Applied Cryptography and Network Security

William Garrison
bill@cs.pitt.edu
6311 Sennott Square

Post-Quantum Cryptography

University of Pittsburgh

Overview

Terminology

- … and some background about post-quantum schemes in general

Details of classes of schemes

- Hash-based signature schemes
- Lattice-based cryptography
- Code-based schemes
- Multivariate cryptography

We won't talk much about isogeny-based schemes, but Homework 2 includes an article about one in particular

First, settling on the name of this category

I will try to stick with post-quantum cryptography, but alternatives include:

- quantum-resistant cryptography
- quantum-safe cryptography

"Post-quantum" is sometimes misleading

- They are not meant to run on quantum computers
- They are not useful only after large quantum computers exist
- We can use them now…
  - (well, once we agree on them)
- … on conventional computers…
- … and we should migrate ASAP, if we think large quantum computers are coming anytime soon

Reminder: If there were large quantum computers today…

Our current public key algorithms would be broken by Shor's algorithm

- e.g., RSA, Diffie-Hellman, DSA, ECDH, ECDSA…

But our hashes and secret key cryptography (SHA2, SHA3, AES) are OK

- Other than Grover's algorithm
- Doubling key and hash sizes is sufficient (and, in most cases, overkill) to defend against Grover's algorithm

Some general information about post-quantum schemes

Most post-quantum schemes are signature-only or encryption-only

- RSA has some interesting properties that allow it to be used for both
- (with different padding)

Formally, some encryption schemes are really "key encapsulation mechanisms"

- These are more like Diffie–Hellman, where a key is derived from an exchange
- This distinction is relatively minor
- e.g., sender chooses M and sends $h(M)$ as shared secret

NIST is working to standardize post-quantum schemes as they have AES, the SHA series

- Soliciting proposals rather than proposing schemes
- Holds conferences for others to analyze proposals

Proposed schemes should be parameterizable to meet any of a set of security levels

NIST criteria for security levels:

1. At least as hard as key search against 128-bit block cipher
2. At least as hard as collision search against 256-bit hash
3. At least as hard as key search against a 192-bit block cipher
4. At least as hard as collision search against 384-bit hash
5. At least as hard as key search against a 256-bit block cipher

What are the "bits of security" for each of these?

- 128, 128, 192, 192, 256, respectively
- So why are 1/2 and 3/4 distinguished?
- 2, 4 (resp.) could be considered stronger than 1, 3 (resp.) when considering attacks using Grover's algorithm

The proposed schemes are quite different

As we'll see, they use different mathematical problems

Key sizes, signature sizes, and necessary computation vary widely

- … but are generally quite a bit higher than the traditional schemes we've discussed

There is no single "best" scheme

- They make different trade-offs for the various costs

HASH-BASH SIGNATURE SCHEMES

Hash-based signatures are easy to understand, and are secure if we trust hashing

However, there are some downsides

- Lots of computation to generate signatures
- Signatures are very large
- At least verification is very efficient…

What are some scenarios where schemes like this would be most useful?

- Signed data is large (so signature is a small proportion)
- Verification happens much more often than signing
- For instance, they are suggested for signing software and firmware!

Hash-based signature on a single bit

Let's say you want to sign one bit, sometime in the future, e.g.:

- 0: Candidate A won the election
- 1: Candidate B won the election

Pick random numbers S_0 and S_1, and publish $h(S_0)||h(S_1)$ as the public key

S_0 is a signature on 0, and S_1 is a signature on 1

Anyone can verify the signature efficiently by hashing it and seeing which half of the public key it matches

- Assuming the hash function is preimage resistant, nobody else can forge the signature

Extending to allow for signing a (single) arbitrary-length message

As usual, we will sign $h(M)$ rather than signing M directly

- This is secure assuming the hash is collision resistant
- Let's use (say) a 256-bit hash function
  - 128-bit strength in collision resistance

Choose 512 secrets

- $S_{1,0}, S_{1,1}, S_{2,0}, S_{2,1}, S_{3,0}, S_{3,1}, \ldots, S_{256,0}, S_{256,1}$

Publish the 512 hashes as a public key

- $h(S_{1,0}), h(S_{1,1}), h(S_{2,0}), h(S_{2,1}), \ldots, h(S_{256,0}), h(S_{256,1})$

Our signature is a set of 256 secrets out of the 512

- If bit n is 0, include $S_{n,0}$; if bit n is 1, include $S_{n,1}$
- Verify by hashing each secret and seeing which hash it matches

Sizes for this simple hash scheme

Public key: 512 hashes = 16 KiB
Signature: 256 secrets = 8 KiB

Can we do better?

