Proving iris code calculation

Enabling Model Upgrades Under Self-Custody

Interim progress report

Presentation roadmap

1. Problem statement (the development of)
2. Quantization and precision
3. Encoding convolutions in circuit
4. Circuit walkthrough
5. Demo & benchmarking
6. Next steps

Upgrade protocol for self-custody

- The iris code is derived from the normalized iris image via a model (convolutional filters)
- Self custody: normalized iris image is stored on the user's smartphone
- Upgrade = WC deploys new, improved model
- Users must compute the new iris code on their device
- … *and* prove the correctness of that computation.

Upgrade protocol for self-custody

More precisely, user proves they know an image that:

- produces the known iris code under the old model
- … *and* produces the new iris code under the new model
- i.e. a conjunction of two instances of the statement S=S(C,F):

  “user knows an image that produces iris code ***C*** *using the filter values* ***F****”*
- WLOG we consider how to prove any such statement.
- As a non-interactive argument of knowledge.

Further requirements

(to be tackled in the new year):

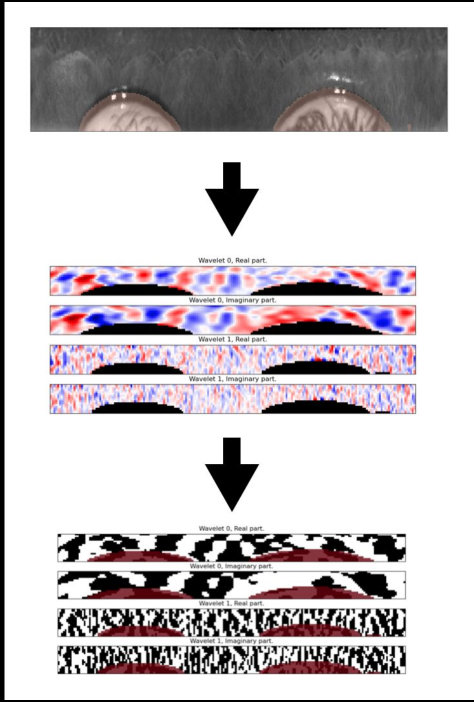
- prove in zero knowledge (disclose *no* information about image to WC).
- ensure the authenticity of the image.

Calculating iris codes

- Inputs: normalized image (resp. mask); filter values.
- Convolve filter at specified placements.
- Outputs: iris code (b&w bitmap) (resp. transformed mask, red).

A single pipeline works for image & mask separately. (passing in appropriate filter values and thresholds.)

WLOG, we consider only the normalized image here.

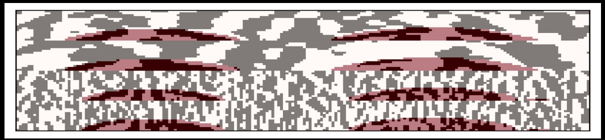


Quantization

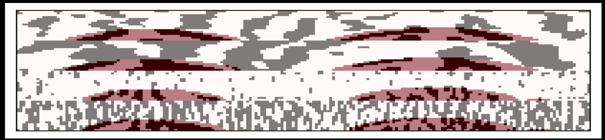
Quantization & precision

- Normalized image is 32 bit f.p.
- Filters are complex-valued with 64 bit f.p. components
- All inputs must be quantized (represented as integral values) for the proof system.
- Coarser quantizations (via rounding) allow for faster provers & smaller proofs
- … ***but*** maintaining precision is critical to Sybil detection!

Iris code

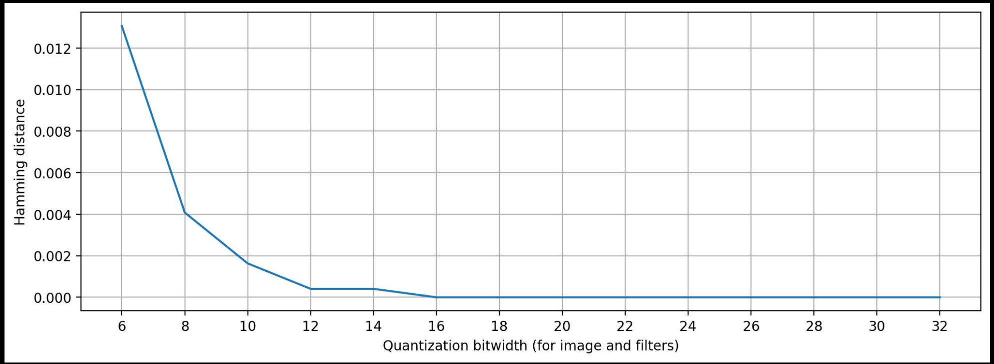


Iris code with errors due to over aggressive quantization



How much precision is enough?

- Experiment: quantize image & filter to different bitwidths, then calculate iris code
- Calculate Hamming distance to the original iris code.
- Achtung: based on a single sample! (More data required).
- So use large margin-of-error for proof system: 26 bits each for filter and image.



How to circuitize convolutions?

Circuitizing convolutions: approaches

- Build a circuit using addition and multiplication gates.
- Encode as multiplication of univariate polynomials, and use FFT to compute the product (from zkCNN).
- Encode as a single matrix multiplication [1], then use Thaler's super-efficient IP for matrix multiplication.

Adopted matrix multiplication: fits perfectly into Remainder (since both it the IP & GKR are sumcheck-based).

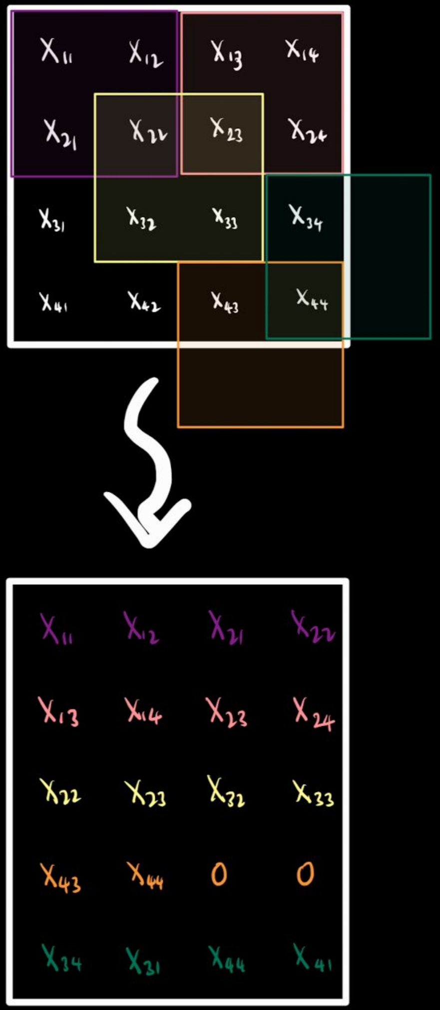
[1] Balbás et al., Modular Sumcheck Proofs with Applications to Machine Learning and Image Processing, ACM CCS 2023.

Encoding 2d convolution as matrix multiplication: The “re-routed input matrix”

Example:

- 4x4 input image
- 5 placements of a 2x2 convolutional filter (the 5 colored squares)
- Each placement corresponds to a row in the “re-routed image matrix” (at bottom).
- Assumes pad-with-0 vertically & cyclic padding horizontally.
- Resulting “re-routed matrix” has dimensions:

  #placements x (#filter entries)

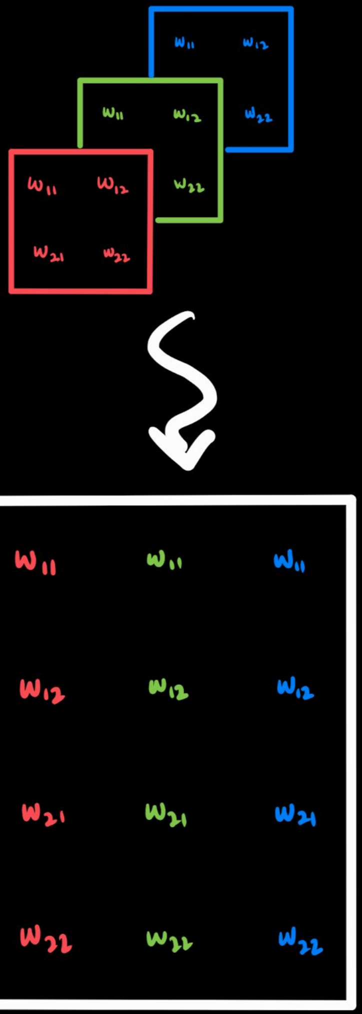


Encoding 2d convolution as matrix multiplication:
The “flattened kernel matrix”

Continuing the previous example:

- Suppose we have 3 separate 2x2 convolutional filters (colored) (equivalently, a 2x2 convolutional layer with 3 output channels).
- Each filter becomes a single column in the flattened kernel matrix.
- “Flattened kernel matrix” has dimensions:

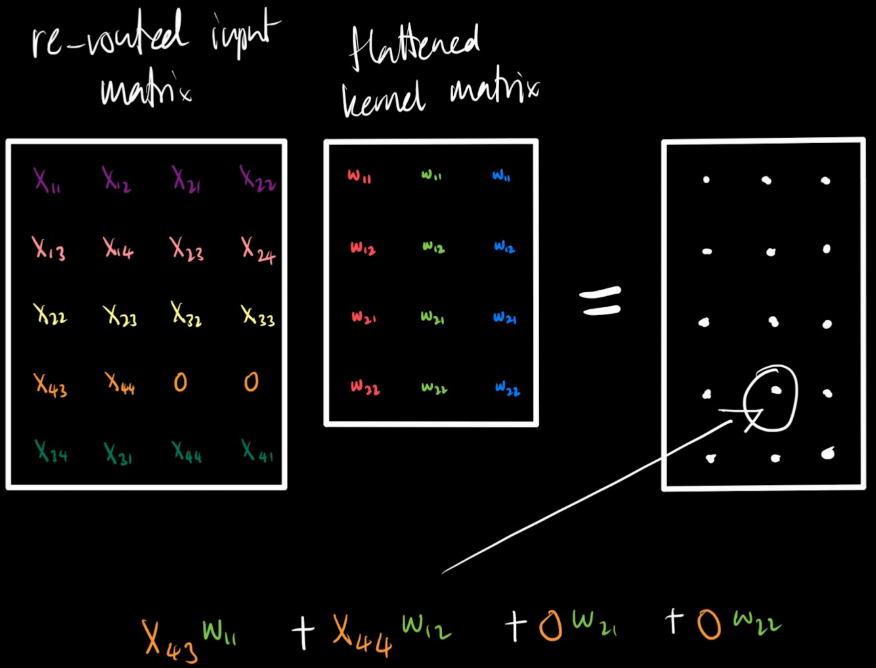
  (#filter entries) x # filters.



