Thresholdizing DSA, Schnorr, EdDSA, HashEdDSA,….

Nigel Smart
KU Leuven

Thresholdizing Signatures

Why would you want to thresholdize a signature?

- You want better key control for high value applications
- You want quorum approval
- ….

Most applications allow you to pick which threshold signature you want to use.

In which case the choice is obvious…

Schnorr, Schnorr, Schnorr and then Schnorr

Thresholdizing Signatures

But sometimes you have a legacy, or regulatory, reason to use a specific signature scheme

Sometimes you want to thresholdize to secure a library

- Think like an Unbound vHSM application

In the latter case you also want to validate the algorithm is the same as the non-thresholdized one.

Lets see these issues for different (elliptic curve based) signature schemes…

DSA

Key Gen: $Q = [x]\, P$

Sign: $r = f(\, [k]\, P\,)$

$s = (H(m) + x\, r)/k \bmod q$

Lot of work recently on new threshold implementations

- Complex due to the k inversion in signing

No “nice” security proof (DSA is an old algorithm, not surprising)

As long as the threshold k is random, threshold code is equivalent to non-threshold code (if correct)

If k is not random loads of nonce-attacks over the last 20 years.

Schnorr

Key Gen: $Q = [x]\, P$

Sign: $r = f([k]\, P)$

$s = (k - x * H(m \mid r)) \bmod q$

Almost trivial to produce actively secure threshold version

Nice security proof (forking Lemma)

As long as the threshold k is random, threshold code is equivalent to non-threshold code (if correct)

If k is not random loads of nonce-attacks over the last 20 years.

EdDSA [Slightly modified to show similarity]

Key Gen: $Q = [x]\, P$

Sign: $k = H'(x \,||\, M)$

$r = f(\, [k]\, P\,)$

$s = (k - x * H(m \mid r)) \bmod q$

Almost trivial to produce actively secure threshold version (its Schnorr)

Nice security proof (forking Lemma, its Schnorr)

As k is deterministic if implemented correctly you cannot have nonce-attacks

From a code-audit point of view should a threshold version be functionally equivalent to a non-threshold version?

- The problem would be evaluating H’ to get a thresholded k.

HashEdDSA [Slightly modified to show similarity]

Key Gen: $Q = [x] P$

Sign: $k = H'(x \,||\, H''(M))$

$r = f([k] P)$

$s = (k - x * H(m \,|\, r)) \bmod q$

Almost trivial to produce actively secure threshold version (its Schnorr)

Nice security proof (forking Lemma, its Schnorr)

As k is deterministic if implemented correctly you cannot have nonce-attacks

From a code-audit point of view should a threshold version be functionally equivalent to a non-threshold version?

- The problem would be evaluating H to get a thresholded k.

The Non-Problem With H’ in EdDSA

H’ is introduced to avoid the bad nonce attacks

The verifier cannot check whether you generate k in this way

So why bother trying to emulate the EdDSA standard exactly?

- Get the parties to generate the shared k in a way which is secure
- Ensure that as long as one party has enough entropy the k is protected against nonce-attacks
- No one will notice in any way

The Problem With H’ in EdDSA

This is fine in practice (or a theoretical version of practice) but this is not the same algorithm

A code auditor/testing lab would never sign off you had implemented a threshold version of EdDSA

So a true thresholdized implementation which could be audited would need to generate k between the parties

This would seem to imply a need for a MPC-like implementation of H’

But H’ could be on a very long message $k = H'(x \,||\, M)$

The Problem With H’ (but not H’’) in HashEdDSA

In HashEdDSA things can be slightly simpler…

$$k = H'(x \,||\, H''(M)\,)$$

H’’ is applied to a public message

- So H’’ can be applied in the clear
- x is kind of small
- So H’ is really on a small input, so could be done in a secure manner