- Public key: $h\left(h(S_{1,0}), h(S_{1,1}), \ldots, h(S_{256,0}), h(S_{256,1})\right)$
- To sign, for each bit i:
  - If 0, reveal $S_{i,0}$ and $h(S_{i,1})$
  - If 1, reveal $S_{i,1}$ and $h(S_{i,0})$
- Verifier hashes each $S_{i,j}$, then hashes all the hashes together to confirm a match with the public key
- Public key: 1 hash = 32 B
- Signature: 256 secrets + 256 hashes = 16 KiB

Note: We can always reduce the public key to a single hash at the expense of a larger signature

A few other optimizations worth mentioning

Use one hash per bit rather than two

- $H_n = h(S_n)$ for each of the N bits
- Signature includes S_n if bit n is 1, H_n if 0
- But then anyone could easily modify by removing 1s
- So also sign the total number of 0 bits
  - e.g., use 8 more bits to represent 0–255 zero bits
  - Removing any 1s will invalidate
- Say, 256 + 8 hashes rather than 512 (16,384 bytes → 8,512 bytes)

Use one hash per nibble (4 bits) instead of per bit

- If nth nibble is value i, release $h^{15-i}(S_n)$
- Also sign the sum of the nibbles: 256-bit value needs 64 nibbles, so the sum can fit in three more nibbles
- 64 + 3 hashes (8,512 bytes → 2,144 bytes)

Extending to a scheme where we can sign multiple messages

Idea: Combine multiple one-time public keys into one "super public key"

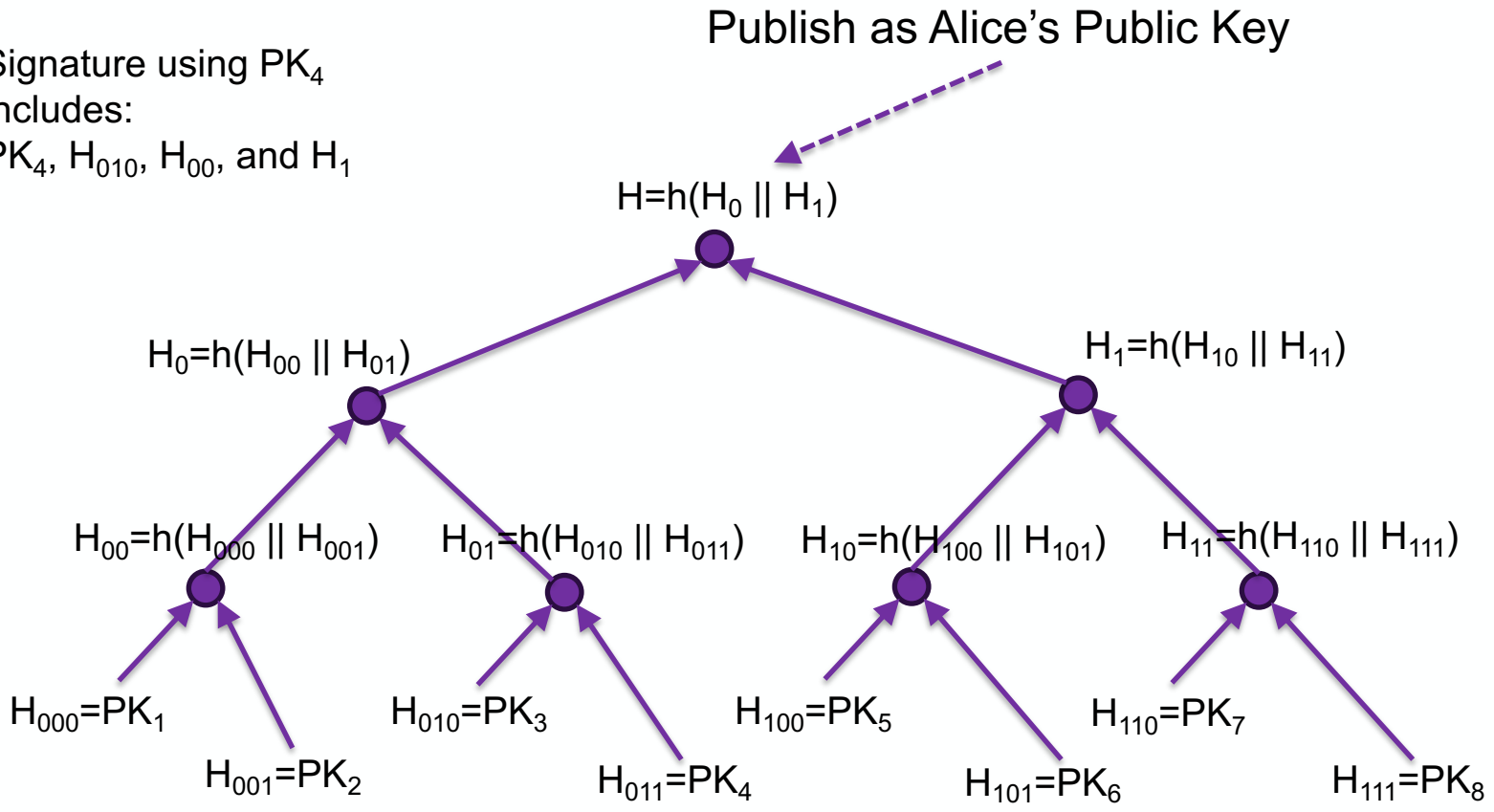
- To sign 10 messages, 320 B public key 👌
- To sign 1 M messages, 32 MB public key 🙅

Remember Merkle trees?

