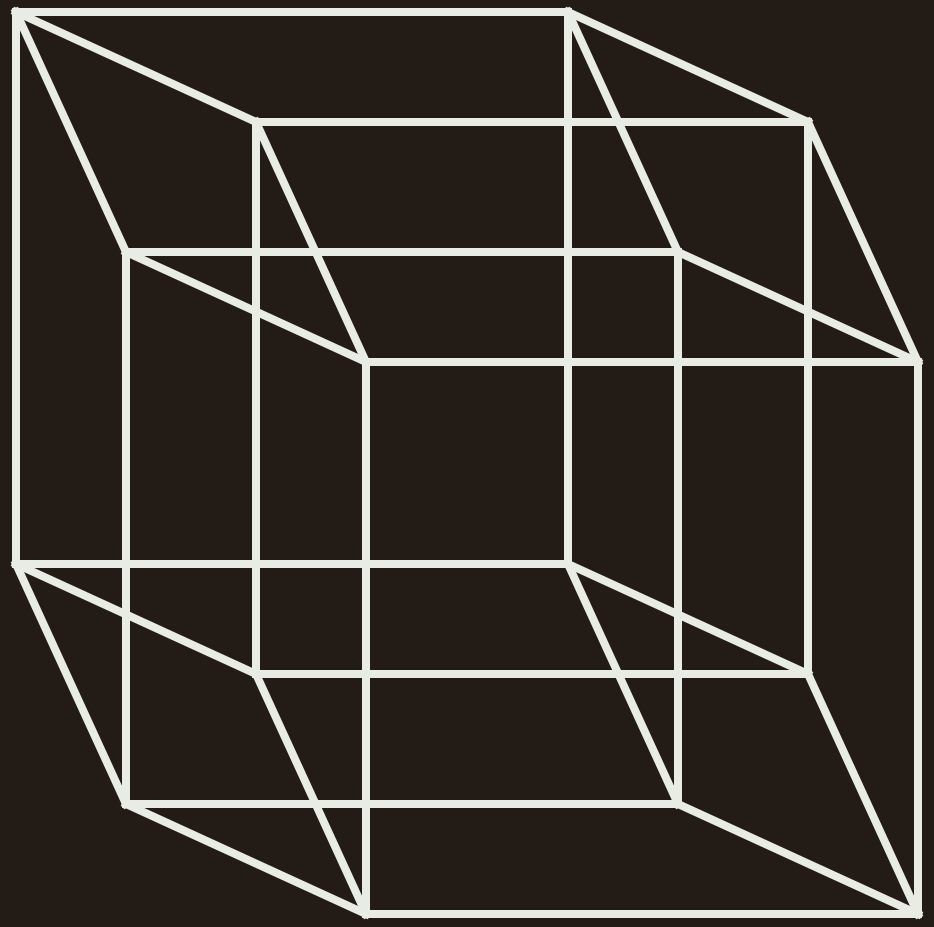
*Bernoulli Isogenies, November 16th 2026*

# A Constructivist Perspective *on* MIKE

## Implementing the Module Isogeny Key Exchange



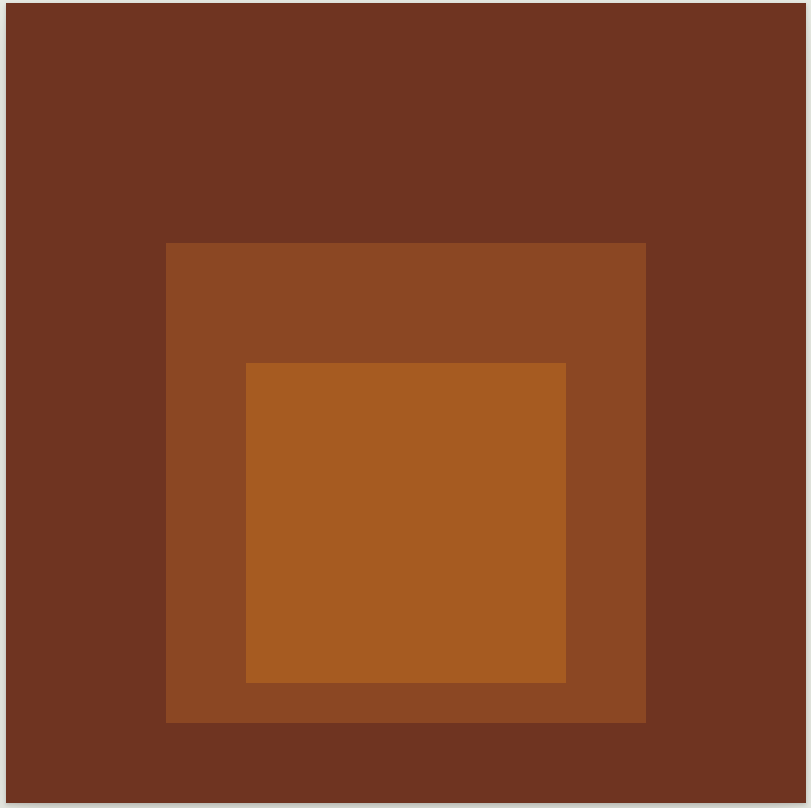
*Giacomo Pope*

*NCC Group · University of Bristol*

ncc group

# What's the Plan?

- My focus today is on **how** MIKE works, not *why* it works
- I hope that this perspective makes MIKE accessible to beginners
- Feel free to interrupt with questions
- Deeper understanding: ask *Krijn* and *Michael* for their beginners' guide



*Their beginners' guide is only 47 pages!*

# Meet the MIKE Team

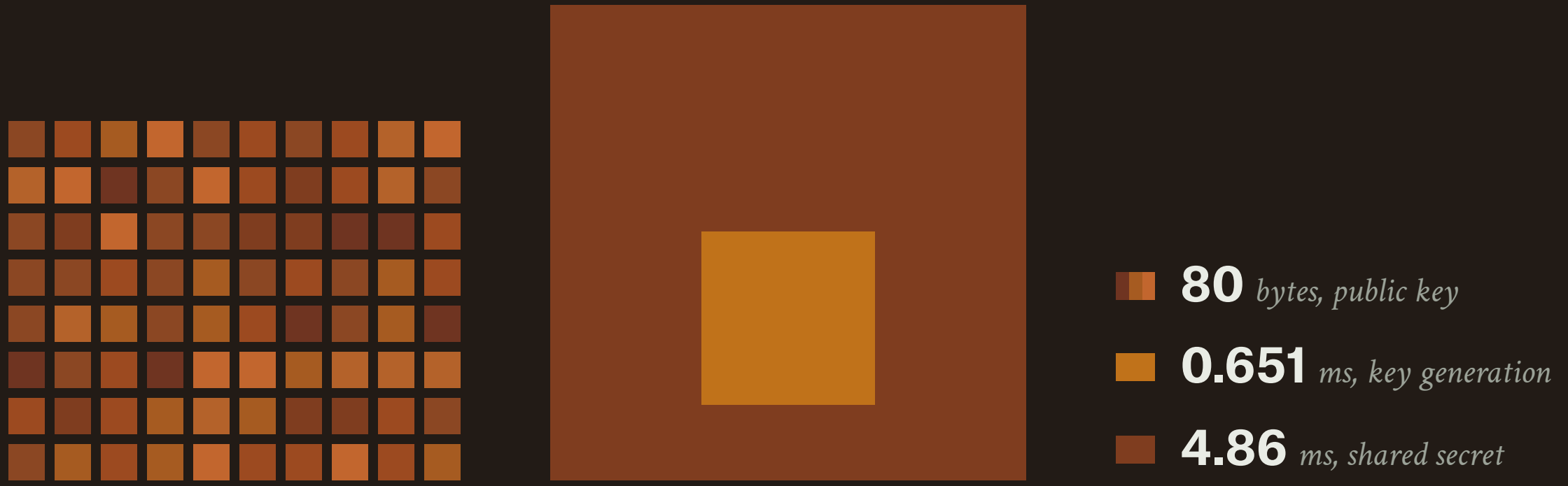
*Andrea Basso, Pierrick Dartois, Max Duparc, Jonathan Komada Eriksen, Sabrina Kunzweiler, Michael Meyer, GP, Krijn Reijnders, Damien Robert, Ryan Rueger, and Sina Schaeffler*

# MIKE Down the Hallway

- MIKE is a **NIKE**
- The public key is a **curve**
- The secret key is a **scalar**
- The whole thing hinges on a commutative module action

$$M_B * (M_A * E) = M_A * (M_B * E)$$

# MIKE is Compact *and* Fast



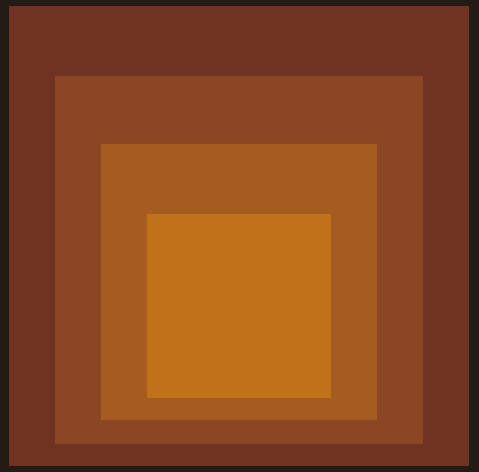
**80** *bytes, public key*

**0.651** *ms, key generation*

**4.86** *ms, shared secret*

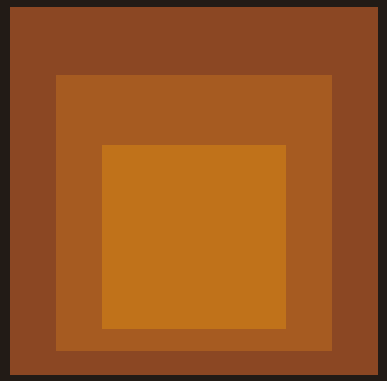
*NIST level I. Constant-time C with assembly on an AMD Ryzen 7 PRO 7840U at 3.3 GHz.*