Summary

	Nice Security Proof	Resistance to Nonce Attacks	Trivial to Thresholdize (no cheating)	Trivial to Thresholdize (with cheating)
DSA	👎	👎	👎	👎
Schnorr	👍	👎	👍	👍
EdDSA	👍	👍	👎	👍
HashEdDSA	👍	👍	👎	👍

Summary

	Nice Security Proof	Resistance to Nonce Attacks	Trivial to Thresholdize (no cheating)	Trivial to Thresholdize (with cheating)
DSA	👎	👎	👎	👎
Schnorr	👍	👎	👍	👍
EdDSA	👍	👍	👎	👍
HashEdDSA	👍	👍	👎/👍	👍

Thresholdizing HashEdDSA

Key Gen: $Q = [x]\, P$

Sign: $k = H'(x \,||\, H''(M)\,)$

$r = f(\,[k]\, P\,)$

$s = (k - x * H(m \mid r)) \bmod q$

Step 1: Notice if you secret share x and k in some actively secure LSSS based generic MPC over GF(q) then computation of r and s is trivial

Indeed active security of the generic MPC implies you don't need the ZKPoKs needs for traditional Schnorr threshold signatures

Thresholdizing HashEdDSA

$$k = H'(x \,||\, H''(M)\,)$$

Step 2: Compute H''(M) in the clear.

Step 3: Use the generic MPC system to compute H'

- And this is where the fun starts….
- In the standard H' is SHA512 or SHAKE256

Note: Techniques which follow will also work for EdDSA, but much less efficient if M is very long

The Function H’

So the MPC-ified H’ will take shared input in GF(q), i.e. x

It takes some public `bits’ from H(m)

It applies a bit-oriented hash function SHA512 or SHAKE256

Takes the bit outputs in the MPC domain and maps them to a shared output value k in GF(q)

The Tricky Bit

SHA512 and SHAKE256 are more efficient when implemented via Garbled-Circuit protocols (e.g. Yao or HSS).

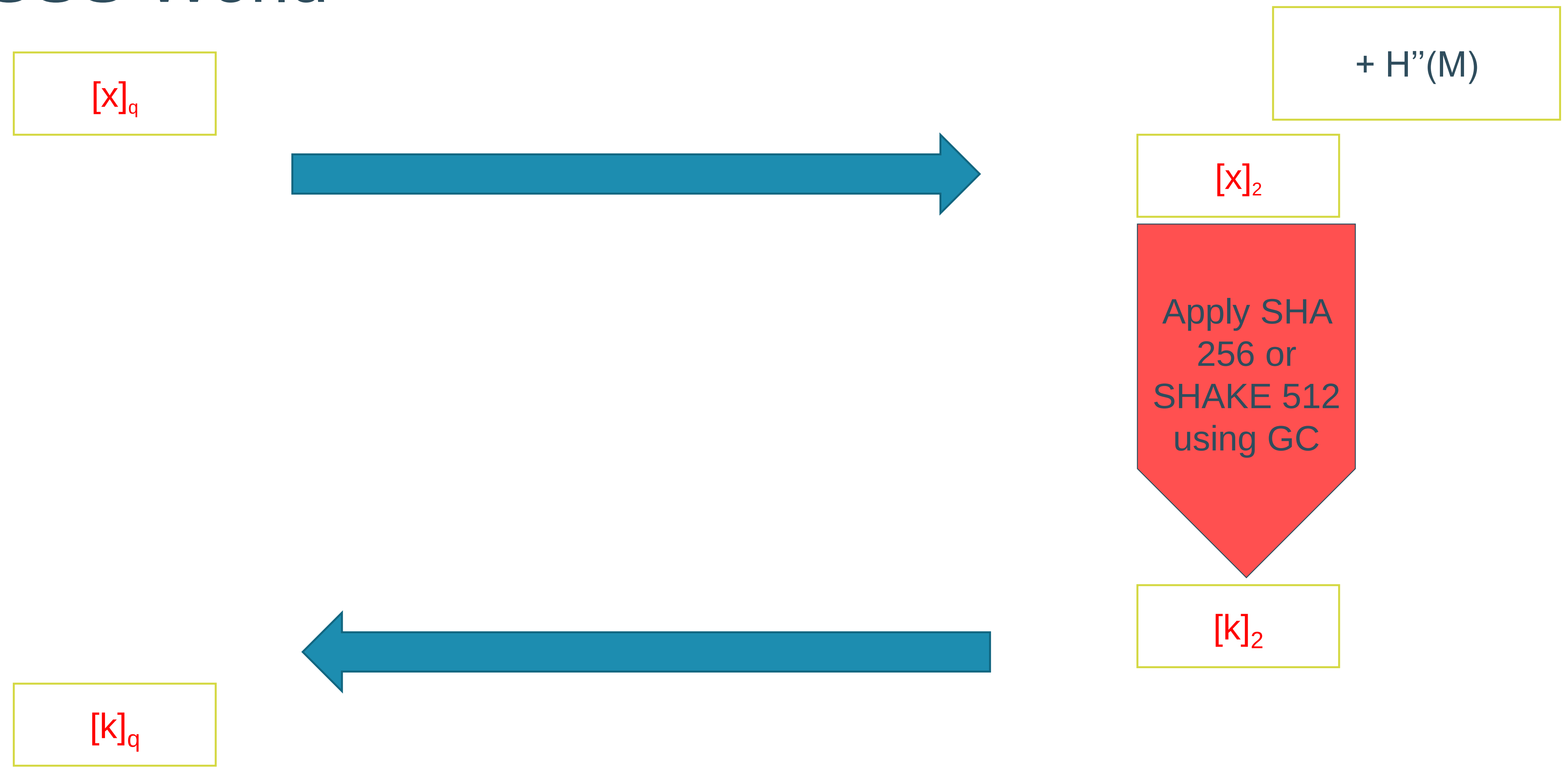
- Note only makes sense in full threshold setting given current MPC technology