Encoding 2d convolution as matrix multiplication: Putting it all together

The product matrix:

- Each entry computes one placement of one convolutional filter
- Row determines placement
- Column determines filter



Circuit Walkthrough

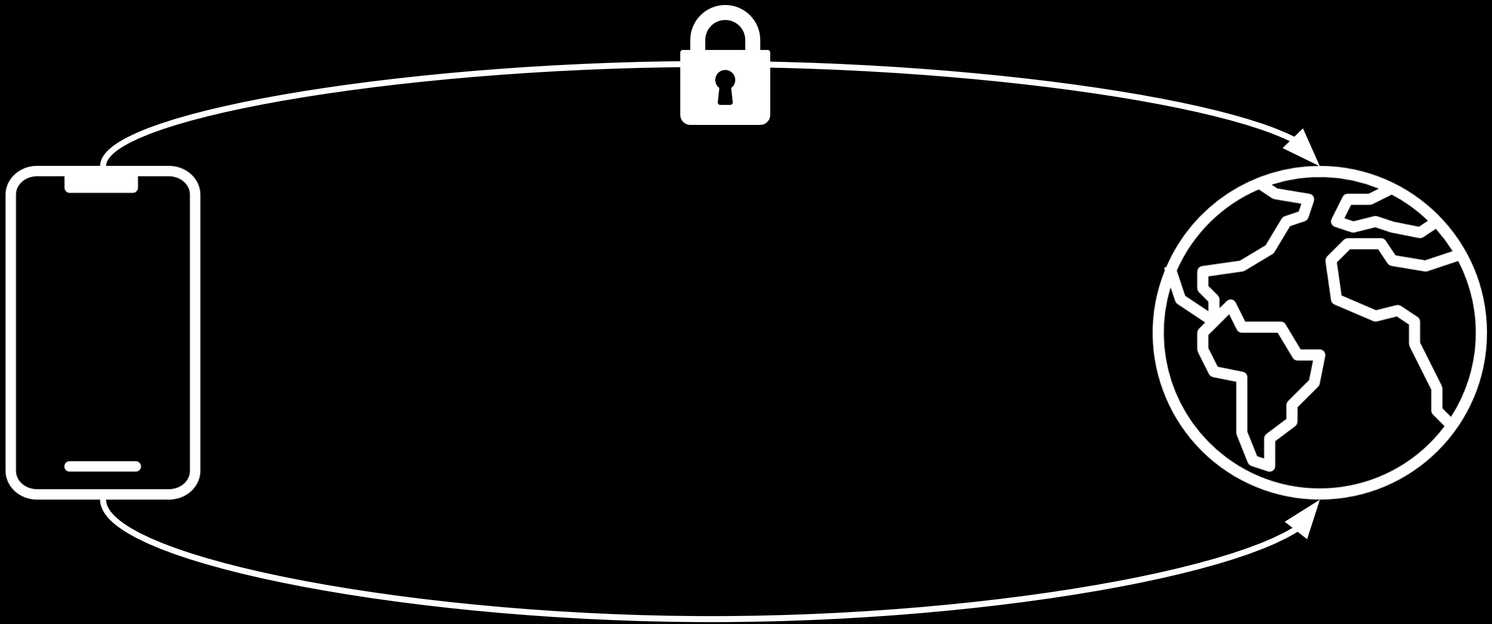
General Problem Statement

"Under a circuit C, an input x correctly results in an output y."