# Less Structure, Infinite Categories...



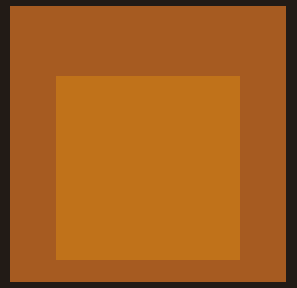
**DH**

*Subexponential; Classical*



**ECDH**

*Polynomial; Quantum*



**CSIDH**

*Subexponential; Quantum*



**MIKE**

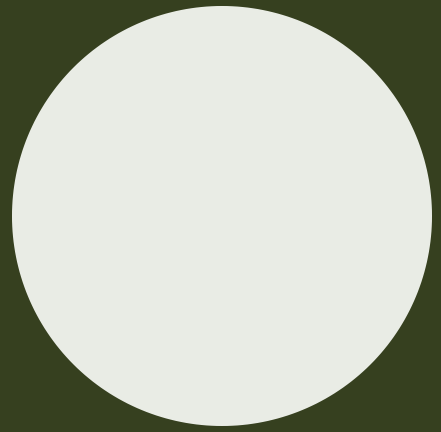
*Exponential; So far...*

# It's a Module Action, *What Could it Cost?*

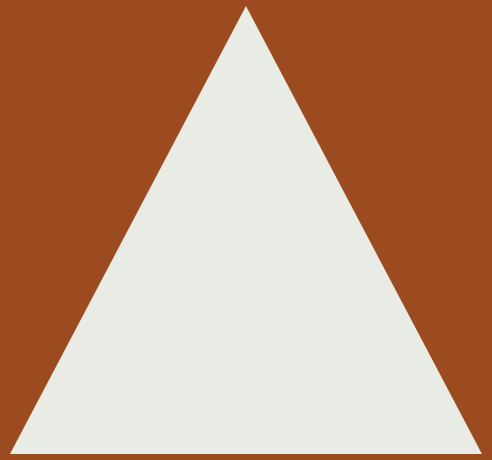
$$M_A * (M_B * E_0)$$

- **Key generation.** Bob computes $M_B * E_0$, and publishes it as a curve $E_B$
- **Shared secret.** Alice computes $M_A * (E_B \times E_B^{\sigma})$ and lands on $X_4$

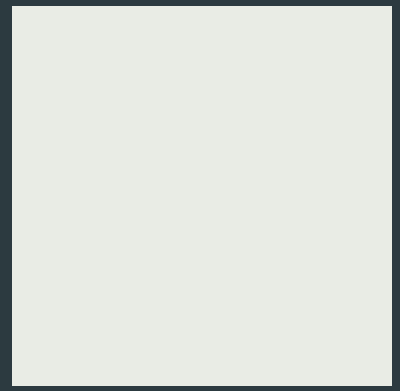
# MIKE *in* Three Steps



**Kernel**



**Isogeny**



**Invariant**

# MIKE *in* Three Steps

| | *Key Generation* | *Shared Secret* |
|---|---|---|
| **Kernel** | $K = P_0 + [x]Q_0$ on $E_0$ | $T_3, T_4$ from $x$ on $E_B$ |
| **Isogeny** | A chain of curves over $\mathbb{F}_{p^2}$ | Glue, then fourfolds over $\mathbb{F}_p$ |
| **Invariant** | A normalised $A$ | Absolute invariants $J_k$ |

# "MIKE *is* Compact"

```
def keygen(self):
    sk = self._secret_key()
    K = three_point_ladder(self.P0, self.Q0, self.PQ0, sk)
    T = dim_one_isogeny_chain(self.E0, K, self.e)
    pk = self.curve_from_four_torsion(T)
    return pk, sk


def shared_secret(self, x, EB):
    glue, kernel = self.generate_kernel_data(EB, x)
    X, kernel = gluing_isogeny(EB, glue, kernel)
    X = dim_four_isogeny_chain(X, kernel, self.e - 3)
    return self._hash(X.absolute_invariants())
```

> *Although ideals and quaternions play a central role in the theoretical discussion of MIKE, they are **completely absent** in the practical implementation.*

— *MIKE paper, §6*

# A Finite Foundation

| | | | | |
|---|---|---|---|---|
| 278F FFFF FFFF FFFF | FFFF FFFF FFFF FFFF | FFFF FFFF FFFF FFFF | FFFF FFFF FFFF FFFF | FFFF FFFF FFFF FFFF |

*$p = 633 \cdot 2^{308} - 1$ in five 64-bit limbs, most significant first*

- $p + 1 = c \cdot 2^f$: **Montgomery-friendly** reduction
- $p \equiv 3 \bmod 4$, $\mathbb{F}_{p^2} = \mathbb{F}_p(i)$ with $i^2 = -1$
- Key generation lives in $\mathbb{F}_{p^2}$; after gluing, everything is $\mathbb{F}_p$

# Generating Arithmetic

## Rust

- The `fp2` crate: generated from a macro
- Constant time, portable, no assembly

```
fp2::define_fp_core!(
    typename = MikeIFp,
    modulus = MIKE_I_MODULUS,
);
```

## C

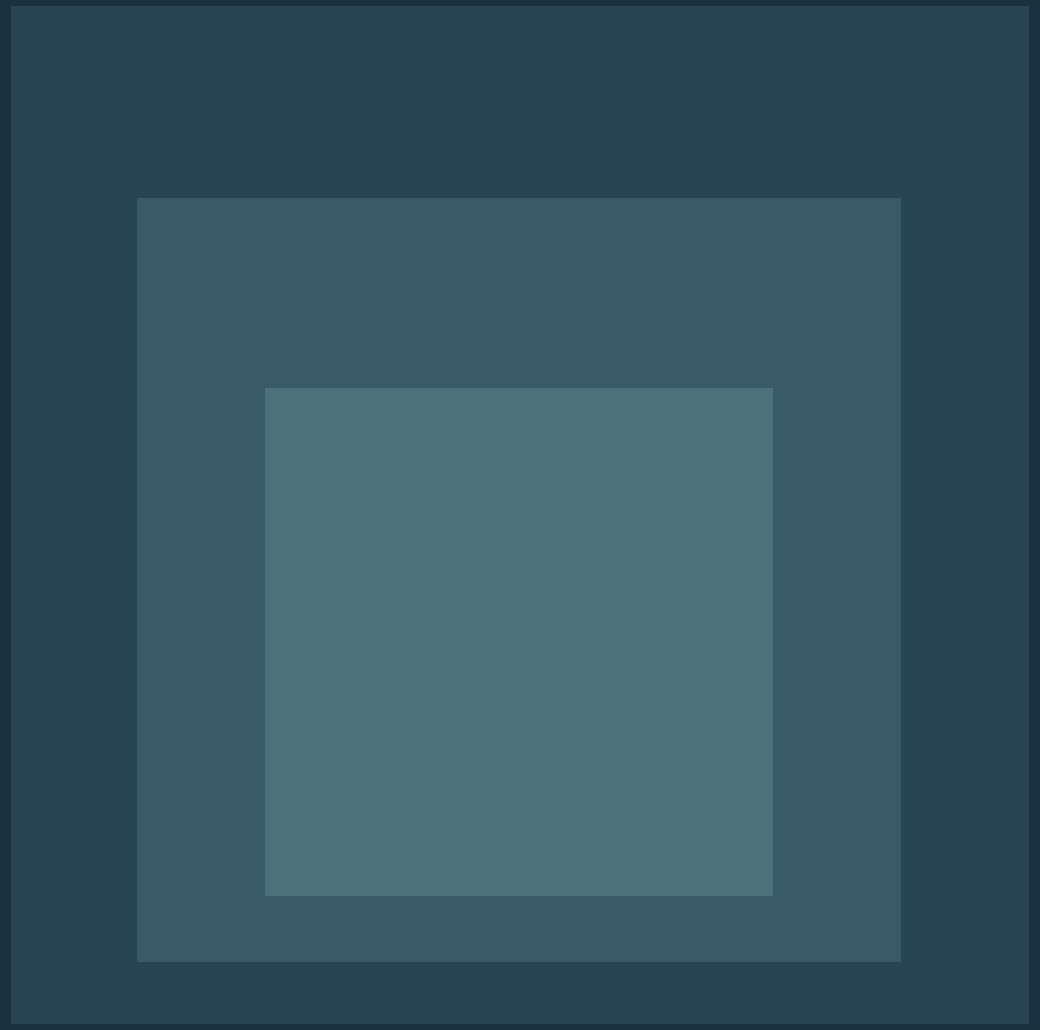
- Mike Scott's generated modular arithmetic
- Hand / LLM-written **assembly** generator for x86-64 *and* ARM64

## Ongoing

- **AVX-512**: more than **2×** faster than the fastest C assembly

# Key Generation

$(M_B * E_0)$: *SIDH sans torsion.*



# Key Generation

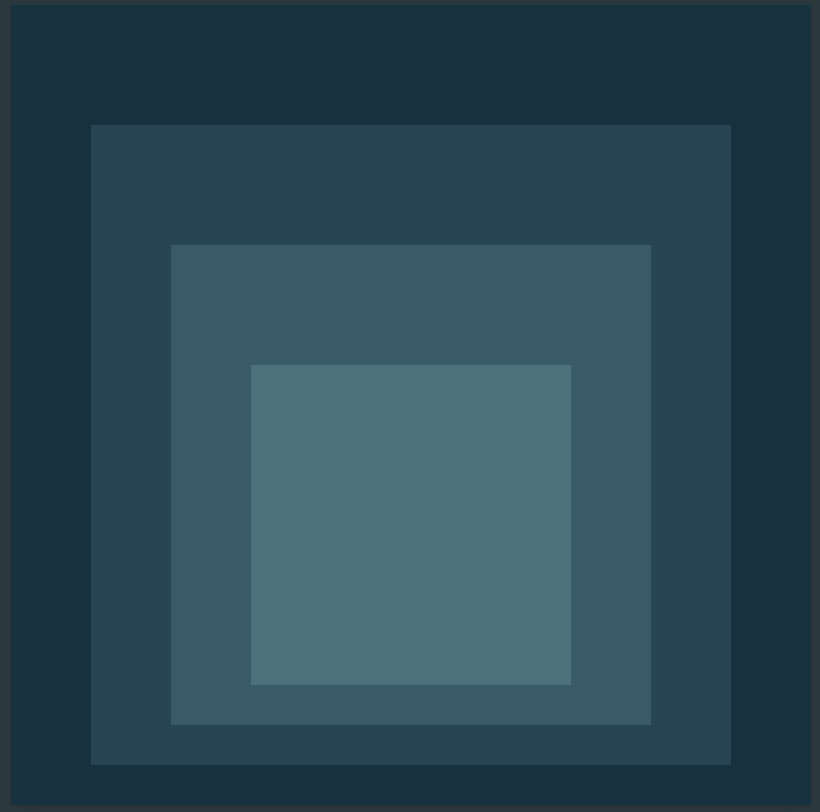
01 Sample $x \in \mathbb{Z}/2^e$ with $x \equiv 3 \bmod 8$