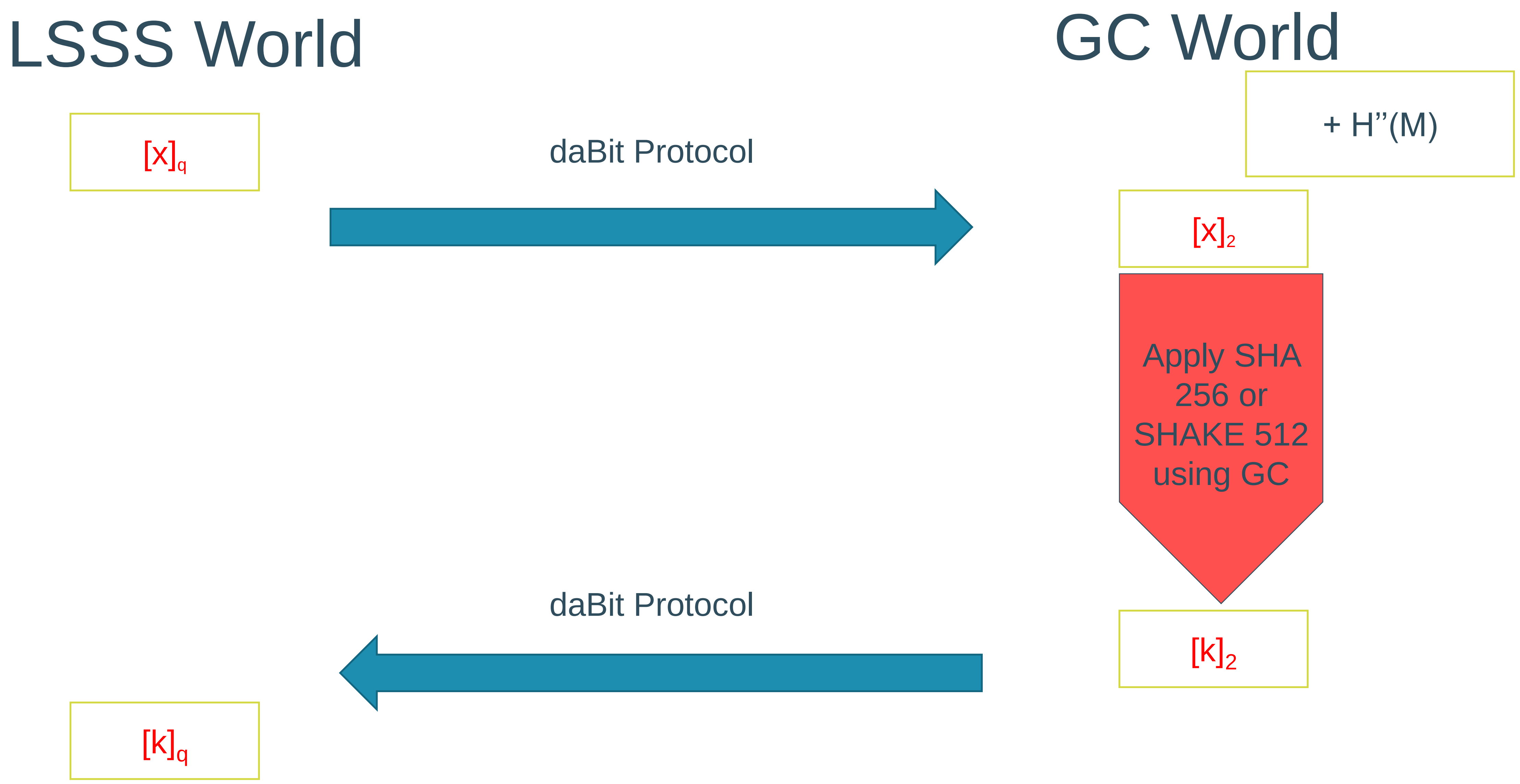
x and k live mod q, so best implemented using LSSS based protocols (e.g. SPDZ etc