C: Circuitized version of the iriscode model operations

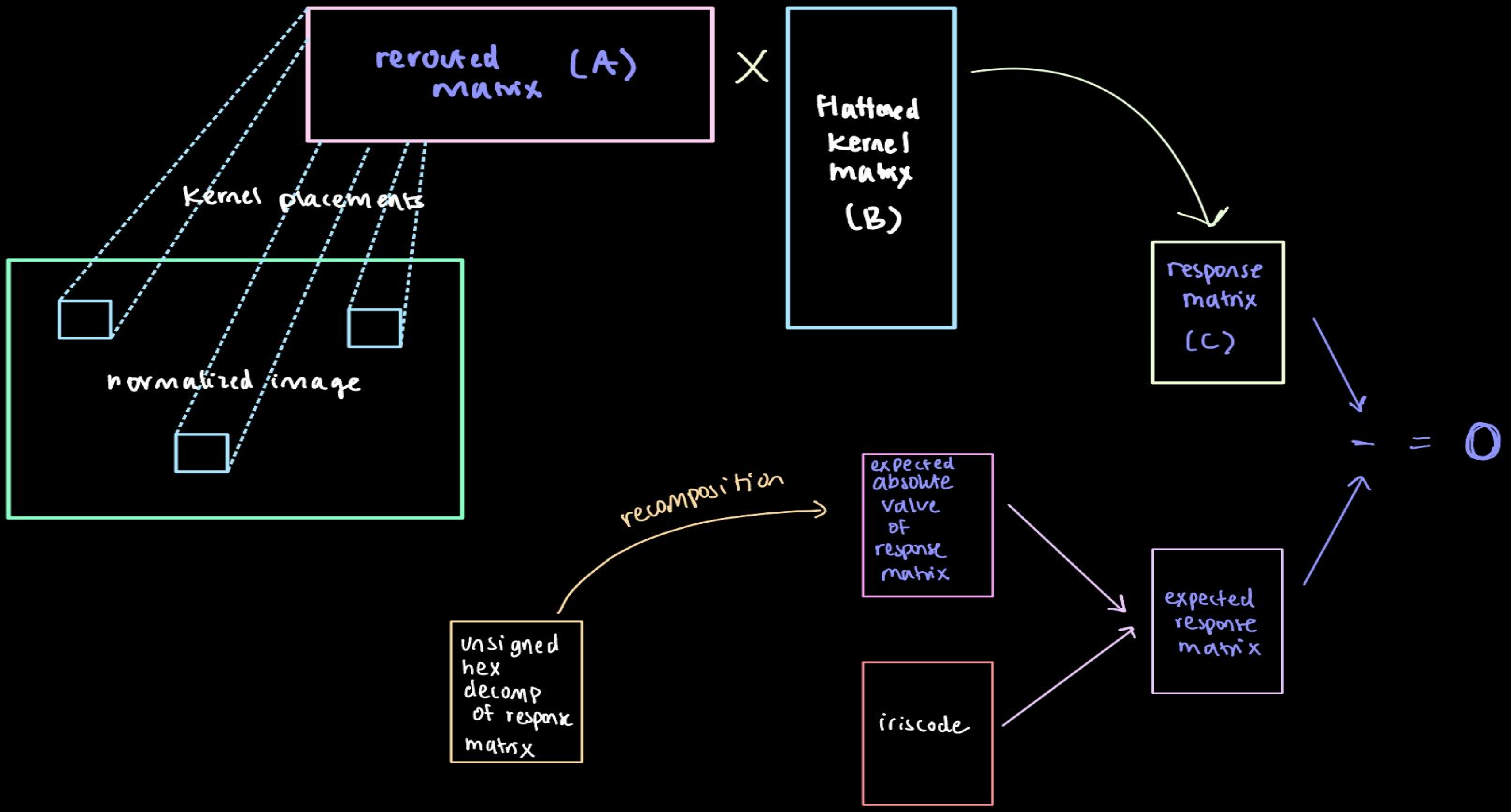
x: Normalized image

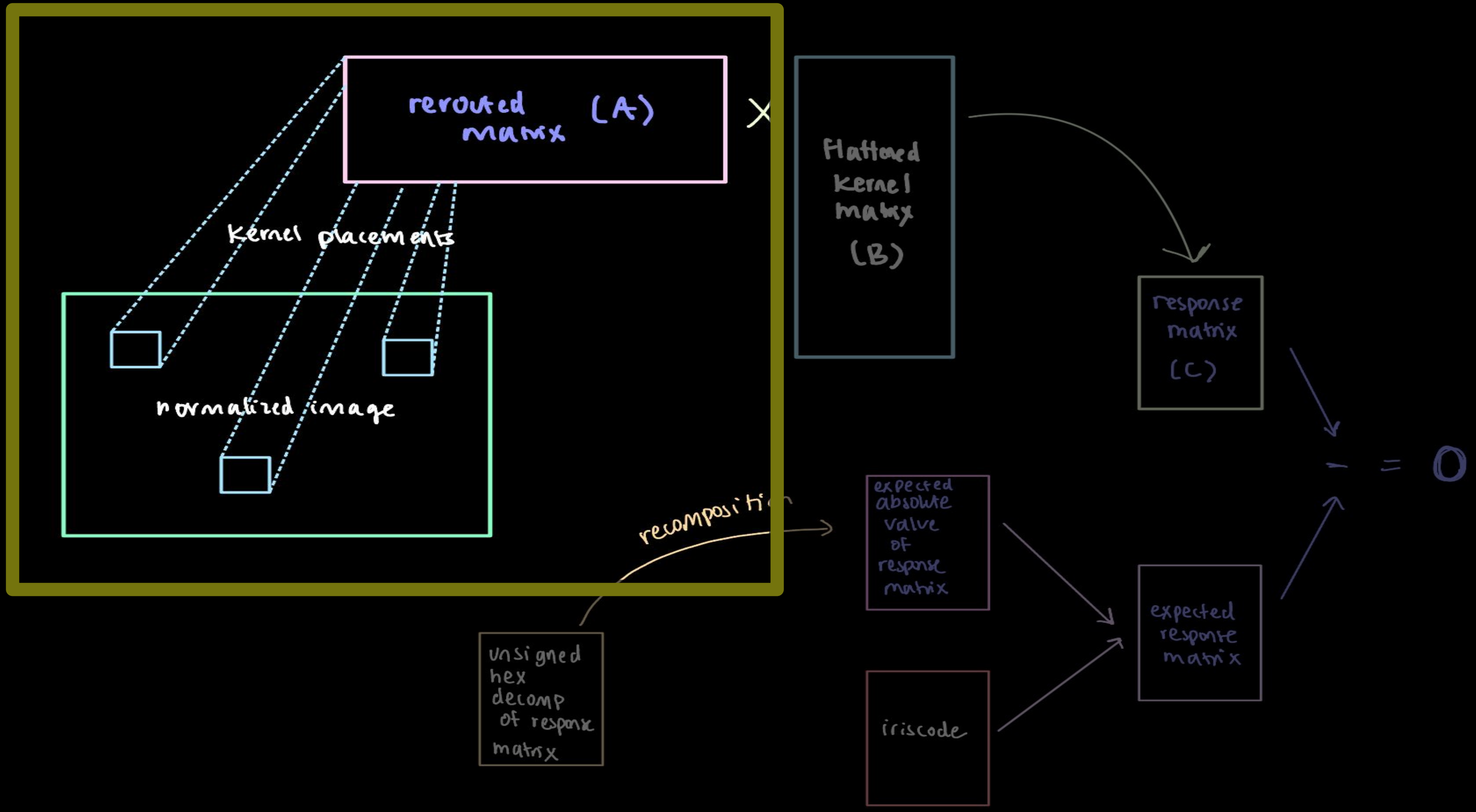
y: Iriscode



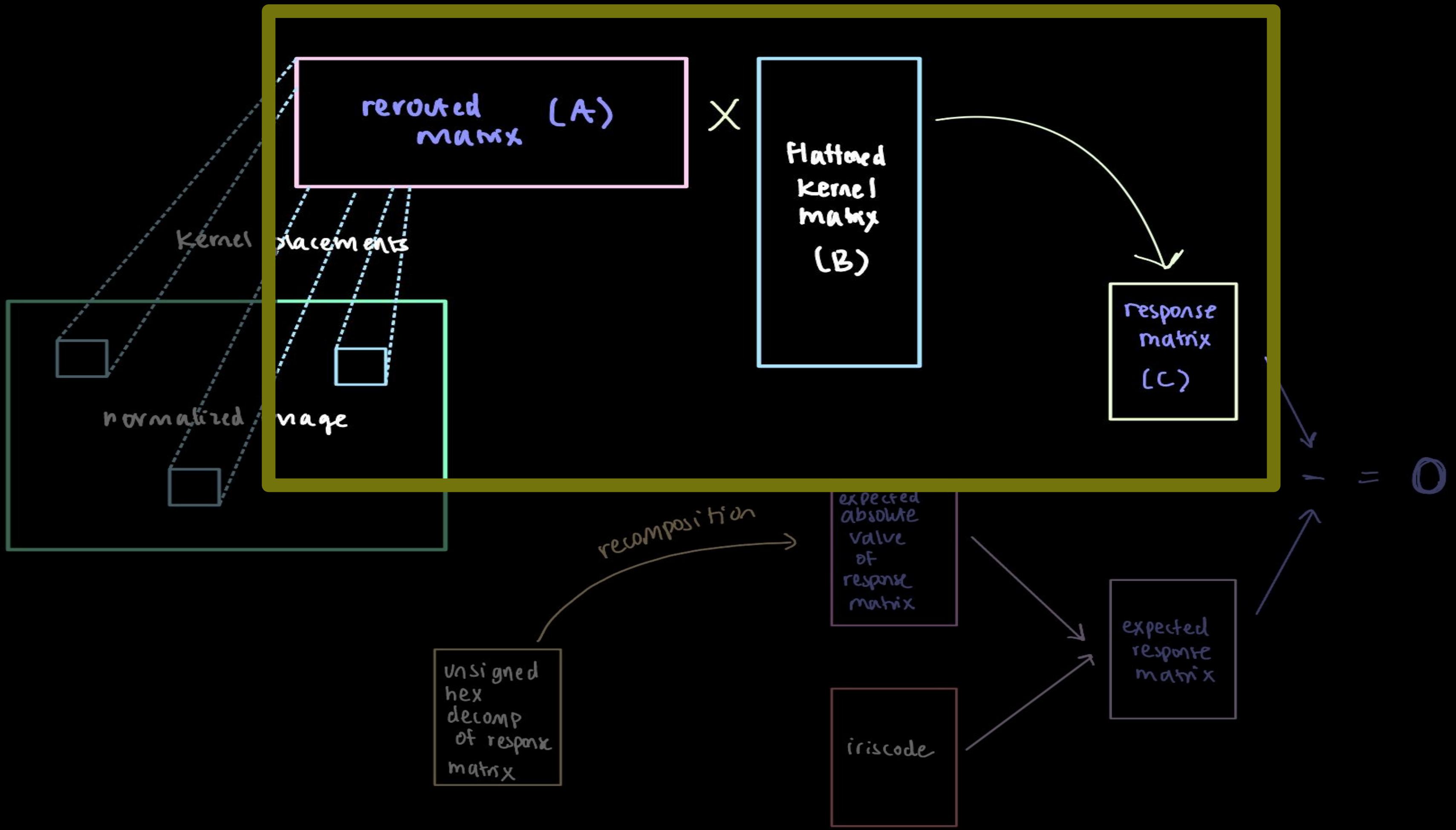
Prover:
Smartphone User

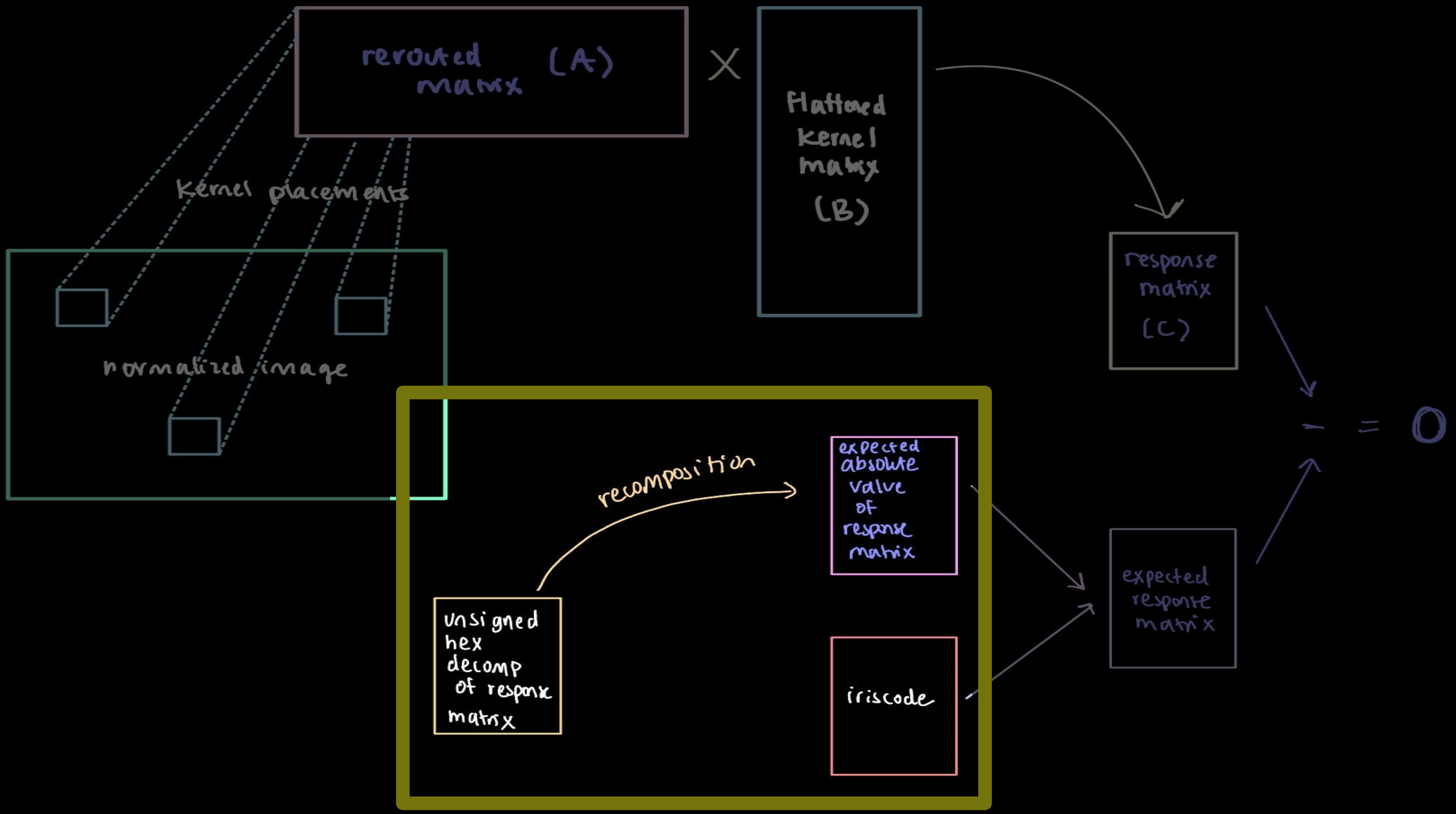
Verifier:
Worldcoin Servers

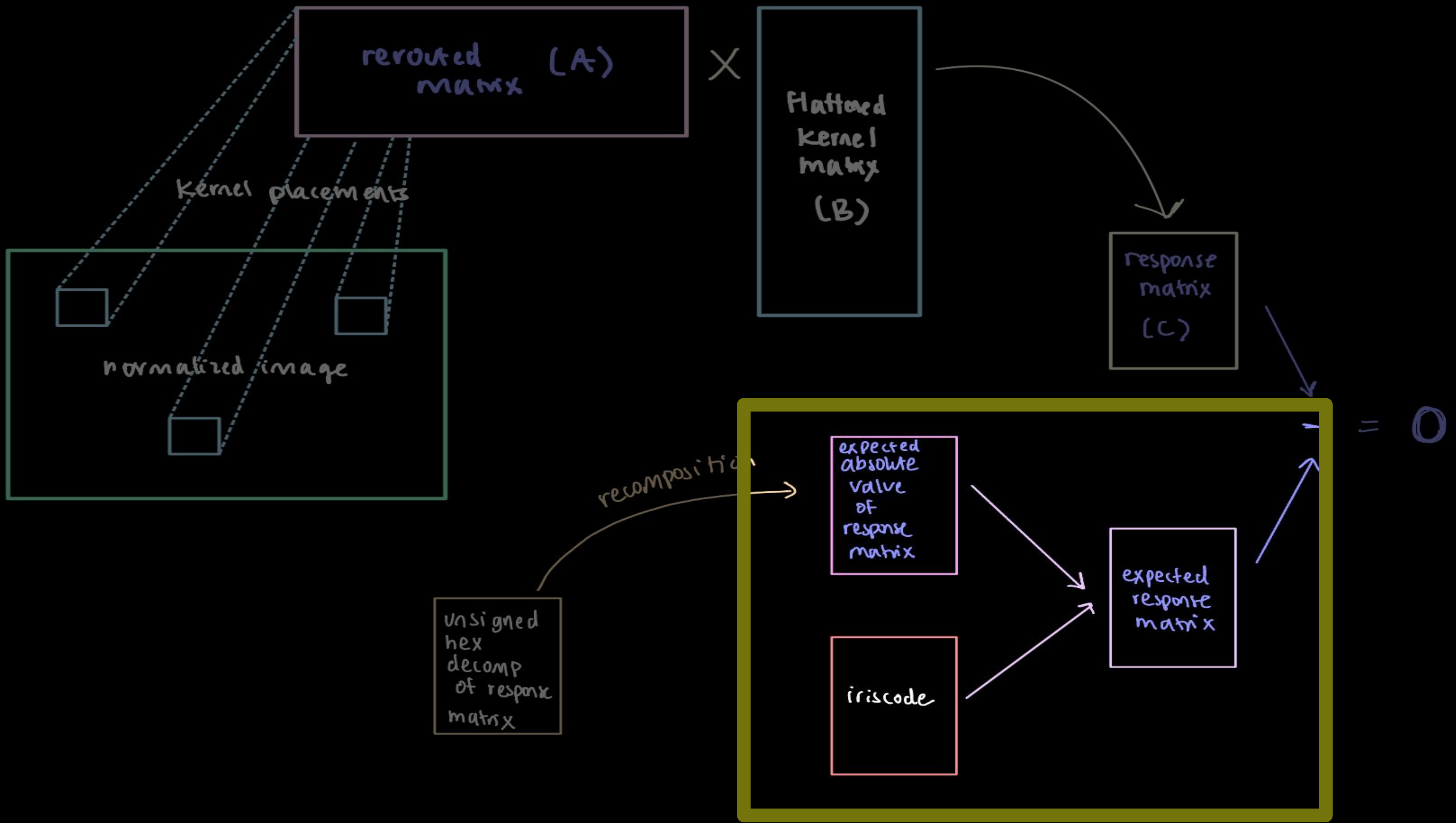

rerouted matrix (A)
X
Flattened kernel matrix (B)
kernel placements
normalized image
response matrix (C)
- = 0
recomposition
expected absolute value of response matrix
unsigned hex decomp of response matrix
expected response matrix
iriscode