02 **Kernel:** $K = P_0 + [x]Q_0$ by a three-point ladder

03 **Isogeny:** $\phi : E_0 \to E_A = E_0/\langle [4]K \rangle$

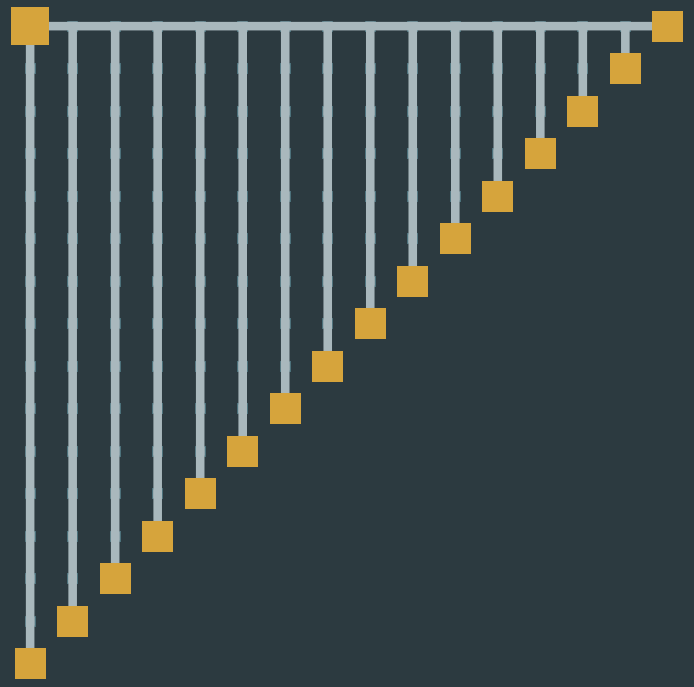
04 **Invariant:** a normalised Montgomery $A$

*SIKE key generation, without pushing any torsion points through.*

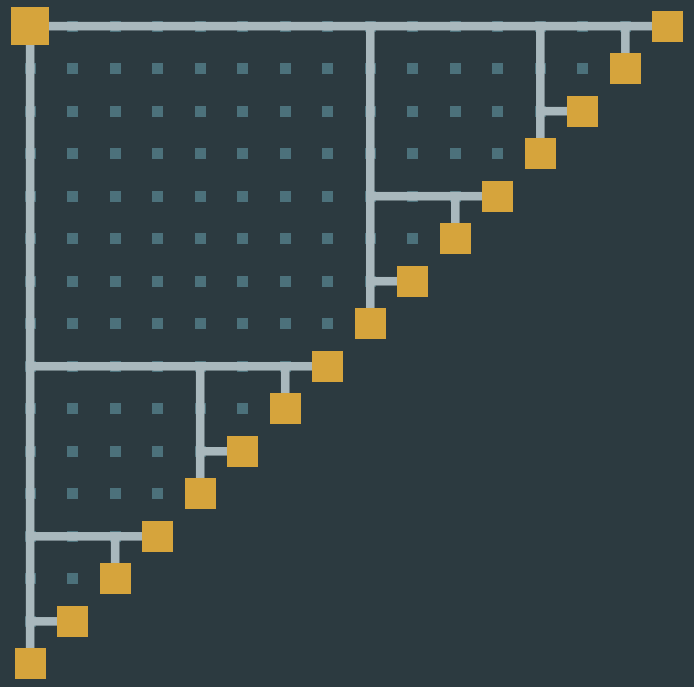


# All those SIKE optimisation papers are still useful!

# A Chain *of* Small Steps



**Naive.** *Double everything every step: $O(e^2)$*



**Balanced.** *Double less, push more: $O(e \log e)$*

# The Cost *of a* Step

```
def four_isogeny_codomain(X, Z):
    c1 = X - Z
    c2 = X + Z

    t = 2 * Z**2
    C24 = t**2
    c0 = 2 * t

    A24 = (2 * X**2) ** 2

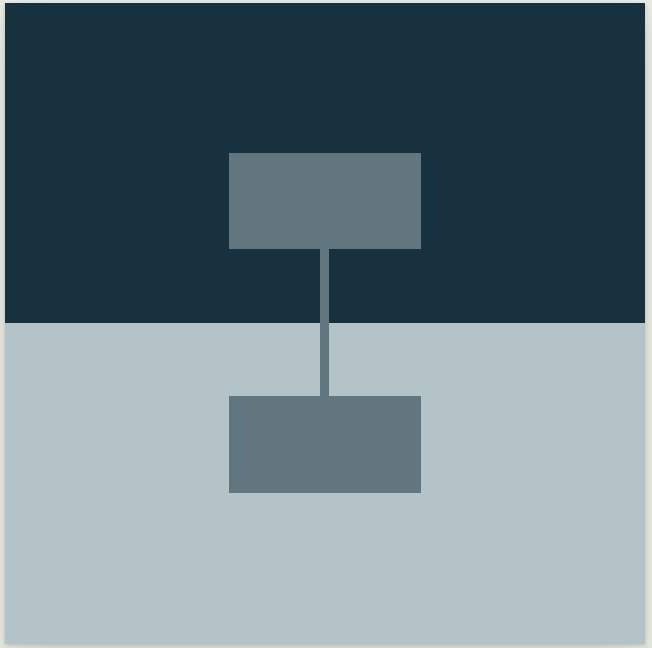
    return A24, C24, c0, c1, c2
```

| *Operation* | *Cost in* $\mathbb{F}_{p^2}$ |
|---|---|
| xDBL | $4M + 2S + 4a$ |
| 4-codomain | $4S + 5a$ |
| 4-image | $6M + 2S + 6a$ |

*Two xDBLs per four-isogeny step.*

# Colliding Keys

- The public key represents $E_A \times E_A^\sigma$, the **Weil restriction**
- $E_A$ has **six** isomorphic Montgomery models
- $E_A^\sigma$ has **six more**
- We pick exactly one of the 12
- **Different** secrets can give the **same** key



# Colliding Keys

| $E_A$ | $E_A{}^{\sigma}$ |
| --- | --- |
| $x$ | $-x$ |
| $-x^{-1}$ | $x^{-1}$ |

- Kernel $P_0 + [x]Q_0$, with $Q_0 = [i]P_0$
- **Frobenius:** $x \mapsto -x$, onto $E_A^{\sigma}$
- **The map $[i]$:** $x \mapsto -x^{-1}$, onto $E_A$
- Valid secrets pair up: $x$ and $x^{-1}$

# Deterministically MIKE

| | | | | | | |
|---|---|---|---|---|---|---|
| $E_A$ | $A_0$ | $A_1$ | $A_2$ | $-A_0$ | $-A_1$ | $-A_2$ |
| $E_A{}^{\sigma}$ | $\bar{A}_0$ | $\bar{A}_1$ | $\bar{A}_2$ | $-\bar{A}_0$ | $-\bar{A}_1$ | $-\bar{A}_2$ |

- Encode each as bytes, keep the **lexicographically largest**
- Constant time: byte comparisons and conditional swaps

# Two Extra Bits



*e bits: the kernel ⟨[4]K⟩* *2 bits: free*

- Given the Montgomery coefficient $A$, normalisation requires a square root
- We can avoid this by computing a point of four-torsion on the codomain
- Picking $K$ of order $2^{e+2}$, $\phi(K)$ has order **four** on $E_A$
- This four-torsion point gives every $A$ with **no square roots**

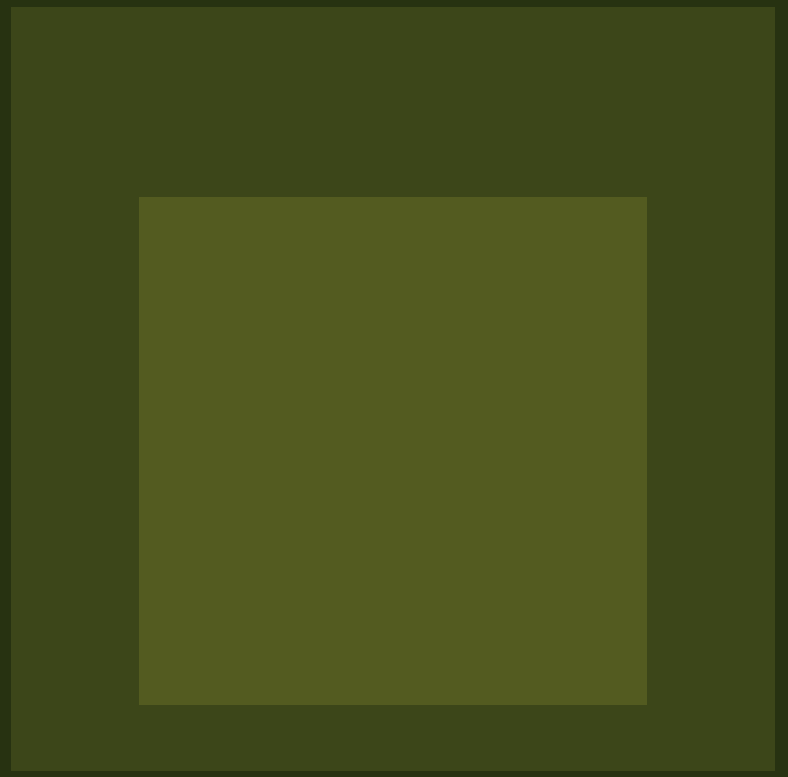
# Normalised Montgomery

```
def normalise(a0, b0):
    a1, b1 = a0 + b0, a0 - b0
    a2, b2 = i * a1 + b1, a1 + i * b1

    A0 = -2 * (a0**4 + b0**4) / (a0**4 - b0**4)
    A1 = -2 * (a1**4 + b1**4) / (a1**4 - b1**4)
    A2 = -2 * (a2**4 + b2**4) / (a2**4 - b2**4)
    As = [A0, A1, A2, -A0, -A1, -A2]
    As += [A.conjugate() for A in As]
    return max(As, key=lambda A: (A[1], A[0]))
```