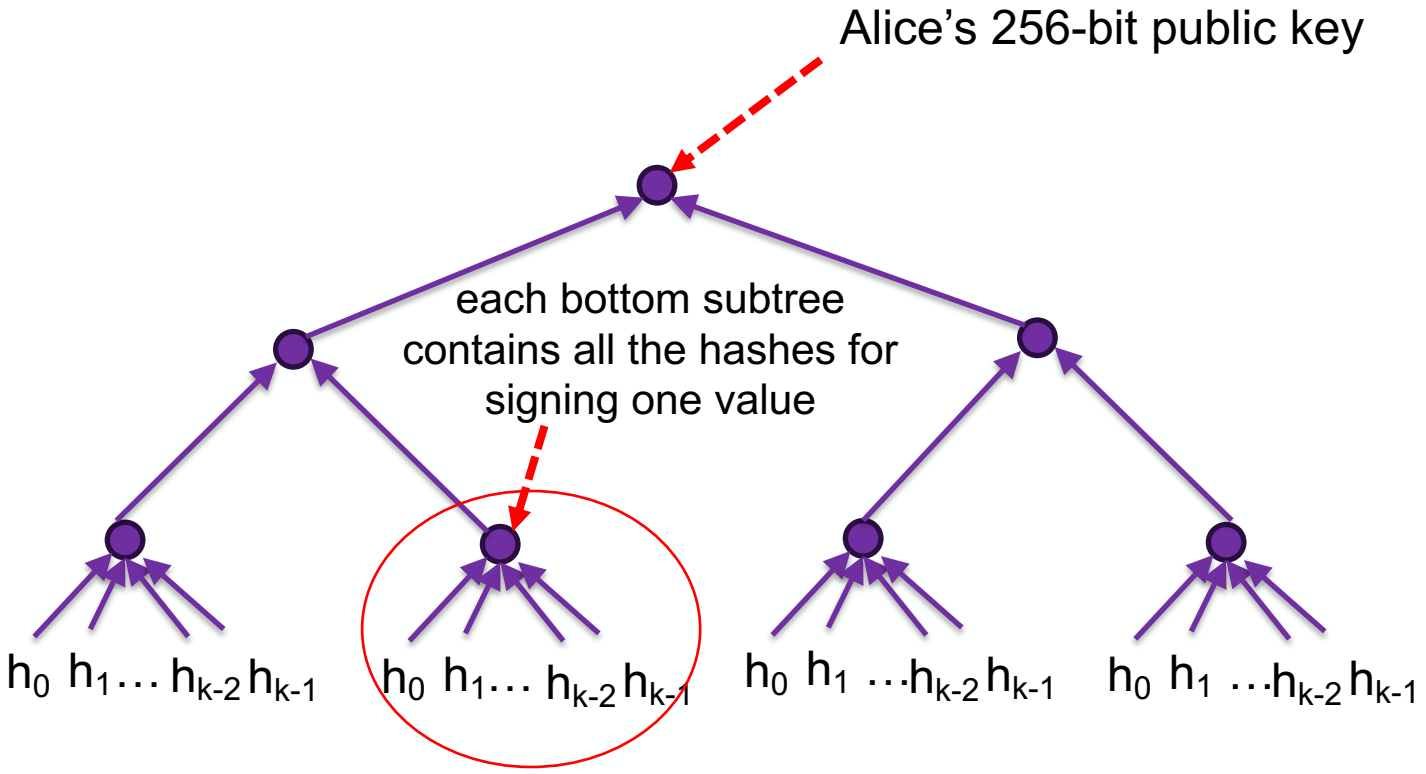
- With a Merkle tree, one hash can represent 2^N hashes
- Any one of the 2^N can be shown to be on the list by showing N extra hash values
  - Essentially, reveal the siblings along the path to demonstrate how each level hashes

Merkle tree for 8 use-once (binary only) public keys

Signature using PK_4 includes:
PK_4, H_{010}, H_{00}, and H_1



Merkle tree for 8 use-once k-bit public keys



Signing Many Messages

If we knew in advance the number of messages we'd ever need to sign, we could precompute a sufficiently large Merkle tree

Signature grows by 16 bytes (1 hash) to double the number of possible signatures

But, even with some efficiency improvements, 1 million signatures require > 1 billion hashes to create the keypair

- … and 2 million hashes must be stored in a private key or recomputed for each signature

We can trade space for runtime by recomputing the entire tree pseudo-randomly from a seed (now the private key)

Stateless hash schemes

The schemes so far all require the signer to count signatures

- This can be problematic given any parallelism, errors

Brute-force stateless hashing scheme

- Pseudo-random tree large enough for 2^{256} signatures
  - All possible 256-bit values
- Use treelet k to sign value k
- Intractably large tree!