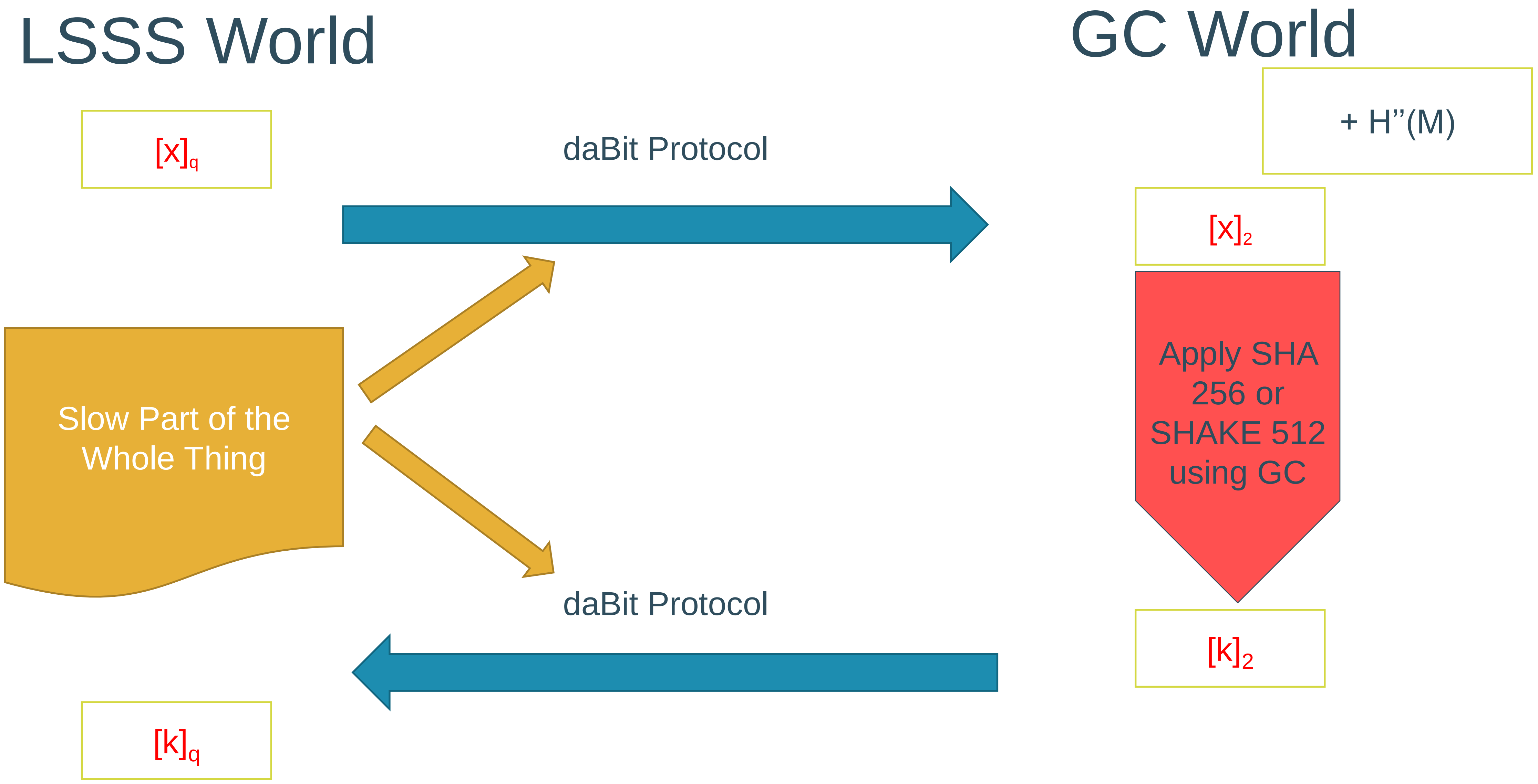
So we need to translate between the LSSS world, the GC world, and then back to the LSSS world