# Shared Secret

*$M_A * (M_B * E_0)$ requires four dimensions*



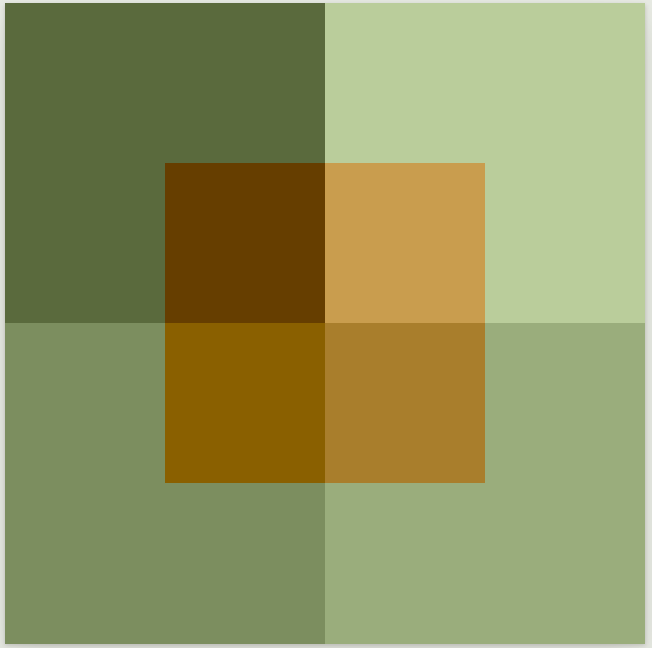
# Here *we go* Again!

01 **Kernel:** points on $E_B$ from $x_A$, over $\mathbb{F}_{p^2}$

02 **Gluing:** three special steps, ending over $\mathbb{F}_p$

03 **Isogeny:** $e - 3$ generic $(2, 2, 2, 2)$-steps

04 **Invariant:** of a fourfold, then hash



# Kernel Data on $E_B$

- A deterministic basis $\langle P, Q \rangle = E_B[2^{e+2}]$
- With $s = (x_A - 1)/2$, Theorem 4.1 boils down to

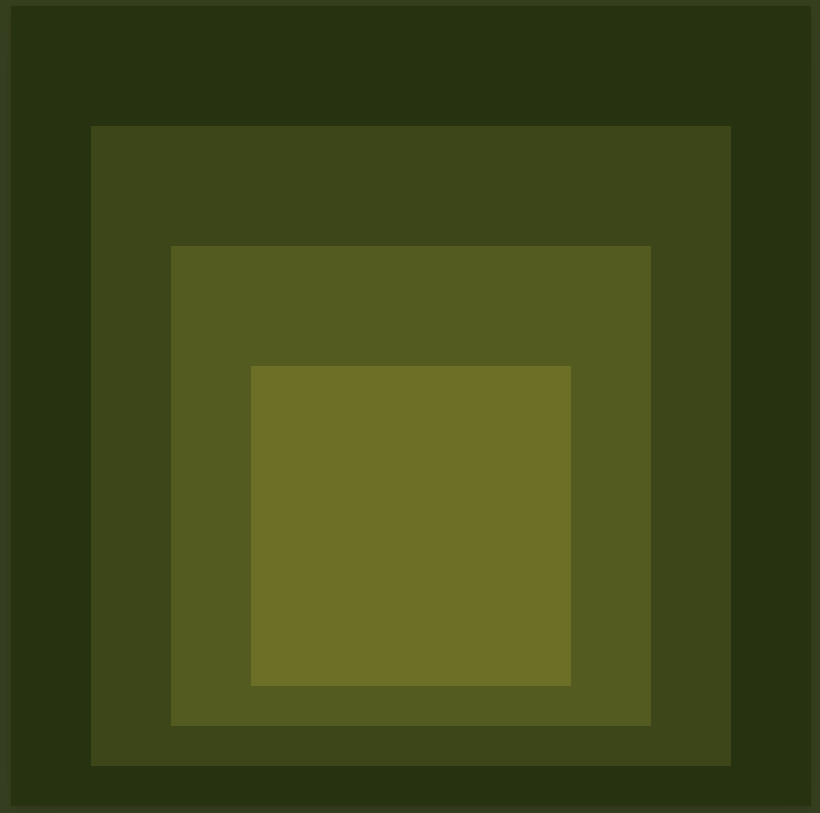
$$T_3 = \big([s+1]Q,\ [s]Q\big), \qquad T_4 = \big(-[s+1]P,\ -[s]P\big)$$

- The $\sigma$-components are determined, so we only store points on $E_B$

*$\sigma(P) = (X^p : Y^p : Z^p)$: three conjugations in $F_{p^2}$, tracked algebraically in the gluing formulas.*

# A Kernel from *two* Ladders

- An x-only ladder gives $x([s]P)$ and $x([s+1]P)$ together
- **Lift** both to full points with Okeya–Sakurai
- $T_3, T_4$ are then additions on $E_B$
- Roughly **a fifth** of the shared-secret cost

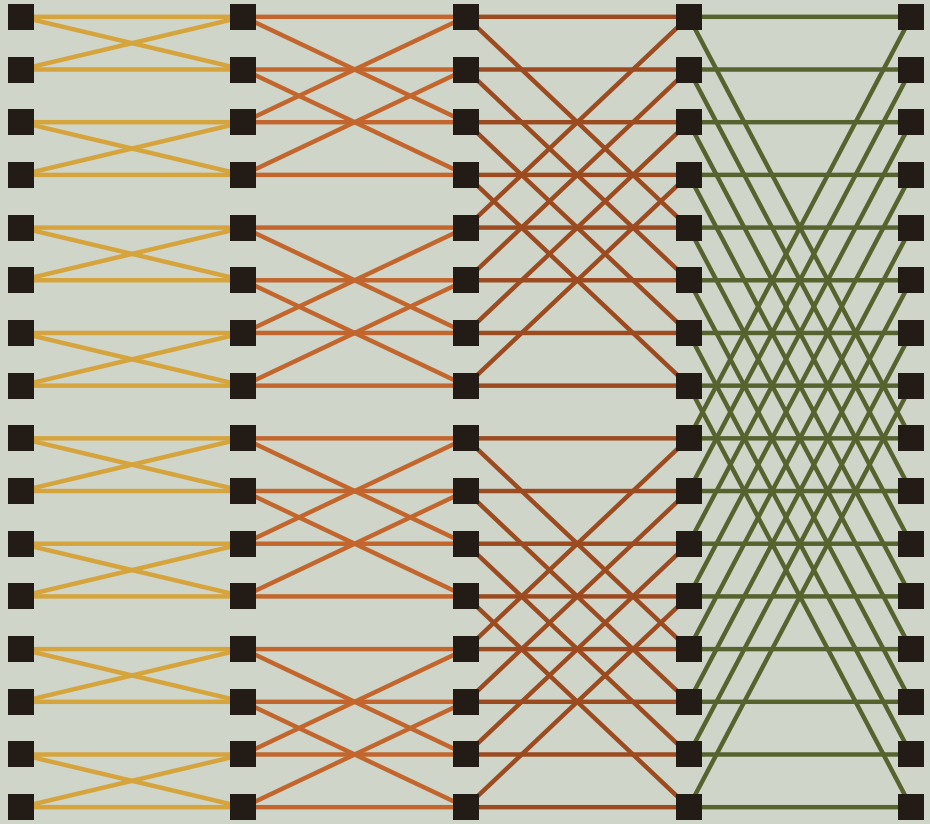


# Gluing is totally fine...

*Trust me! No, you don't need that appendix, it's fine!*

# Theta Arithmetic *is* Beautiful

- A point is **16** elements of $\mathbb{F}_p$
- Three helpers
  - **S**: coordinate squaring (16S)
  - **M**: pairwise coordinate multiplication (16M)
  - **H**: Hadamard transform (64a)
- Doubling and Images build from only these maps



# Doubling

```
def double(self, P):
    P = P.square()
    P = P.hadamard()
    P = P.square()
    P = P.pointwise_mul(self.inv_null_dual_sq)
    P = P.hadamard()
    P = P.pointwise_mul(self.inv_null)
    return P
```