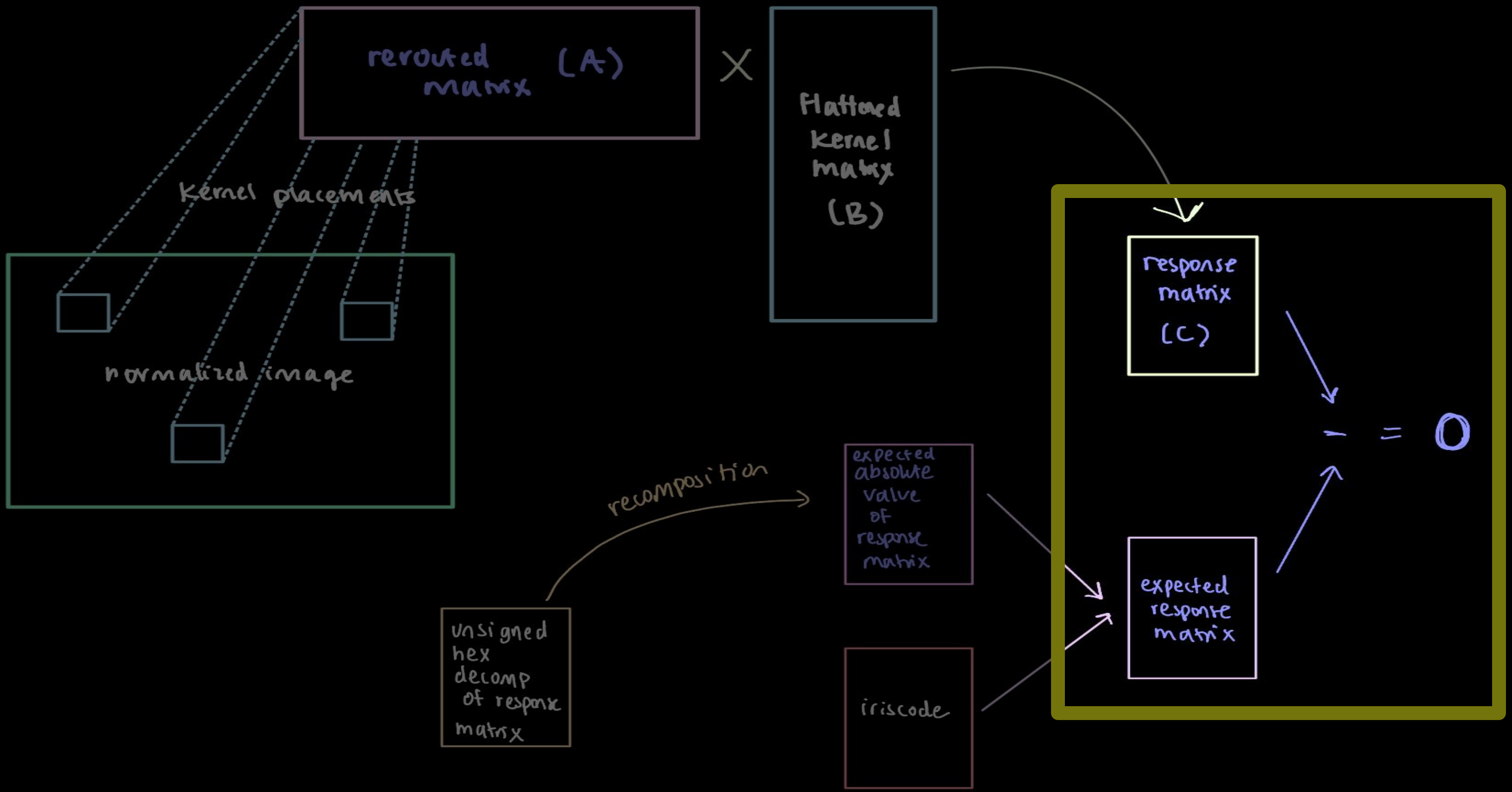
rerouted matrix (A)
X
Flattened kernel matrix (B)
kernel placements
normalized image
response matrix (C)
recomposition
expected absolute valve of response matrix
unsigned hex decomp of response matrix
iriscode
expected response matrix
= 0

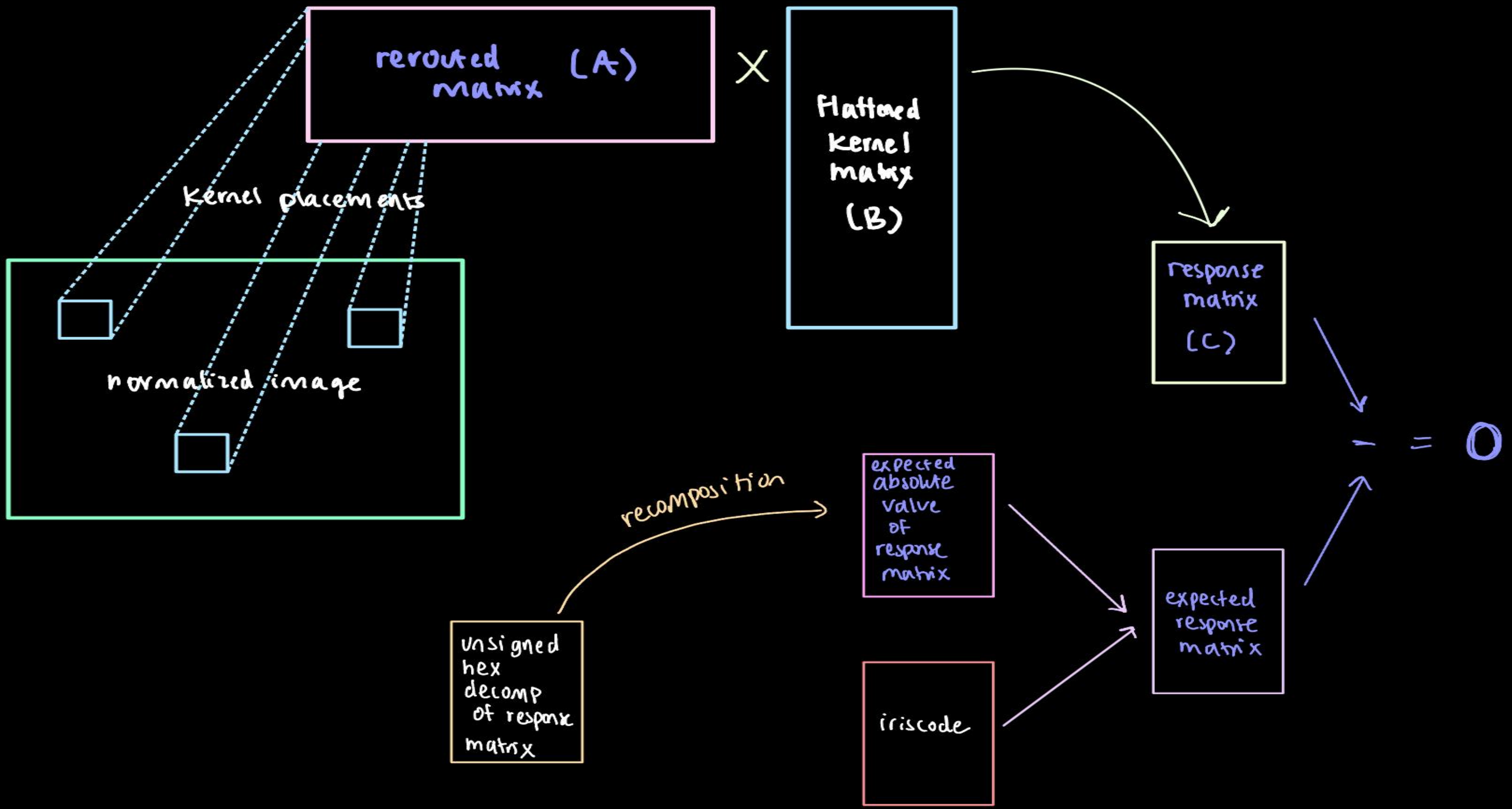

rerouted matrix (A)
X
Flattened kernel matrix (B)
response matrix (C)
kernel placements
normalized image
recomposition
expected absolute value of response matrix
unsigned hex decomp of response matrix
iriscode
expected response matrix
= 0


rerouted matrix (A)
X
Flattened kernel matrix (B)
kernel placements
normalized image
response matrix (C)
recomposition
expected absolute value of response matrix
unsigned hex decomp of response matrix
iriscode
expected response matrix
- = 0