Up to 2^n stateless scheme

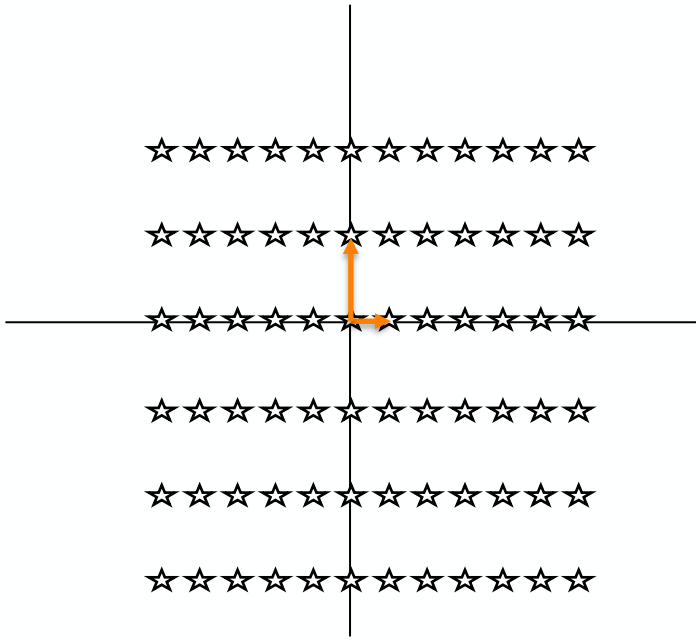
- Pseudo-random tree large enough for 2^{2n+30} signatures
- To sign, pick a signing key randomly
- Given less than 2^n signatures, key reuse is 1 in a billion
- SPHINCS+ uses this (and some other optimization tricks)

LATTICE-BASED CRYPTOGRAPHY

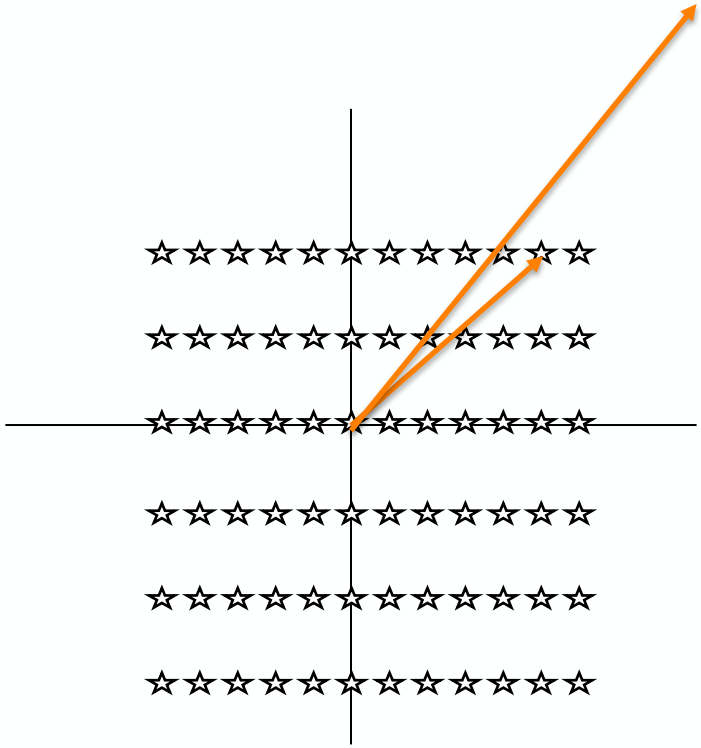
What is a “lattice”?

A lattice is a set of points generated from a “basis” of vectors

- Any integer linear combination of the basis vectors is a point in the lattice



Lattice with basis (0,3), (1,0)



Alt. basis (5,6), (12,15)

The same lattice can be formed with different bases

A basis is "good" if the coefficients are small

- … and "bad" if the coefficients are large

Problem: Find a lattice point that is nearby to a given point in n-dimensional space

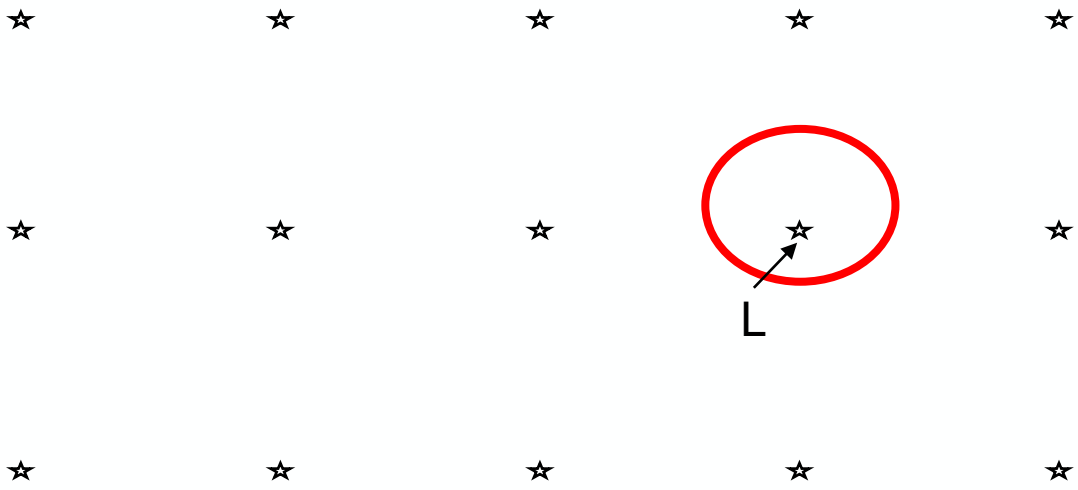
- This is known to be easy if you know a good basis
- … and thought to be hard if you only know a bad basis

Main idea: A good n-dimensional basis is a private key, and a bad basis is a public key