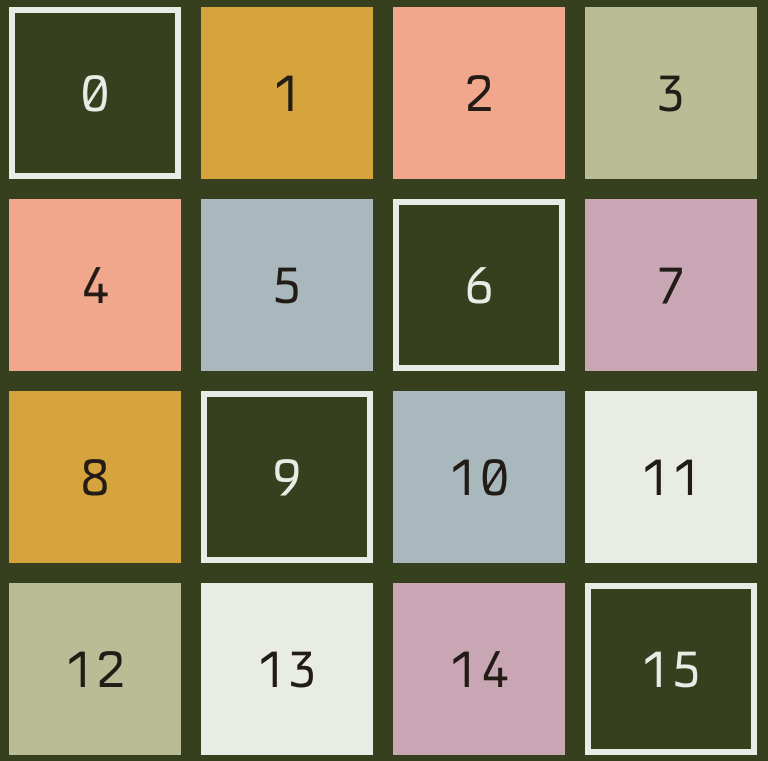
- 32S + 32M + 128a

# Images

```
def isogeny_eval(P, inv_dual_null):
    P = P.square()
    P = P.hadamard()
    P = P.pointwise_mul(inv_dual_null)
    P = P.hadamard()
    return P
```

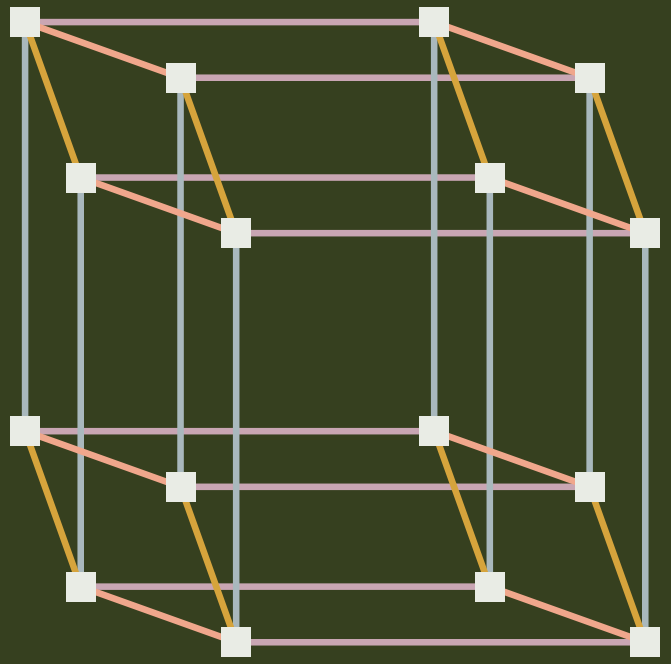
- 16S + 16M + 128a

# The $[i]$ Symmetry

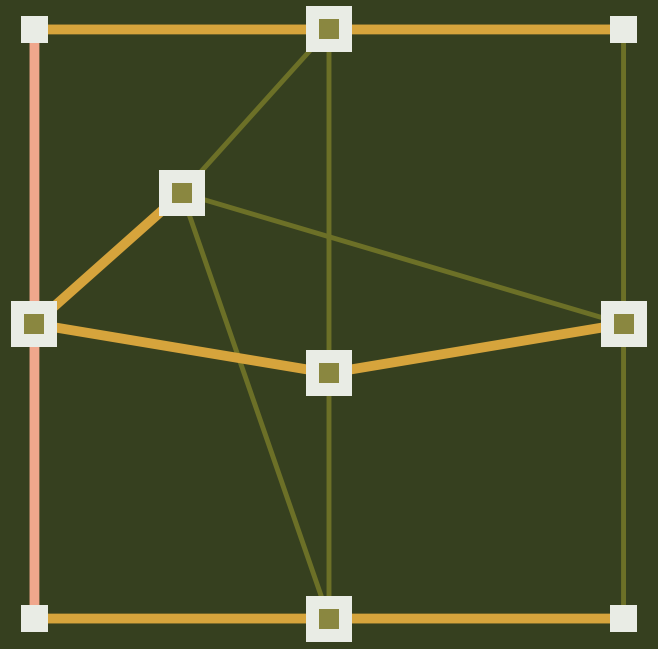



- Every fourfold in the chain has an automorphism $[i]$, with $[i]^2 = -1$
- On coordinates: swap $x_j \leftrightarrow x_{\bar{j}}$, bit-reversed
- The null point is fixed, so **16 coordinates, 10 values**

# Symmetric Kernels



**Generic.** *The codomain graph is a tesseract: one direction per kernel point, four points*



**MIKE.** *Modulo [i], ten vertices (nested squares are pairs); a spanning tree needs only $T_3$ and $T_4$*

# The Codomain

```
def isogeny_codomain(K1, K2):
    P8 = K1.square().hadamard()
    Q8 = K2.square().hadamard()
    inv = compute_inv_dual_codomain(P8, Q8)   # 25M
    return ThetaStructure.from_inv_dual(inv), inv
```

- Thanks to $[i]$: only **10** coordinates computed
- 32S + 25M + 128a, plus a precomputation

# Compressed Precomputation

| *From* $1/\tilde{\theta}$ | *16 coordinates* | *10 coordinates* |
|---|---|---|
| Invert | $45M$ | $27M$ |
| Hadamard | $64a$ | $34a$ |
| Square, Hadamard, Invert | $16S + 64a + 45M$ | $10S + 34a + 27M$ |
| Invert | $45M$ | $27M$ |
| **Total** | $135M + 16S + 128a$ | $81M + 10S + 68a$ |

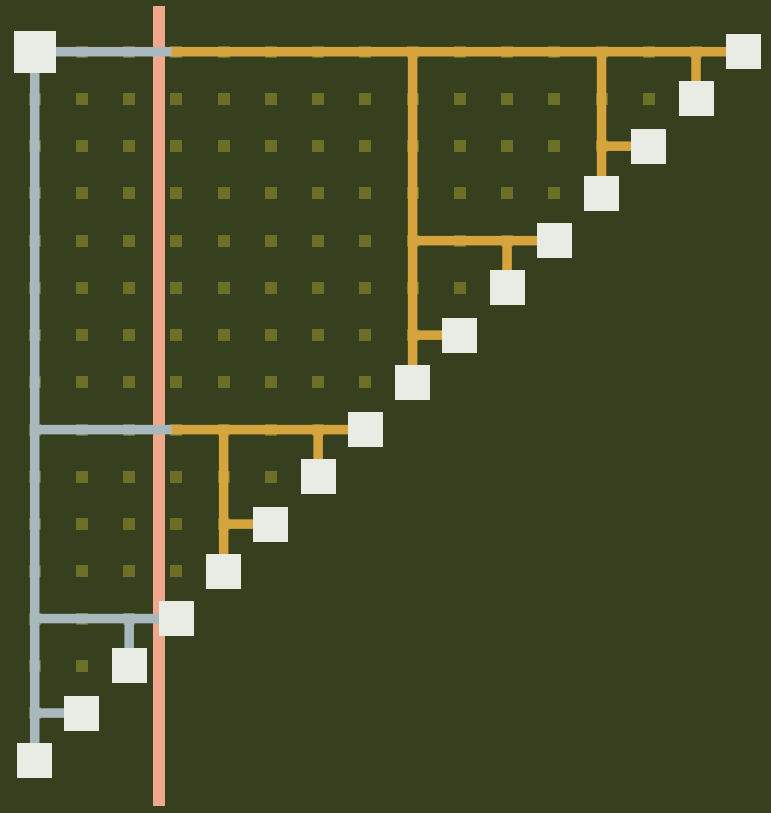
*Projective inversion: Montgomery's trick without the inversion, 3(n – 1) multiplications.*

# Comparing Dimensions

| *Operation* | *Dimension 1 over* $\mathbb{F}_{p^2}$ | *Dimension 4 over* $\mathbb{F}_p$ |
|---|---|---|
| Double | $4M + 2S + 4a$ | $32M + 32S + 128a$ |
| Codomain | $4S + 5a$ | $2 \cdot (16S + 64a) + 25M$ |
| Precomputation | — | $81M + 10S + 68a$ |
| Image | $6M + 2S + 6a$ | $16M + 16S + 128a$ |
| Points carried | **1** | **2** |

*Dimension 1 is key generation: four-isogenies on Montgomery curves, x-only.*

# Double *in* Dimension One



- Weierstrass doubling on $E_B$ is cheaper than in dimension four
- But every point we double in dimension one is pushed through an expensive gluing
- Optimising the chain means balancing these two costs
- The rest of the chain can be computed with a balanced strategy

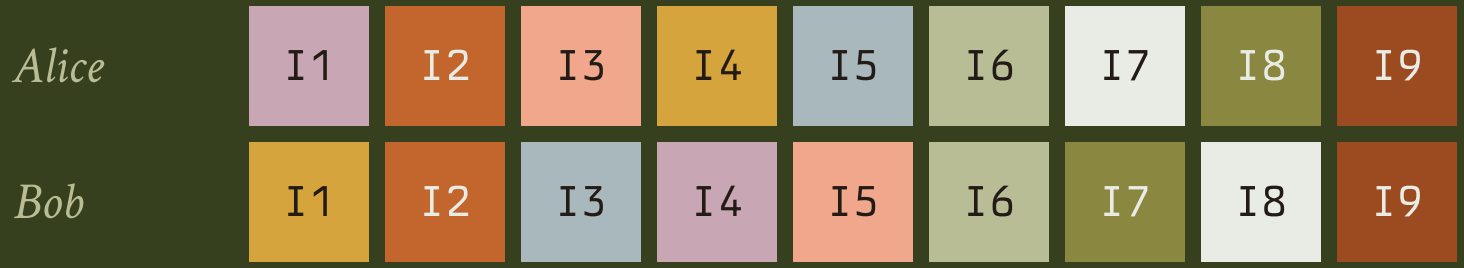
# Isomorphic Fourfolds

- Alice and Bob land on **isomorphic**, not equal, fourfolds
- Generic invariants are still an open problem
- Solved for MIKE thanks to ~~Sabrina and Damien~~ the symmetry of $[i]$

# Isomorphic Fourfolds

- From the null point $(a_0 : \cdots : a_{15})$, take ten squares:

$$I_0 = (a_0 + a_6 + a_9 + a_{15})^2, \quad \ldots, \quad I_9 = 4(a_3 - a_5)^2$$




- The same nine values $I_k/I_0$, up to **72** permutations
- We could just sort them lexicographically and pick one…

# Absolute Invariants

- Sabrina saves the day!