LSSS World

GC World

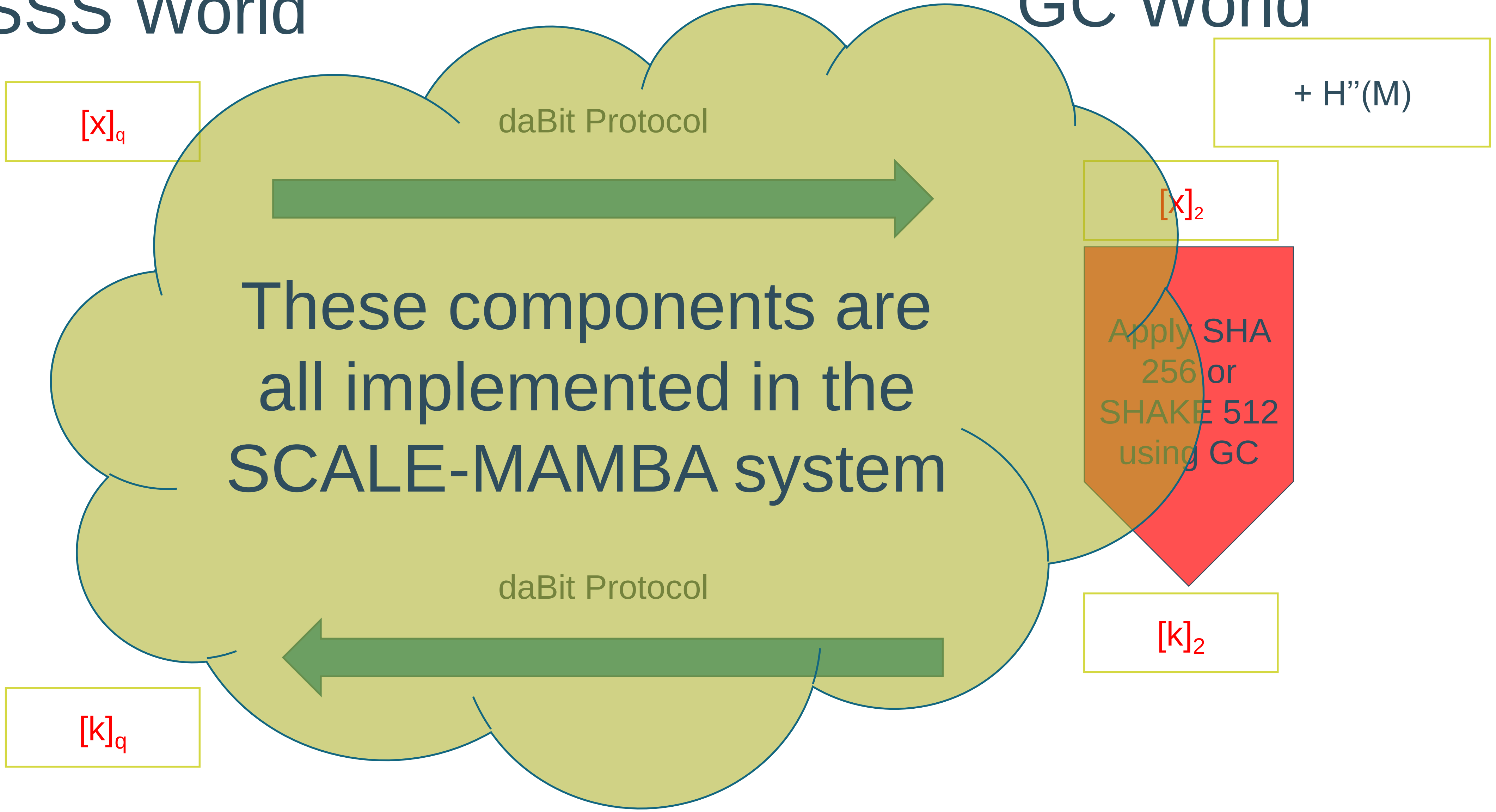



LSSS World
GC World
+ H''(M)
$[x]_q$
daBit Protocol
$[x]_2$
Apply SHA 256 or SHAKE 512 using GC
daBit Protocol
$[k]_2$
$[k]_q$


LSSS World
GC World
+ H''(M)
$[x]_q$
daBit Protocol
$[x]_2$
Slow Part of the Whole Thing
Apply SHA 256 or SHAKE 512 using GC
daBit Protocol
$[k]_2$
$[k]_q$

LSSS World

GC World



These components are all implemented in the SCALE-MAMBA system

Rescue to the Rescue

The problem is we needed to move from the LSSS world to the GC world as the hash function was not friendly to the LSSS world

Fix: Pick a different hash function

- Luckily for use in STARKs etc people have designed efficient hash functions which work natively for data mod q
- Best in class is Rescue

Replacing SHA512/SHAKE256 for H' with a Rescue based implementation makes a big difference…

Run Times

	HashEd25519 + SHA512	HashEd25519 + Rescue	HashEd448 + SHAKE256	HashEd448 + Rescue
Shamir(3,1)	1406 ms	7 ms	1887 ms	14 ms
Shamir(4,1)	1792 ms	9 ms	2515 ms	14 ms
Shamir(5,1)	2190 ms	11 ms	2925 ms	15 ms
Shamir(5,2)	2195 ms	15 ms	2959 ms	17 ms

Note run times for Ed448 depend on the size of the context field (see the standard for what this is)

We give the runtimes for the smallest context field possible above.

See the paper for the full details: https://eprint.iacr.org/2020/214.pdf

Questions for Standards…

Is functional equivalence with the non-threshold version important?

- Even if it cannot be detected during normal usage?

Should basic primitives be also standardized which are MPC-friendly?

- MPC-friendly block ciphers, hash functions etc
- These are being used increasingly in MPC + ZK + Blockchain applications

When designing basic primitives (e.g. signatures in future) should threshold possibilities be taken into account?

- What does this mean for the ongoing PQC “competition”?

Sorry I am Not Around to Take Questions…..