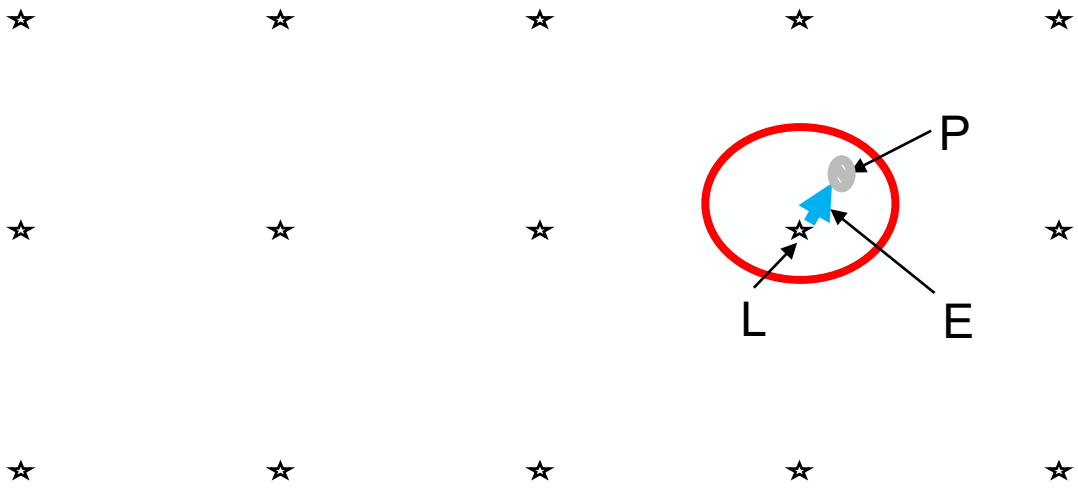
- A bad basis can be created from a good basis

Bob sending a secret value to Alice

Bob uses Alice's public key to choose a random lattice point, L



Bob picks random small “error vector” E
Adds E to L to get non-lattice point P

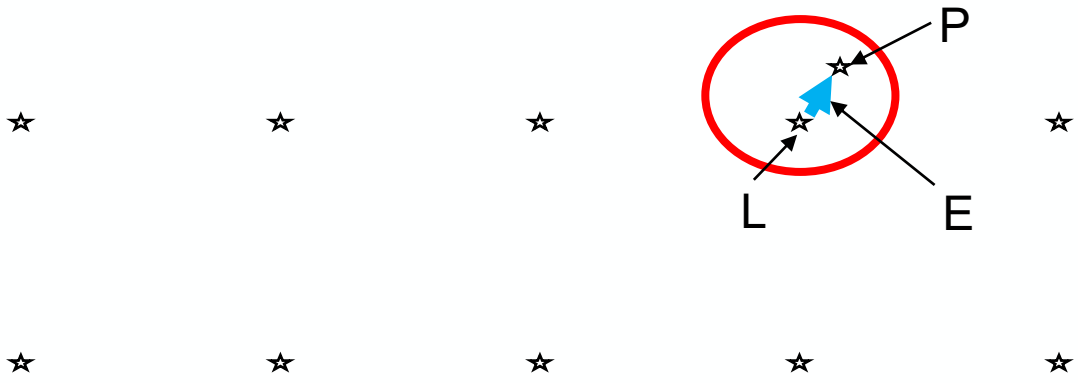


… then, sends P to Alice

Bob sends P to Alice

Alice uses her good basis to calculate nearest lattice point, L

- Subtracts L from P to get E
- Secret is hash(E)



Here, the secret is encoded as an error vector

Lattice-based signatures

Big insight: Interpret the hash value as a point in n-dimensional space

To sign a value, find a lattice point close to it

- This is easy, because you know the good basis
- You can describe the point based on the bad basis
- A verifier can easily tell that the lattice point is close to the "signed" point given this evidence
- As mentioned, it is thought to be hard to find close lattice points to arbitrary points using only a bad basis

Here, the signed value is encoded as a non-lattice point

Learning with Errors

LWE is a lattice-based problem used in public-key primitives

- Reminiscent of Diffie–Hellman, derive shared secret from publicly-exchanged values
- Lots of variants
- Public parameters:
  - a modulus size q (let's say $q = 2^{15}$)
  - an $n \times n$ matrix A, where n might be 500–1000 (let's use 1000)
  - elements of A are integers mod q

Alice and Bob's secret values are two n-element vectors with "small" coefficients (e.g., -2, -1, 0, 1, or 2)

- Maybe from a non-uniform distribution, such as binomial

A simple, but insecure version of LWE

An $n \times n$ matrix A, from finite field (mod q), is public

- Alice chooses secret n-element row vector r
- Bob chooses secret n-element column vector s
  - Both r and s have small coefficients
- Alice sends $r * A$
- Bob sends $A * s$
  - Both use matrix multiplication
- Each uses their secret to multiply what they received, to obtain $r * A * s$, a single value

This is insecure, because A is known

- This means an eavesdropper can invert A to get r and/or s

To address the insecurities, both parties add n-element error vectors

Alice chooses e_A, Bob chooses e_B

- Both error vectors also have small coefficients
- Alice adds e_A to $r * A$
- Bob adds e_B to $A * s$
- After multiplying, they each get approximately $r * A * s$
  - Alice will get $r * A * s + r * e_B$
  - Bob will get $r * A * s + e_A * s$

When parameters are chosen carefully, these approximates are "close enough" for Alice and Bob

- Bob can send a value to communicate a bit, starting from his approximate $r * A * s$
- To send a 0, add a small perturbation
- To send a 1, add a large value (close to $q/2, \bmod q$)

What can an eavesdropper see?