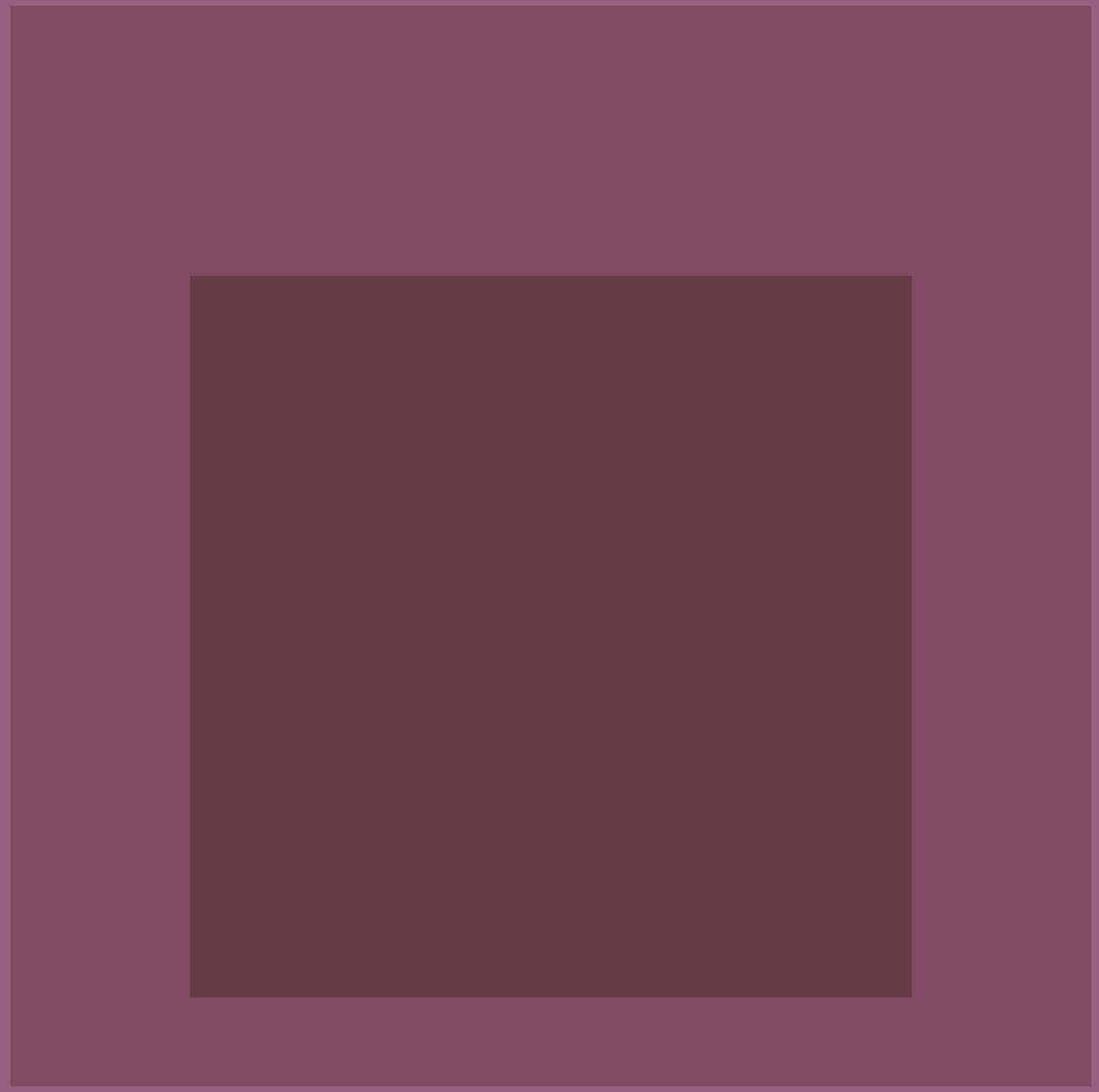
$$J_k = \sum_{j=1}^{9} \left( \frac{I_j}{I_0} \right)^k, \qquad k = 2, 3, 4, 5$$



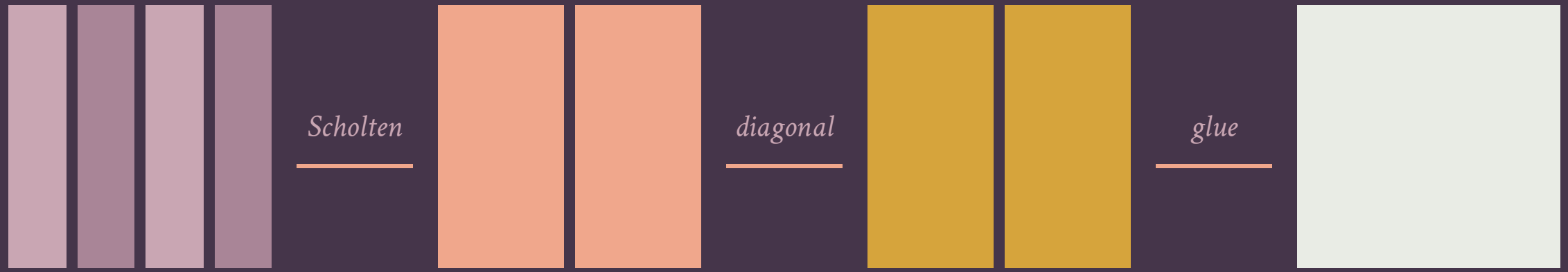

- We finally hash the invariants $\mathsf{SHA3}(J_2, J_3, J_4, J_5)$

# Gluing

*It's just three isogenies in a trench coat...*



# Meet Vincent Adultman...




$(E \times E^\sigma)^2$

*four curves,* $\mathbb{F}_{p^2}$

$W'(E)^2$

*two surfaces,* $\mathbb{F}_p$

$S_U \times S_V$

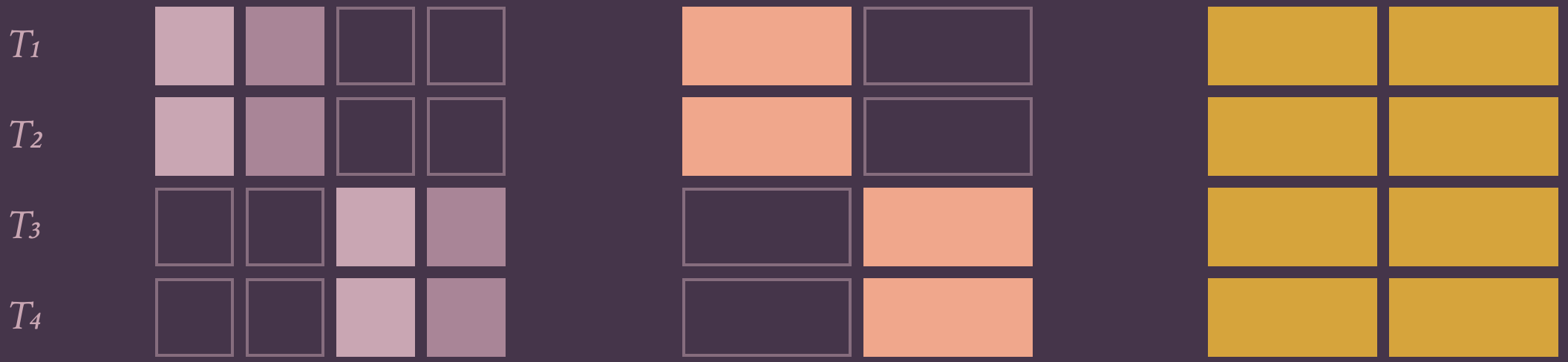
*two surfaces,* $\mathbb{F}_p$

$X$

*one fourfold,* $\mathbb{F}_p$

*The first three 2-isogenies of the chain.*

# The Kernel Says When *to* Glue




Step 1: splits

*on $(E \times E^\sigma)^2$, two copies of $\Gamma$*

Step 2: splits

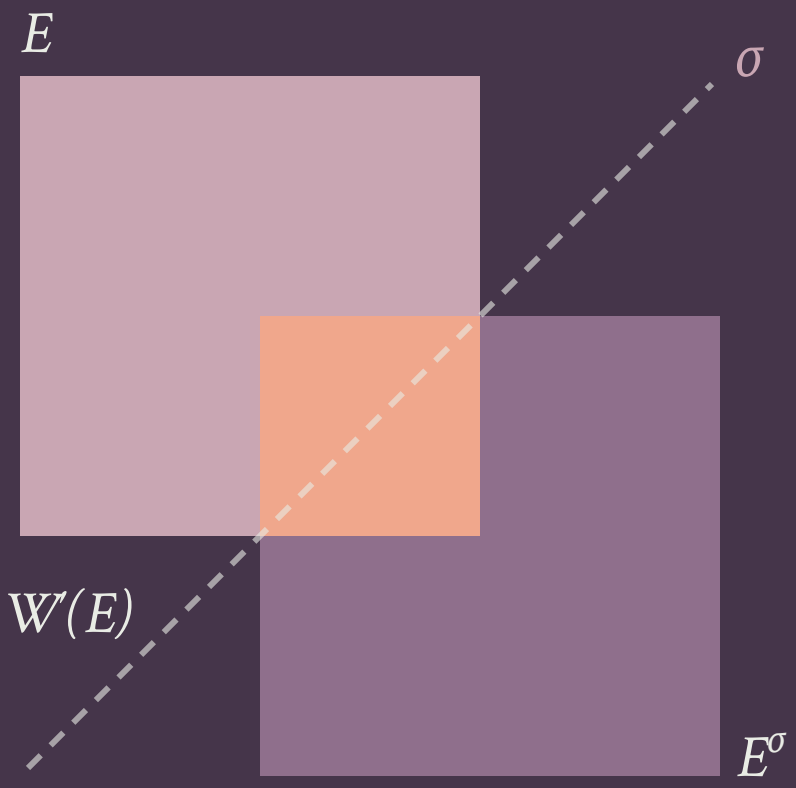
*on $W'(E)^2$, into $\phi_U$ and $\phi_V$*

