}_p$. Moving to the base field saves about a 2-3x cost for arithmetic, which somewhat makes up for the fact we move from points with 3 to 16 coordinates!

Nerdy Gluing Comment: gluing is by far the most complex part of MIKE and is also the hardest part to implement. It is not, however, the bottleneck of the algorithm. A full description does not fit in the margins of this post, but here is a little more detail. Gluing begins with some point P on E_B , but the actual data we need are points $(P, -\sigma(P))$ on $E_B \times E_B^\sigma$ where E_B^σ is the Frobenius conjugate $E_B^\sigma : y^2 = x^3 + A^p x^2 + x$. Luckily, for any $A = a_0 + ia_1$ in $\mathbb{F}_{p^2}$, $A^p = a_0 - ia_1$ so this expensive looking exponentiation is simply a conjugation (one negation in $\mathbb{F}_p$). The action of σ on P is similarly just complex conjugation, so in the code we never represent E_B^σ and just take $P = (X : Y : Z)$ as coefficients and work with $(X^p : -Y^p : Z^p)$ when we need them. Knowing we're working with coefficients and their conjugates is maybe a good hint as to why we end up with coefficients in $\mathbb{F}_p$ after gluing. Anyway! After you take these points on this special product, you end up on a generic dimension two surface and after a little more work you glue again to end up with something in dimension four. The whole process concretely takes 3 distinct steps and when implementing MIKE you'll end up with three distinct gluing isogenies before ending up doing the rest of the chain with the much cleaner and beautiful functions showcased above.

At the end of the chain, we have a theta structure, represented by its theta null point. We take the sixteen coordinates (although we actually only need 6 of them) and use these to compute absolute invariants of the codomain X_{AB} .

More nerdy comments: The existence (and efficient computation) of these invariants is absolutely miraculous and due to symmetries of the moduli space of the codomain of this isogeny chain. Without these symmetries, computing an invariant for Alice and Bob to use as a secret would cost more than the isogeny chain itself!!

Future Work

1. MIKE is new, and there's a lot going on. The most important thing to start doing is pen-and-paper cryptanalysis of the protocol.
2. Wesolowski recently had a breakthrough attack on the [supersingular isogeny problem in cube root time](#). We made our best efforts to propose meaningful parameters for NIST levels I, III and V, but I expect further research could change parameters in the future (either making things smaller by better understanding $o(1)$ costs, or making things larger for wider margins).
3. There's some beautiful symmetries in MIKE and I believe there's potential for faster secret sharing computation if the dimension four chain can be sped up using them.