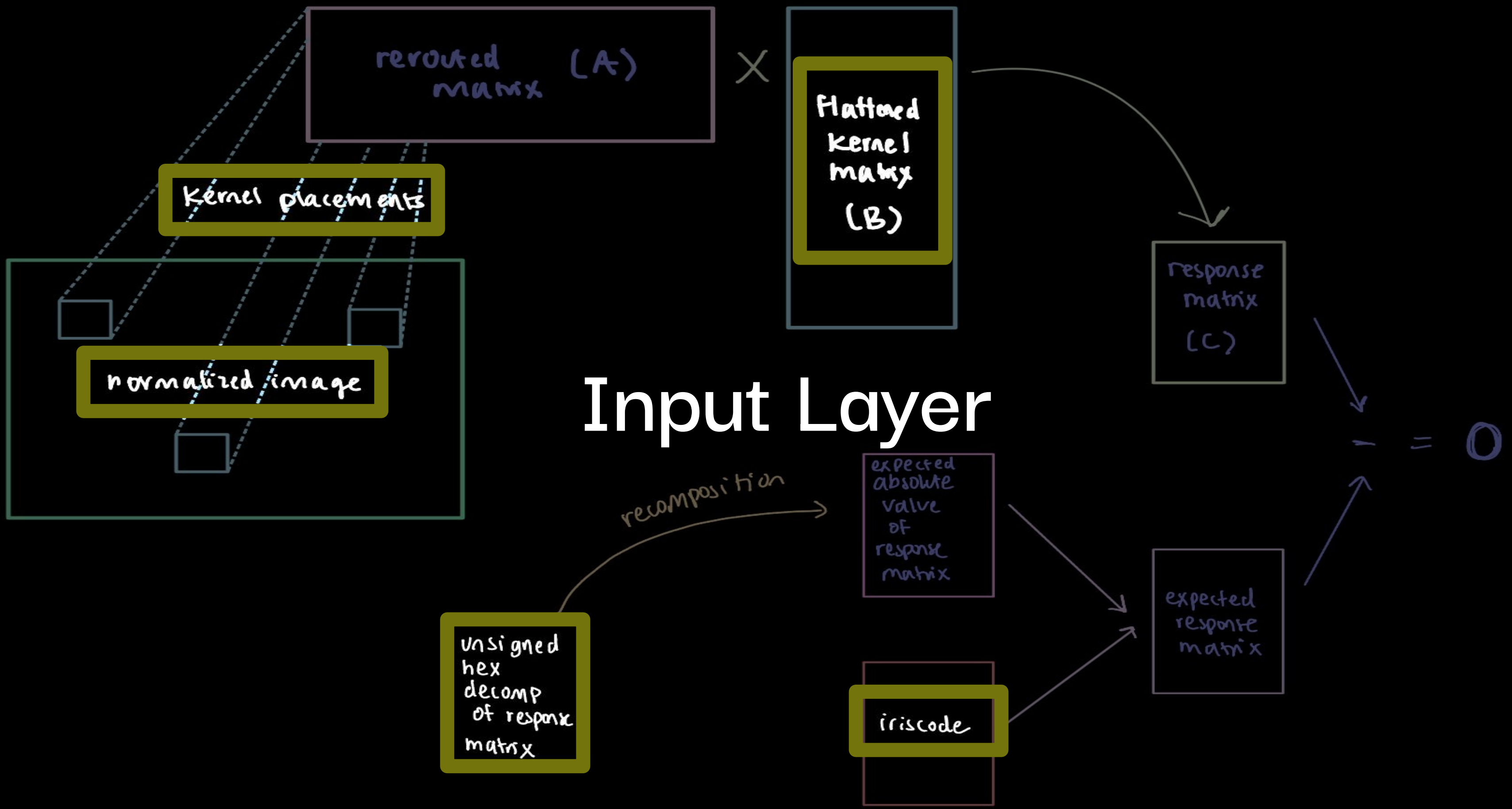
rerouted matrix (A)
X
Flattened kernel matrix (B)
kernel placements
normalized image
response matrix (C)
expected absolute value of response matrix
expected response matrix
iriscode
recomposition
unsigned hex decomp of response matrix
- = 0


rerouted matrix (A)
X
Flattened kernel matrix (B)
kernel placements
normalized image
response matrix (C)
- = 0
expected absolute value of response matrix
recomposition
unsigned hex decomp of response matrix
expected response matrix
iriscode

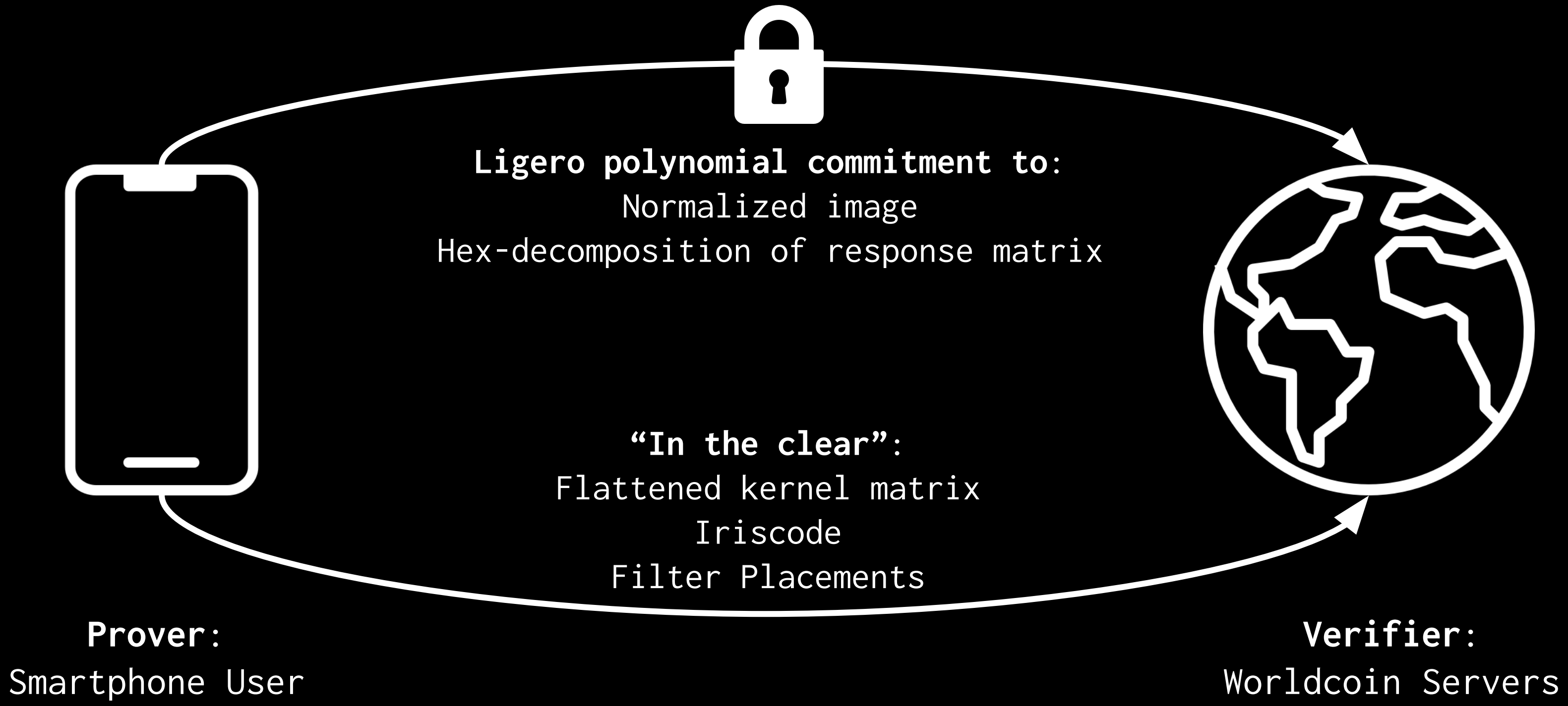


rerouted matrix (A)
X
Flattened kernel matrix (B)
kernel placements
normalized image
response matrix (C)
expected absolute value of response matrix
recomposition
unsigned hex decomp of response matrix
expected response matrix
iriscode
- = 0

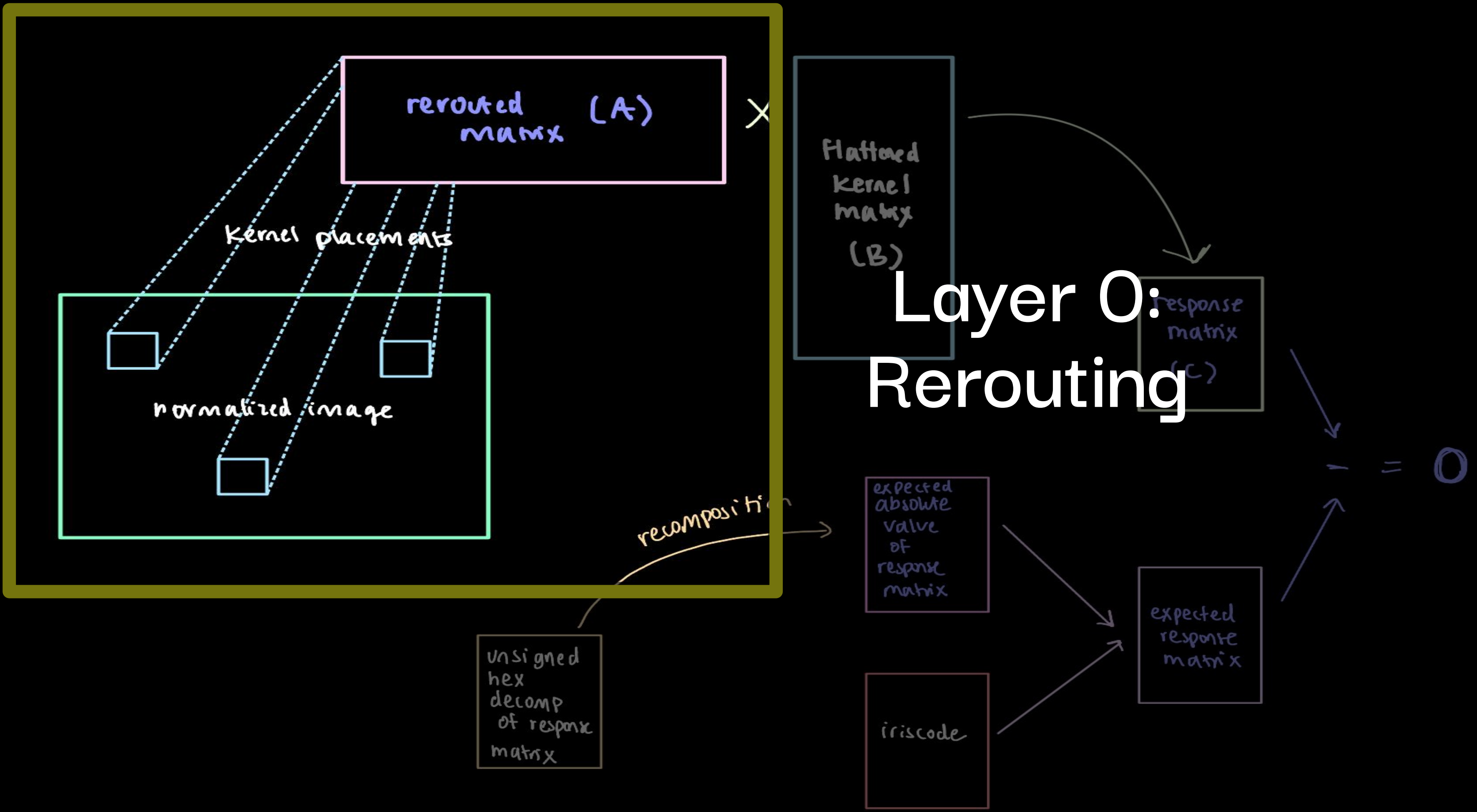
Input Layer



rerouted matrix (A)
X
Flattened kernel matrix (B)
kernel placements
normalized image
response matrix (C)
recomposition
expected absolute value of response matrix
unsigned hex decomp of response matrix
iriscode
expected response matrix
= 0


Ligero polynomial commitment to:
Normalized image
Hex-decomposition of response matrix
“In the clear”:
Flattened kernel matrix
Iriscode
Filter Placements
Prover:
Smartphone User
Verifier:
Worldcoin Servers

Layer 0: Rerouting



Layer 0: Rerouting

Caveat: filters are not applied *regularly* to the matrix; in other words, the "step size" varies within each row and column.