Step 3: glue

*on $S_U \times S_V$, nothing splits*

$$x \equiv 3 \bmod 8 \implies x \pm 1 \text{ has at most two factors of } 2$$

# Step 1: Scholten




- $\sigma$ swaps $E$ and $E^\sigma$: the **pair** is defined over $\mathbb{F}_p$, neither curve is
- $\Gamma$ glues along a $\sigma$-stable kernel
- So $W'(E)$ has its theta null point over $\mathbb{F}_p$: we've **left** $\mathbb{F}_{p^2}$
- We never compute $\sigma(P)$: only norms and conjugates

*Codomain ≈ 100M, images ≈ 115M, $P \pm T$ ≈ 96M, in $F_p$ multiplications.*

# Step 2: Diagonal

- $W'(E)^2 \to S_U \times S_V$: two $(2,2)$-isogenies
- Split by Frobenius: $U$ fixed, $V$ negated
- Ordinary dimension-2 theta arithmetic over $\mathbb{F}_p$

*Codomain $\approx$ 50M, images $\approx$ 12M, in $F_p$ multiplications.*

# Step 3: Welcome to 4D

## Generic image

```
def isogeny_eval(P, inv_dual):
    P = P.square()
    P = P.hadamard()
    P = P.pointwise_mul(inv_dual)
    P = P.hadamard()
    return P
```

## Gluing image

```
def gluing_eval(P_plus, P_minus, inv_T3):
    P = P_plus.pointwise_mul(P_minus)
    P = P.hadamard()
    P = P.pointwise_mul(inv_T3)
    P = P.hadamard()
    return P
```

- Squaring $P \odot P$ becomes $(P + T) \odot (P - T)$ for a fixed $T$
- So $P \pm T$ are computed on the curve, **before** Scholten

*Codomain $\approx$ 390M, images $\approx$ 92M, in $F_p$ multiplications.*

# It's Easy When you Write it Like This

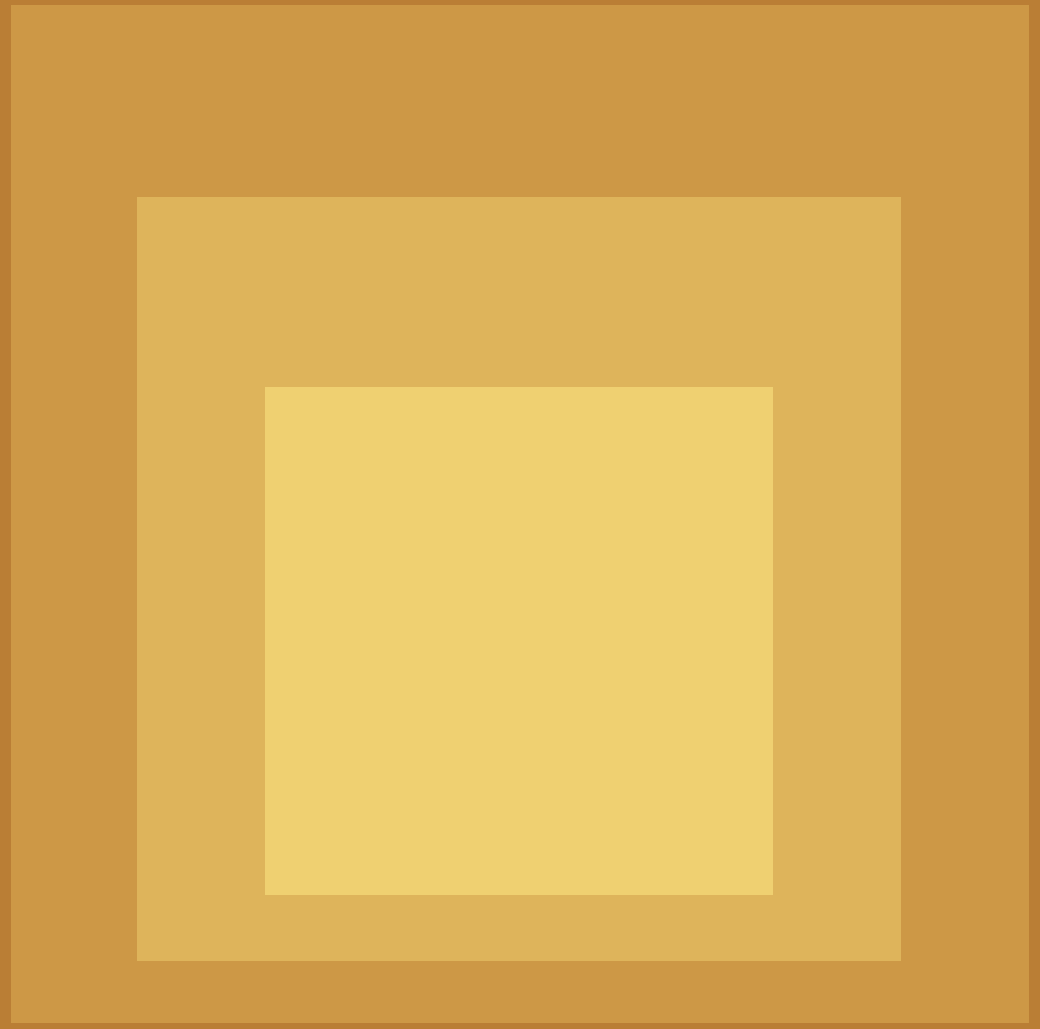
```
def gluing_isogeny(E, bases, chain_kernel):
    # E x E^σ -> A2 x A2
    diag_ker, glue_ker, plus, minus = scholten_isogeny(E, *bases, chain_kernel)

    # A2 x A2 -> A2 x A2
    glue_ker, plus, minus = diagonal_isogeny(diag_ker, glue_ker, plus, minus)

    # A2 x A2 -> A4
    return dim_four_gluing_isogeny(glue_ker, plus, minus)
```

# Implementing MIKE

*Validation, Constant Time and Performance.*



# Validating Public Keys

01 A canonical encoding of a non-zero $A \in \mathbb{F}_{p^2}$, with $A^2 \neq 4$

02 $A \notin \mathbb{F}_p$: we can't glue these, so we reject them (an $O(1/\sqrt{p})$ fraction of honest keys)

03 **Supersingular**: the sampled basis has exactly the expected order

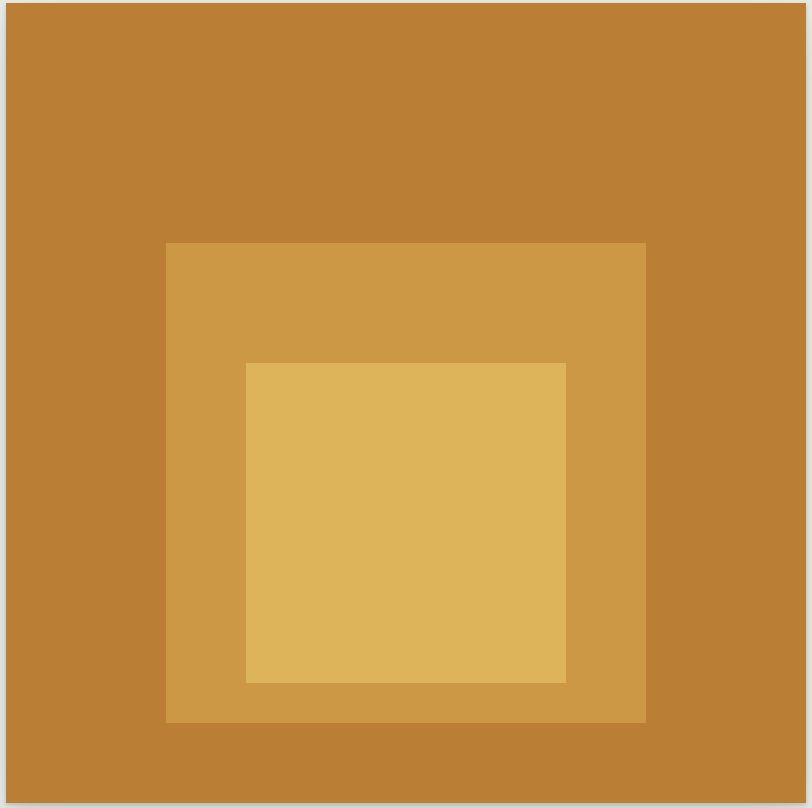
04 **Normalised**: $A$ is the largest of its twelve

*Checks 3 and 4 are essentially free: both happen during kernel generation.*



# Constant Time

- MIKE is **easy** to implement with **existing** constant-time tools
- Secrets are used in only **two places**: the ladders and normalisation
- Ladders: one conditional swap per bit
- Normalisation: masked byte comparisons
- There is **no branching** during isogeny computations



# All Levels

| *Level* | *p* | *PK (bytes)* | *KG x86* | *SS x86* | *KG ARM* | *SS ARM* |
|---|---|---|---|---|---|---|
| I | $633 \cdot 2^{308} - 1$ | 80 | 2.15 | 16.0 | 1.09 | 7.80 |
| III | $593 \cdot 2^{474} - 1$ | 122 | 6.96 | 49.3 | 3.25 | 24.3 |
| V | $317 \cdot 2^{628} - 1$ | 160 | 13.7 | 94.2 | 6.36 | 47.6 |