The publicly-exchanged values aren't enough for an eavesdropper to get the value…

- … unless they can solve approximate linear equations, which is as hard as lattice problems

Without optimizations, LWE requires a lot of communication

It is most secure to use a different lattice A each time

But A is something like 1000 vectors in 1000 dimensions, at 15 bits each

- 1.78 MB exchange, to setup sending one bit!

Instead, exchange a 256-bit seed which is used to pseudo-randomly (but deterministically) generate A

- The seed is not a secret
- … which means A is not a secret

Sending multiple bits

Worst-case difference between Alice and Bob's approximate $r * A * s$ is close enough that Bob can send multiple bits at a time

For instance, to send 2 bits:

- Add approximately 0 for 00
- Add approximately q/4 for 01
- Add approximately q/2 for 10
- Add approximately 3q/4 for 11

Typical scheme sends 4 bits at a time

Reusing (r, e_A) and (s, e_B) values

Alice can choose, say, 16 pairs, and Bob can choose 16 pairs

Every combination can be used, resulting in 256 different approximate $r * A * s$ numbers, each of which can send a bit (or several bits)

- Each will transmit about 30 KB, then one sends 500 bytes to establish the secret

CODE-BASED SCHEMES

Code-based schemes are based on error-correcting codes

Error-correcting codes are "the other ECC"

- Maybe "the original ECC," since they predate elliptic-curve cryptography by about 35 years (1950 vs. 1985)

ECCs are used when read errors are likely

- Optical media
- QR codes
- Multi-drive RAIDs with parity for failure recovery

ECCs are comprised of two primary functions:

- Convert a k-bit string into an n-bit "codeword"
- Convert an n-bit codeword, with up to t bit errors, into the original k-bit string

The steps in an error-correcting code, in more detail

Invention (create a code)

- Expand an k-bit string M to an n-bit codeword
- If the first k bits are M, and the remaining $n-k$ are parity bits C, the codeword is in systematic form $M|C$
  - (This is not required)

Misfortune

- Some phenomenon that can happen and flip up to t bits of the codeword
  - Often, $t = n - k$

Diagnosis (recovery)

- From the mangled codeword, we usually want to recover M
- In cryptographic schemes, we usually want the error itself

Moderate-density parity check (MDPC) ECC from a non-crypto perspective

Invention: Create a binary generator matrix $\boldsymbol{G}$ of size $k \times n$

- Should be sparse: about 1% of entries are 1
- k and n should be in the thousands
- Treat a binary string as a binary row vector
- Multiplying a k-bit string by $\boldsymbol{G}$ produces an n-bit codeword
- Arithmetic is mod 2

We can make $\boldsymbol{G}$ produce systematic form by making the left portion the identity matrix $\boldsymbol{I}$