Solution: Use wiring predicates in order to reroute the original normalized image matrix to a matrix in which the filters are applied to every row of the matrix.

Layer 0: Rerouting

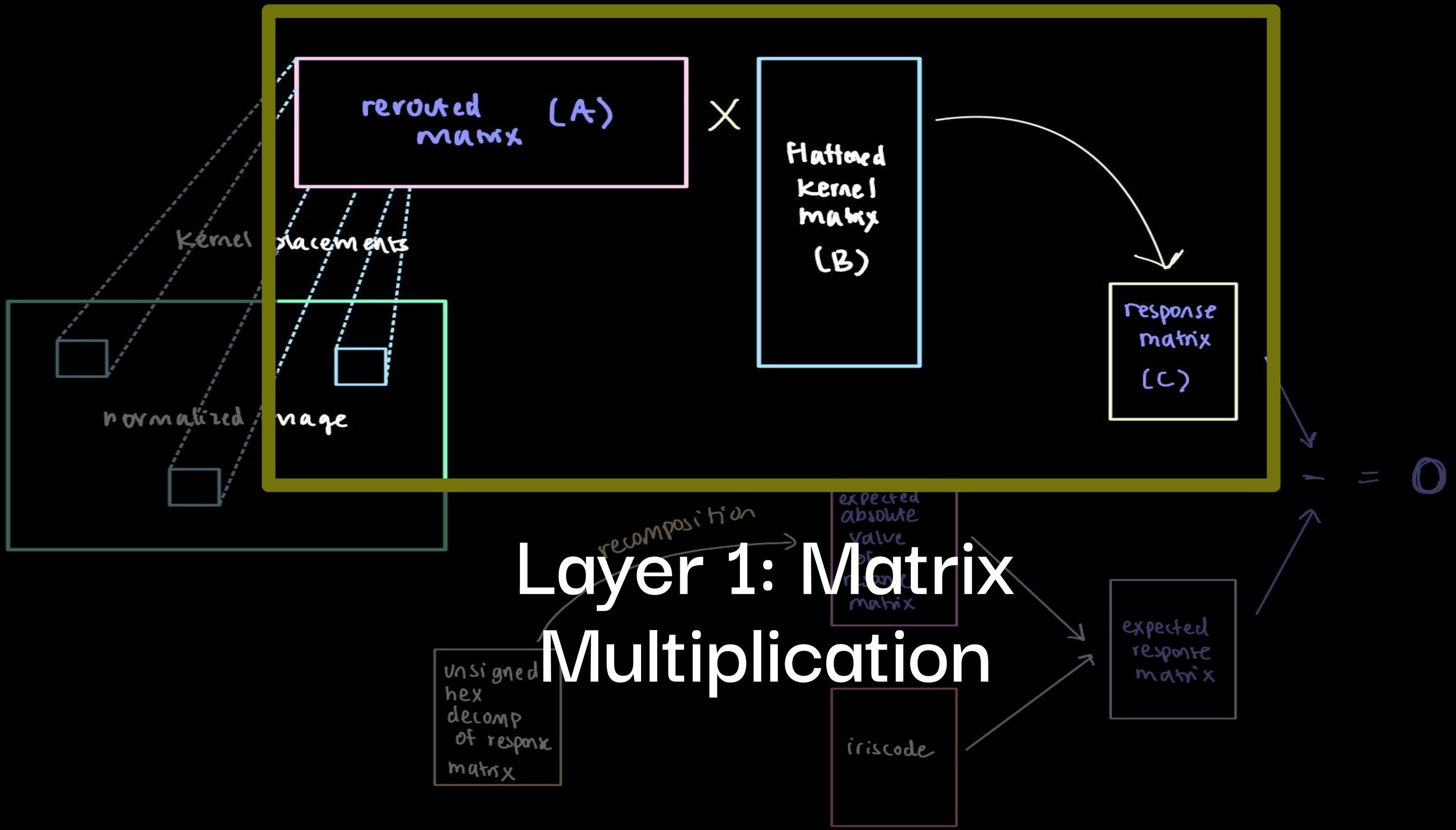
$$f(z, x) = \begin{cases} 1 & \text{if route } z \to x \\ 0 & \text{otherwise} \end{cases}$$