*MIKE, constant-time C with assembly, in megacycles. ARM: Apple M4 Pro. No AVX.*

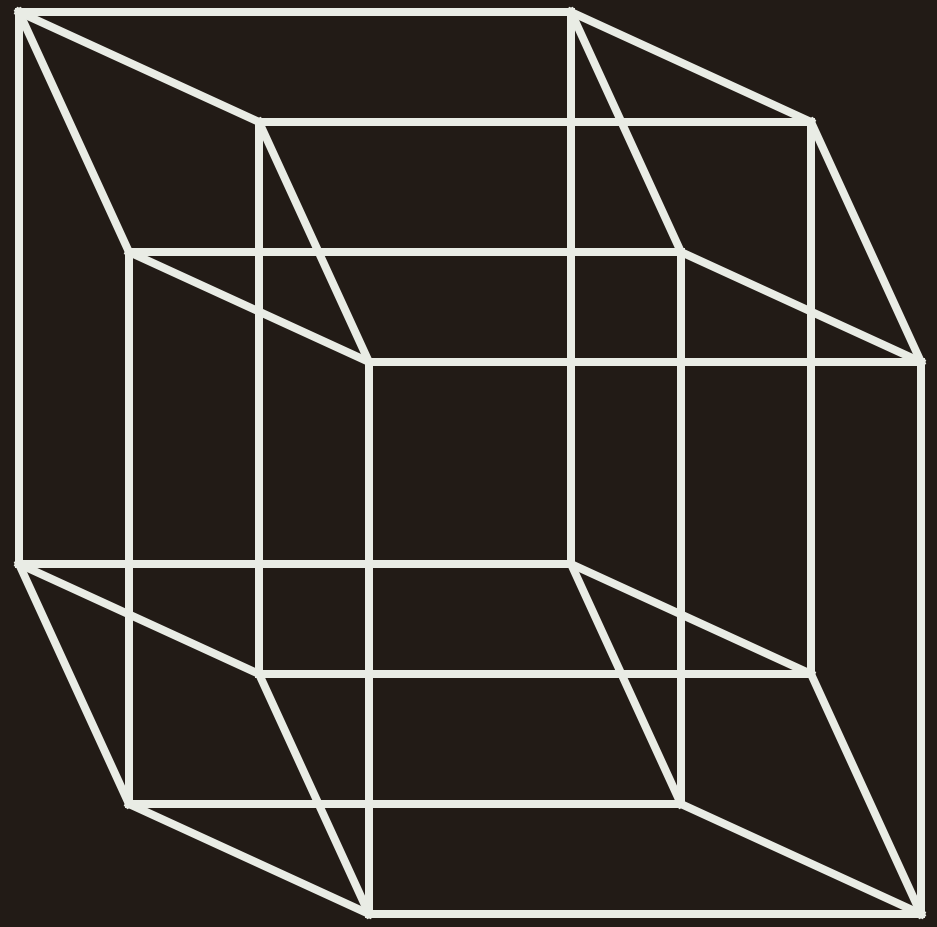
**The past**

The past is gone
If you liked the past
then hard luck
It's gone
and you had better get used to it

*David Shrigley, Creative Writing (2026)*

# Thank *You*

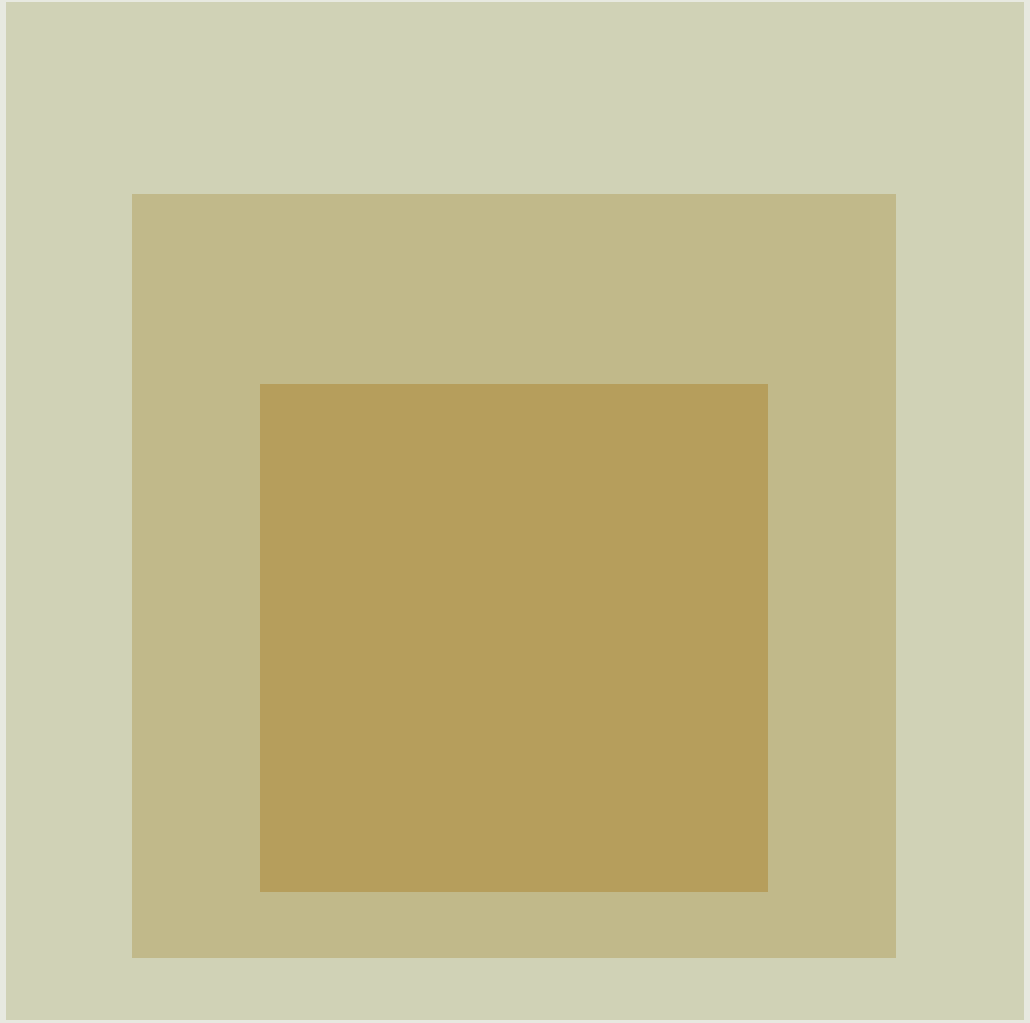
*Questions?*



ncc group

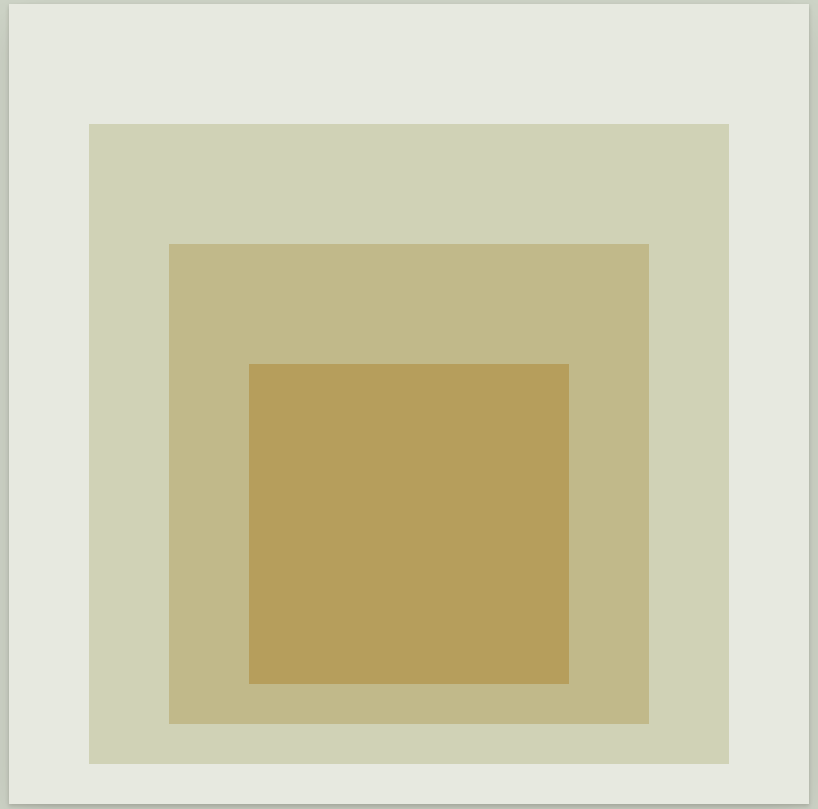
# More Scholten?

*Craig's Dream*



# Key Generation over $\mathbb{F}_p$

- Start from $E_4$: four fixed steps, with $x \equiv 3 \bmod 16$
- Scholten-glue $E_4 \times E_4^{\sigma}$ into a surface over $\mathbb{F}_p$
- A $(2,2)$-chain of length $e-4$, all over $\mathbb{F}_p$
- Split at the end: two square roots in $\mathbb{F}_p$



# Key Generation *in* Dimension Two

```
def public_key_dim_two(self, x):
    z = x >> 4   # skip E0 -> E4
    K4 = R + z * S   # ladder + lift

    u, v, w = barycentric_coordinates(K4, T)   # x(K4 ± T)
    kernel = superglue_dual_image(u, v, w)   # into dim 2, over F_p
    kernel = cond_neg_last(kernel, z & 1)

    OB, kernel = two_two_isogeny_chain(OA, kernel, self.e - 4)
    return normalized_A_from_dual_theta(OB, kernel)   # 2 sqrt in F_p
```

# Dimension Two is Better!




*Key generation in megacycles, constant-time C with assembly. Portable C gains more: 1.22–1.26×.*

# Dimension Two is Better?




*On ARM64 (Apple M4 Pro) the speed-up shrinks.*