- The rest of $\boldsymbol{G}$ is random and called $\boldsymbol{Q}$

Example generator matrix

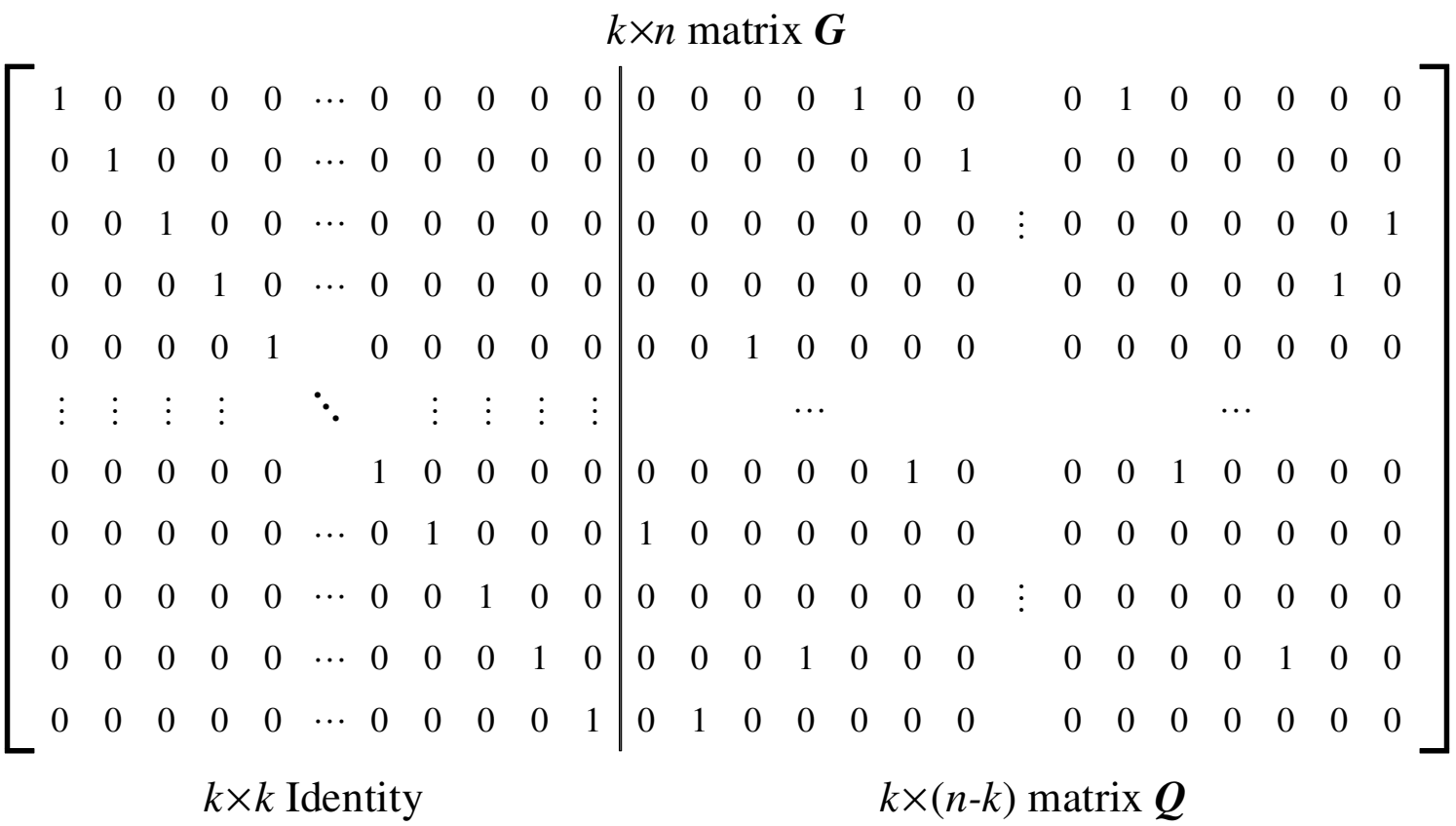


Figure 8-7. Generator Matrix for Systematic Form

Note that Q should be more sparse than it appears here

MDPC recovery step

If we assume systematic form, start by guessing that the first k bits are correctly M

- Use $\boldsymbol{G}$ to calculate the codeword for the candidate M
- XOR the calculated check bits with C, call the result the syndrome
- If there were no errors, the syndrome is the **0** vector

Calculate the likely bit flip: iterative, greedy algorithm

- Find the row of $\boldsymbol{Q}$ that reduces the 1s in the syndrome the most
- This is feasible iff $\boldsymbol{Q}$ is sparse
- This process is probabilistic, but can be parameterized to very rarely loop forever or be incorrect

Parity-check matrices

This is a more convenient way to calculate the syndrome

- Multiple by $\boldsymbol{H}$ instead of by $\boldsymbol{G}$ as in the procedure on the previous slide

There are many parity-check matrixes!

- Let's see one way to build them, which will produce sparse $\boldsymbol{H}$
- For a mangled codeword $\boldsymbol{Y}$, $\boldsymbol{YH}$ is the same as $\boldsymbol{EH}$, where $\boldsymbol{E}$ is the error vector
- Sparse parity-check matrix allows efficient recovery of $\boldsymbol{E}$