$$\forall (z, x) : \quad A(x) = N(z)\, f(z, x)$$

Input: kernel placements

rerouted matrix

Input: Normalized Image Matrix



Layer 1: Matrix Multiplication

Layer 1: Matrix Multiplication

represent a matrix as: $M(x_1, x_2)$

row index

column index

$$C(p_1, p_3) = \sum_{p_2} A(p_1, p_2)\, B(p_2, p_3)$$

Layer 1: Matrix Multiplication

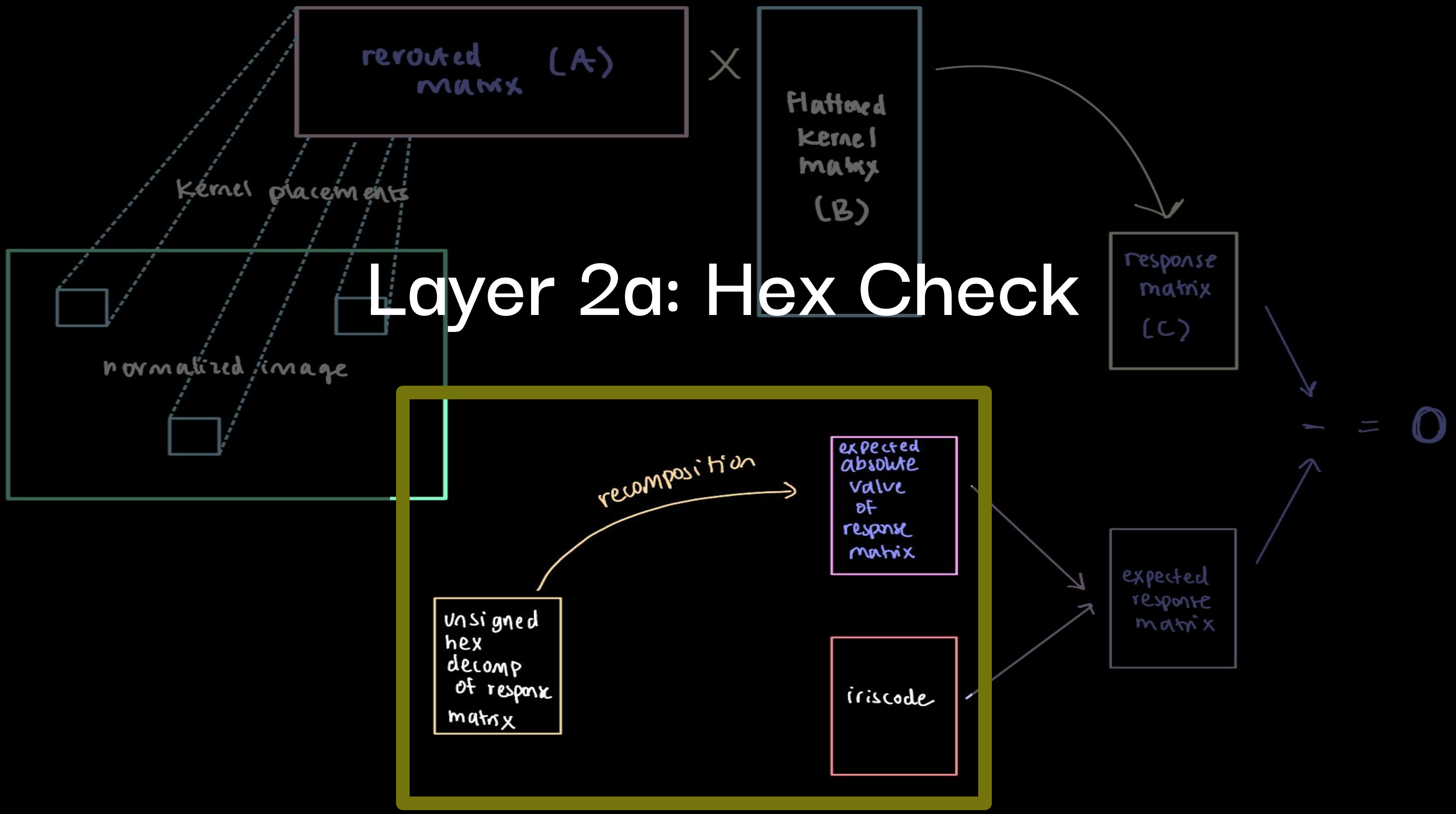
represent a matrix as: $M(x_1, x_2)$

row index

column index

$$C(p_1, p_3) = \sum_{\boxed{p_2}} A(p_1, \boxed{p_2) \; B(p_2,} \; p_3)$$

Layer 2a: Hex Check



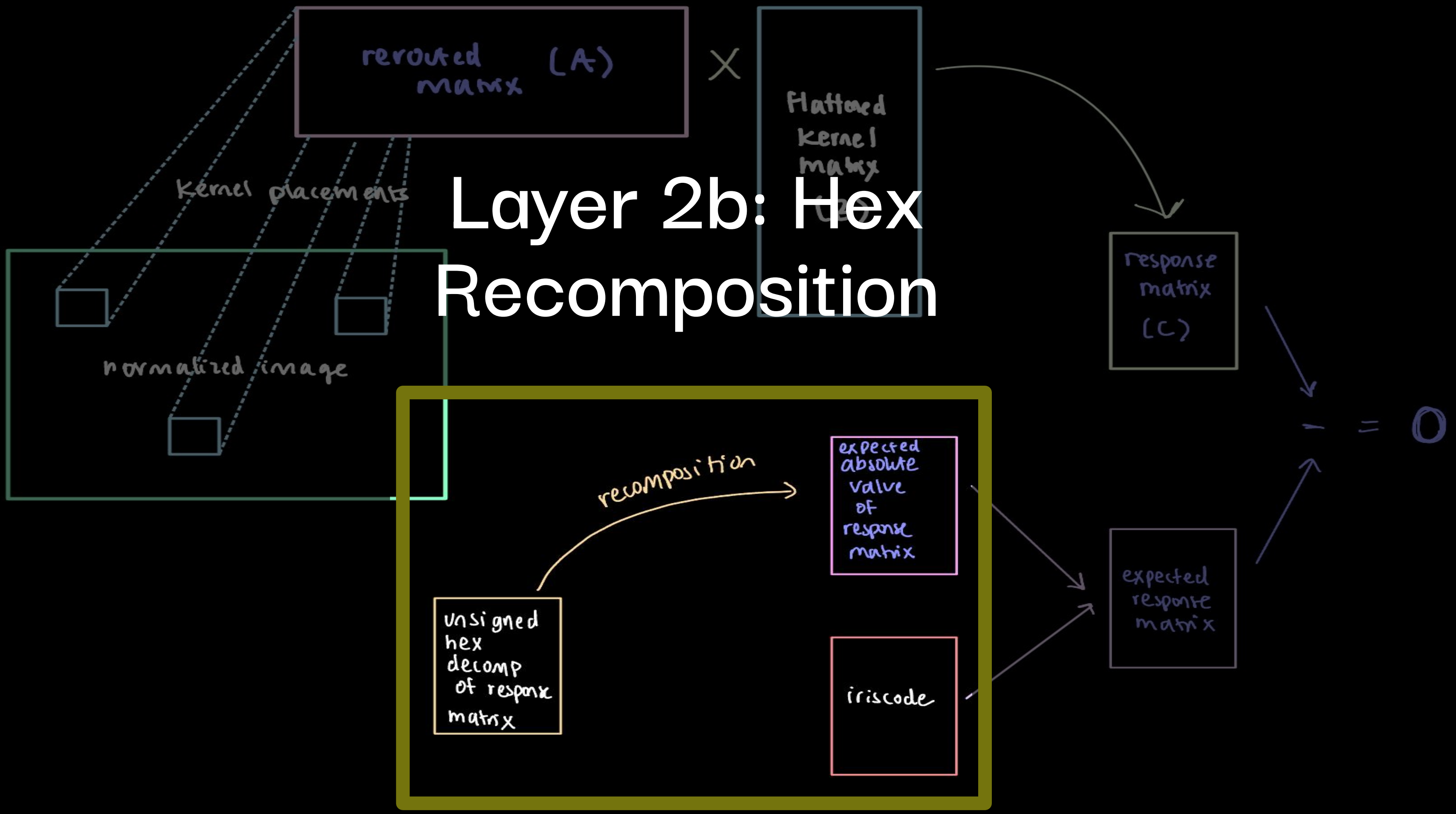
Layer 2a: Hex Check

How do we guarantee that we indeed have unsigned decompositions of the response matrix?

$$d(d-1)(d-2)\cdot\ldots\cdot(d-15) = 0 \quad \forall \text{ digits } d \text{ in the decomposition}$$

either: $d=0$ $d=1$ $d=2$ ... $d=15$

Layer 2b: Hex Recomposition



Layer 2b: Unsigned Hex Recomposition

If computed correctly, this is the **absolute value** of the values in the response matrix calculated in the matrix multiplication layer.

$$\underset{16^{15}}{\underset{\uparrow}{d_1}} \quad \underset{16^{14}}{\underset{\uparrow}{d_2}} \quad \underset{16^{13}}{\underset{\uparrow}{d_3}} \quad \cdots \quad \underset{16^{0}}{\underset{\uparrow}{d_{16}}} \quad \Bigg\} \quad \sum_{i=1}^{16} 16^{16-i} d_i$$

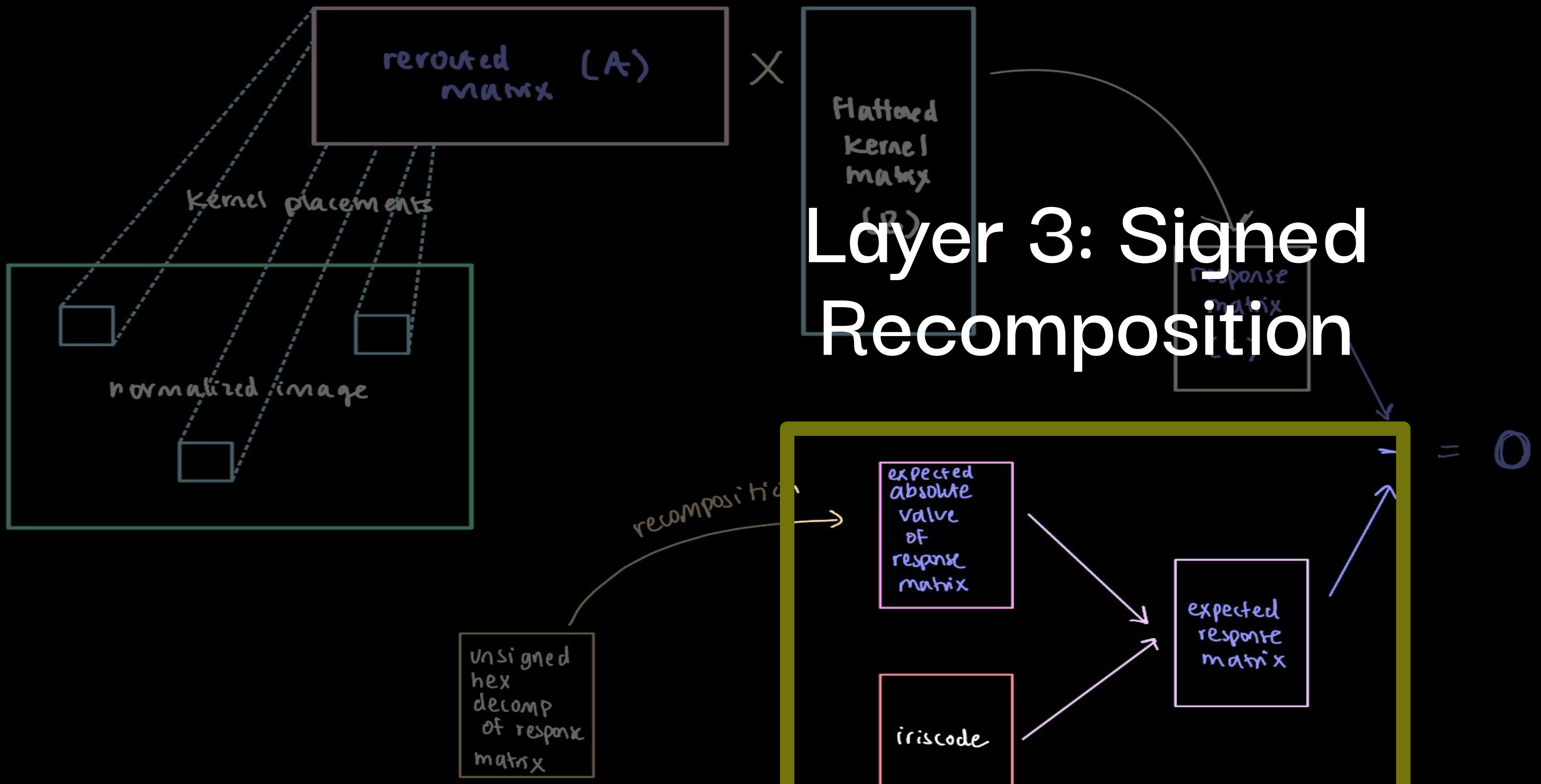
Layer 2b: Unsigned Hex Recomposition

If computed correctly, this is the **absolute value** of the values in the response matrix calculated in the matrix multiplication layer.

$$\begin{array}{ccccc} d_1 & d_2 & d_3 & \cdots & \boxed{d_{16}} \\ \uparrow & \uparrow & \uparrow & & \uparrow \\ 16^{15} & 16^{14} & 16^{13} & & 16^{0} \end{array}$$

$$\sum_{i=1}^{\boxed{16}} 16^{16-i} d_i$$

Layer 3: Signed Recomposition



Layer 3: Signed Recomposition

Use the Iriscode to determine the “signed” value of the recomposition

$b_s(i,j)$ = iriscode bit at position (i,j)

$h_{(i,j)}$ = unsigned hex recomposition at (i,j)

Signed recomposition:

$$(1 - b_{s,(i,j)})(-h_{(i,j)}) + b_{s,(i,j)} h_{(i,j)}$$

$$= (2b_{s,(i,j)} - 1) h_{(i,j)}$$

Layer 3: Signed Recomposition

Use the Iriscode to determine the “signed” value of the recomposition

$b_s(i,j)$ = iriscode bit at position (i,j)

$h_{(i,j)}$ = unsigned hex recomposition at (i,j)

Signed recomposition:

$$\boxed{(1 - b_{s,(i,j)})(-h_{(i,j)})} + b_{s,(i,j)} h_{(i,j)}$$

$$= (2b_{s,(i,j)} - 1) h_{(i,j)}$$

Layer 3: Signed Recomposition

Use the Iriscode to determine the “signed” value of the recomposition

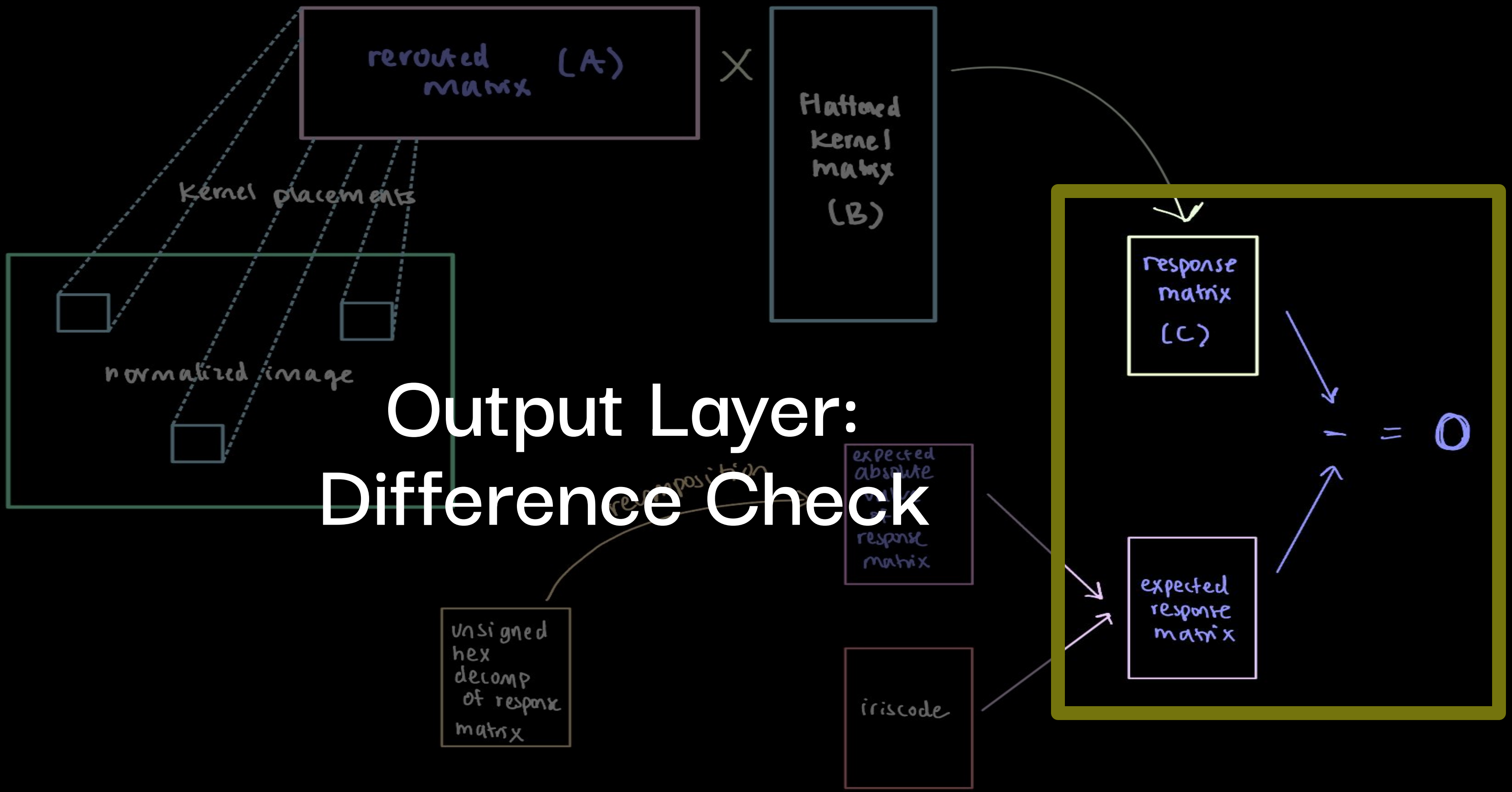
$b_s(i,j)$ = iriscode bit at position (i,j)

$h_{(i,j)}$ = unsigned hex recomposition at (i,j)

Signed recomposition:

$$(1 - b_{s,(i,j)})(-h_{(i,j)}) + \boxed{b_{s,(i,j)} h_{(i,j)}}$$

$$= (2b_{s,(i,j)} - 1) h_{(i,j)}$$



Output Layer: Difference Check

Demo

Benchmarking

Technical constraints

- From discussions with WC.
- Must be capable of running on a low-end smartphone. So:
- Use <4GB of RAM!
- Don’t flatten their battery by computing forever.
- Assume amount of available storage would suffice for just 2 or 3 photos.

Benchmarks

	Our v0	Generic (Halo2 based)
Prover time	19s	6m38s
Proof size	1.9MB	400KB
Prover key size	0	8GB
Prover peak memory	2.6GB	4.1GB
Verifier time	1s	1.5s

*all benchmarks done on a Macbook Air M2

Next steps

Q1 2024

- Mobile ready version.
- Incorporation of new data format (stride is a regular power of two).
- Image in zero-knowledge.
- Face authentication?
- (cue @dangerdan milestone forecasting)

Extension: image authentication

- How to be certain that the user hasn't altered their iris image?
- Proving it produces the old iris code under the old model is insufficient.
- Demonstrated using adversarial ML (see appendices).

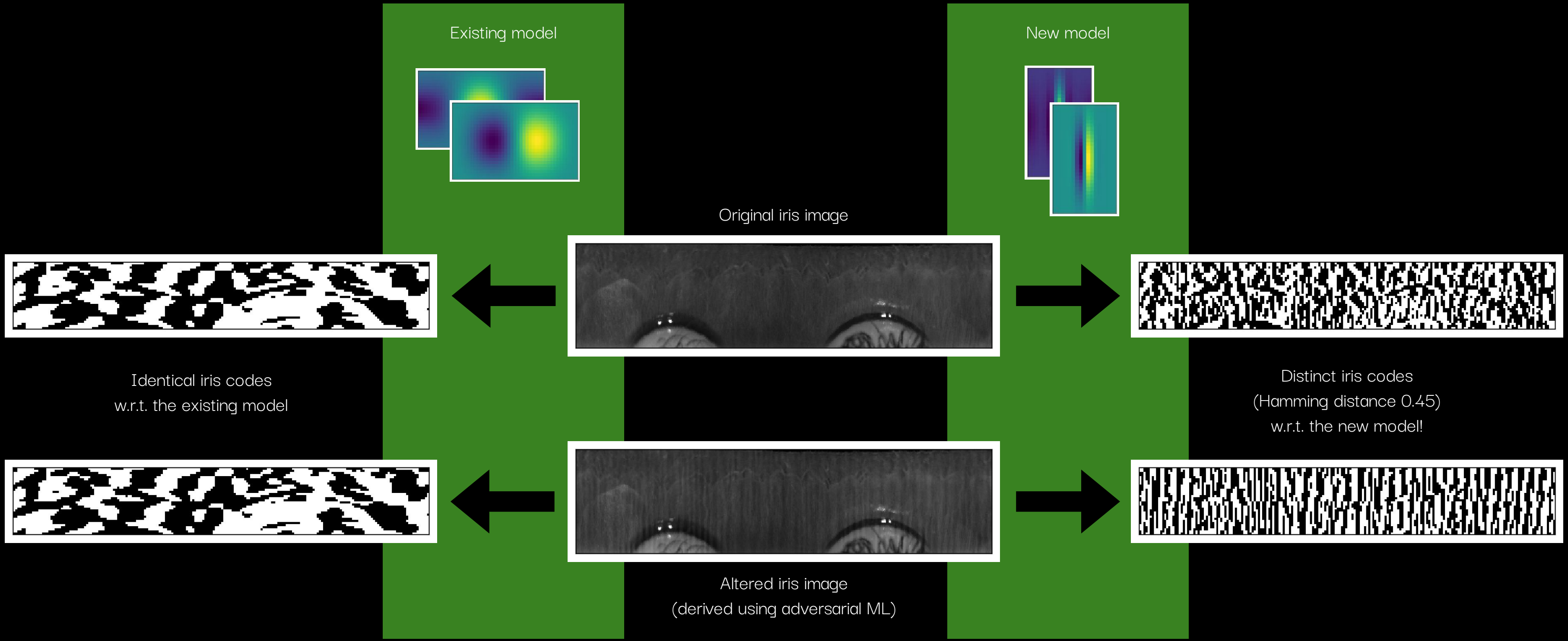
Proposed solutions (c.f. separately sent proposal):

- Verifying the image signature (to be precise: the Merkle tree signature) (computationally very expensive).
- Having the Orb sign a polycommit of the normalized image and mask.

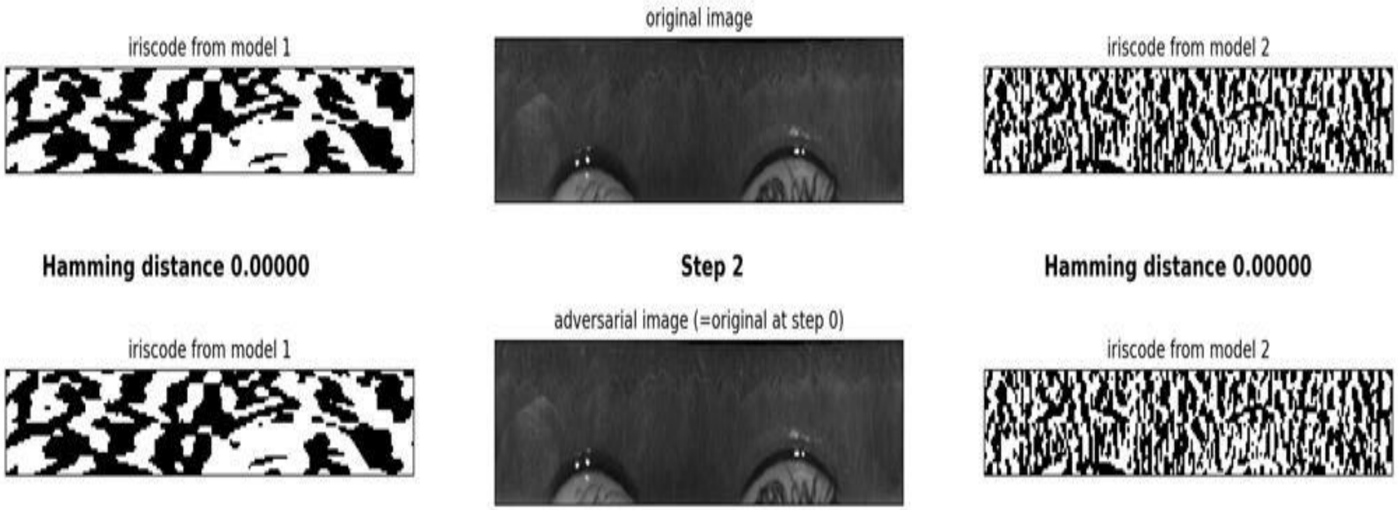
Thank you!

Appendices

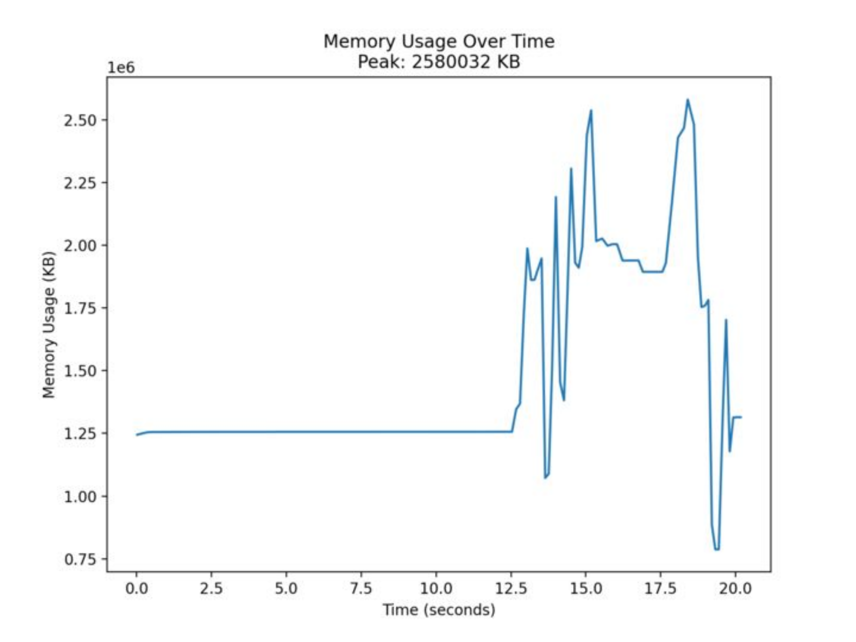
The adversarial image attack



* The currently the WC model uses four real-valued filters, not two. We haven't test this case yet.


original image
iriscode from model 1
iriscode from model 2
Hamming distance 0.00000
Step 2
Hamming distance 0.00000
adversarial image (=original at step 0)
iriscode from model 1
iriscode from model 2

Memory usage: our v0



Memory usage: generic (Halo2-based)