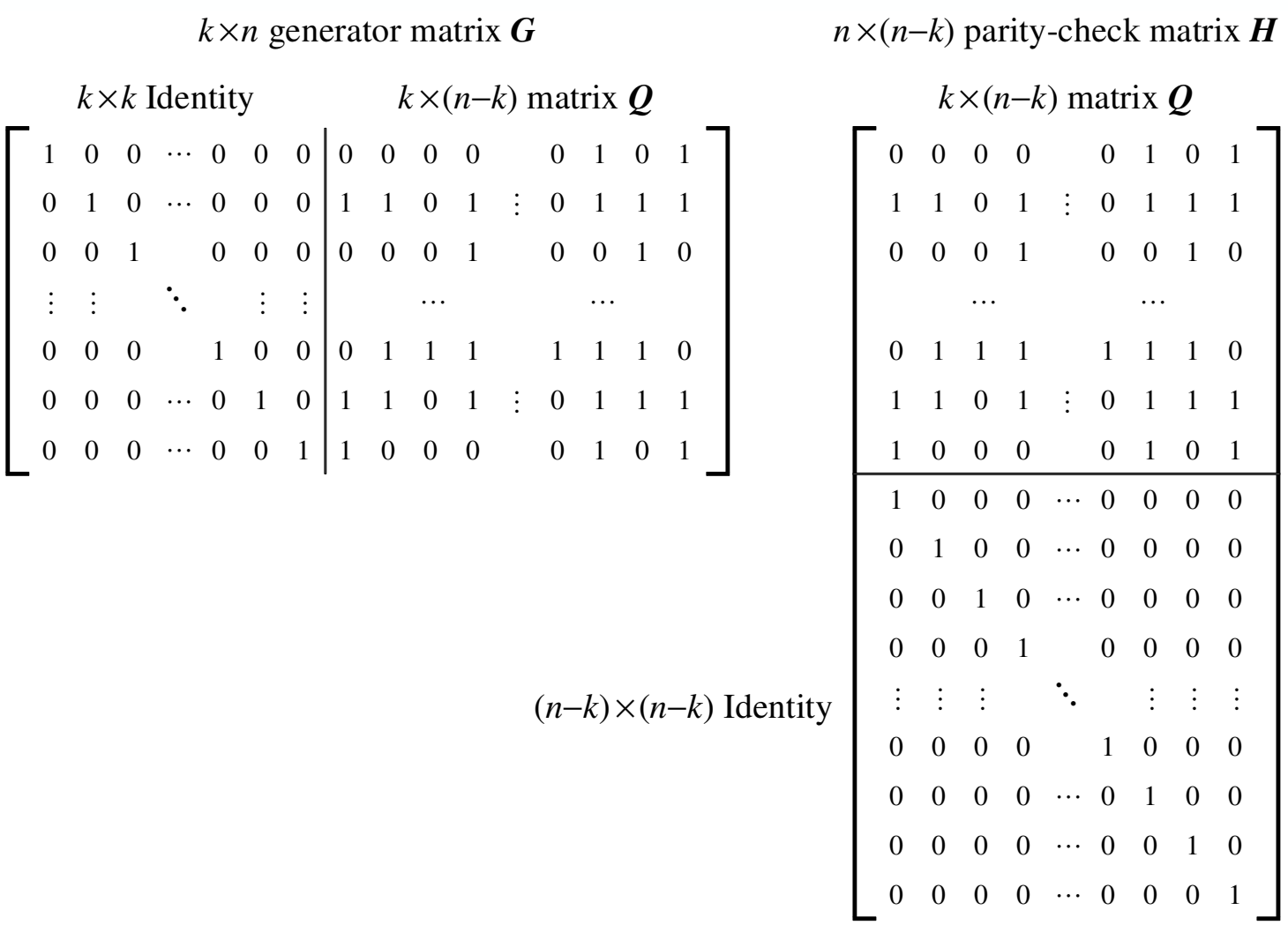


Figure 8-9. Generator Matrix and Parity-check Matrix for Systematic Form

Making MDPC cryptographic

We want someone to be able to make a codeword but not do the error-correcting

- The error will be the secret encrypted to the keypair owner

To accomplish this, we start from a sparse parity-check matrix $\boldsymbol{H}$, which becomes the private key

- Generate a non-sparse generator matrix $\boldsymbol{G}$ from the sparse $\boldsymbol{H}$

To encrypt a secret, Bob first chooses a random vector

- Choose an error vector E with at most t 1 bits
- Use E to mangle the random value
- The exchanged secret will be the hash of E

Note: $\boldsymbol{H}$ will not be systematic, but $\boldsymbol{G}$ will be, for efficiency

Generating a keypair

To generate a keypair, Alice first makes a sparse parity-check matrix $\boldsymbol{H}$

- Again, not in systematic form

Next, convert $\boldsymbol{H}$ into a compatible parity-check matrix…

- … that is dense and in systematic form
- This one is not useful for decoding efficiently, but can be used to create a dense, systematic generator matrix $\boldsymbol{G}$
- $\boldsymbol{G}$ is the public key

Summary of how to use an MDPC keypair

1. Bob acquires Alice's public key $\boldsymbol{G}$
2. Bob chooses an n-bit error vector $\boldsymbol{E}$ with at most t 1 bits
3. Bob multiplies $\boldsymbol{E}$ by $\boldsymbol{G}$ to get n-bit codeword $\boldsymbol{Y}$, then XOR $\boldsymbol{E}$ with $\boldsymbol{Y}$ to get $\boldsymbol{Y}'$, which he sends to Alice
4. Alice multiplies $\boldsymbol{Y}'$ by $\boldsymbol{H}$ to get the syndrome
5. Alice uses the sparse $\boldsymbol{H}$ to calculate $\boldsymbol{E}$

MULTIVARIATE CRYPTOGRAPHY

Multivariate cryptography, main ideas

A system of n linear equations in n variables is easy to solve

We'll extend this idea to quadratics

- i.e., where each term has at most 2 variables multiplied
  - … and we'll have more variables than equations
- e.g., $x^2 + 3xy + 2y^2 + 3y + 4 = 17$
- Best known approaches are not much better than brute force
- For, say, >50 equations over >100 variables, infeasible

To use this for cryptography, a public key is the set of coefficients for the terms in a set of polynomials

- The private key is some trapdoor function that enables solving a system of equations based on the polynomials in one's public key
- For 150 variables, $\binom{n}{2}$ yields 11476 terms per polynomial
- 6-bit coefficients, 50 polynomials yields 3.45 Mbit pubkey

Signing in multivariate cryptography

Get a hash to sign

- Break this value into smaller values, one per equation
- Set each equation to one "chunk"
- A signature is a set of coefficients that solve the equations

Very briefly, the unbalanced oil and vinegar trick

- Label each variable as oil or vinegar (Say, 50 oil and 100 vinegar)
- Generate polynomials where no term has 2 oils multiplied
- These are easy to solve; choose the 100 vinegars arbitrarily, and you have 50 linear equations over 50 variables!
- Set each of the coefficients to a random linear combination of 150 new coefficients
- Substitute the new coefficients into the polynomials
- These new polynomials can be solved by reducing to the constrained case, if you know the combinations!

Conclusions

Post-quantum cryptography uses NP problems with no efficient solution

- Not known to be in P nor BQP

We reviewed a few classes of post-quantum cryptosystems, getting a sense of "what stuff is inside"

- Hash-based
- Lattice-based
- Code-based
- Multivariate

We don't expect one system, or even one class, to "win"

- Different approaches have different trade-offs