Lattice-based cryptography, part 1: simplicity

D. J. Bernstein

University of Illinois at Chicago;
Ruhr University Bochum

2000 Cohen cryptosystem

Public key: vector of integers
$K = (K_1, \ldots, K_N) \in \{-X, \ldots, X\}^N$.

Encryption:

1. Input message $m \in \{0, 1\}$.

2. Generate $r_1, \ldots, r_N \in \{0, 1\}$.
i.e. $r = (r_1, \ldots, r_N) \in \{0, 1\}^N$.

(Cohen says pick "half of the integers in the public key at random": I guess this means $N \in 2\mathbf{Z}$ and $\sum r_i = N/2$.)

3. Compute and send ciphertext
$C = (-1)^m (r_1 K_1 + \cdots + r_N K_N)$.

How can receiver decrypt?

Key generation:

Generate $s \in \{1, \ldots, Y\}$;

$$u_1, \ldots, u_N \in \left\{0, \ldots, \left\lfloor \frac{s-1}{2N} \right\rfloor \right\};$$

$K_i \in (u_i + s\mathbf{Z}) \cap \{-X, \ldots, X\}$.

Decryption:

$m = 0$ if $C \bmod s \le (s-1)/2$;

otherwise $m = 1$.

Why this works:

$K_i \bmod s = u_i \le (s-1)/2N$ so

$$r_1K_1 + \cdots + r_NK_N \bmod s \le \frac{s-1}{2}.$$

(Be careful! What if all $r_i = 0$?)

Let's try this on the computer.

Debian: `apt install sagemath`

Fedora: `dnf install sagemath`

Source: `www.sagemath.org`

Web (use `print(X)` to see `X`):

`sagecell.sagemath.org`

Sage is Python 3

+ many math libraries

+ a few syntax differences:

```
sage: 10^6 # power, not xor
1000000
sage: factor(314159265358979323)
317213509 * 990371647
sage:
```

For integers C, s with $s > 0$, Sage's "C%s" always produces outputs between 0 and $s - 1$.

Matches standard math definition: $C \bmod s = C - \lfloor C/s \rfloor s$.

Warning: Typically $C < 0$ produces $C\%s < 0$ in lower-level languages, so nonzero output leaks input sign.

Warning: For polynomials C, Sage can make the same mistake.

```
sage: N=10
sage: X=2^50
sage: Y=2^20
sage: Y
1048576
sage: s=randrange(1,Y+1)
sage: s
359512
sage: u=[randrange(
....:       (s-1)//(2*N)+1)
....:     for i in range(N)]
sage: u
[14485, 7039, 6945, 15890,
 10493, 17333, 1397, 8656,
 8213, 6370]
```

```
sage: K=[ui+s*randrange(
....:         ceil(-(X+ui)/s),
....:         floor((X-ui)/s)+1)
....:     for ui in u]
sage: K
[870056918917829,
 822006576592695,
 -294765544345815,
 -669275100080982,
 528958455221029,
 426006001074157,
 -641940176080531,
 501543495923784,
 -583064075392587,
 46109390243834]
```

```
sage: [Ki%s for Ki in K]
[14485, 7039, 6945, 15890,
 10493, 17333, 1397, 8656,
 8213, 6370]
sage: u
[14485, 7039, 6945, 15890,
 10493, 17333, 1397, 8656,
 8213, 6370]
sage: sum(K)%s
96821
sage: sum(u)
96821
sage: s//2
179756
sage:
```

```
sage: m=randrange(2)
sage: r=[randrange(2)
....:     for i in range(N)]
sage: C=(-1)^m*sum(r[i]*K[i]
....:    for i in range(N))
sage: C
-202215856043576
sage: C%s
47024
sage: m
0
sage: sum(r[i]*u[i]
....:       for i in range(N))
47024
sage:
```

Some problems with cryptosystem

1. Functionality problem:
System can't encrypt messages that have more than 1 bit.

2. Security problem:
We want cryptosystems to resist "chosen-ciphertext attacks" where attacker can see decryptions of other ciphertexts.

Chosen-ciphertext attack against this system:
Decrypt $-C$. Flip result.

(Works whenever $C \neq 0$.)

2000 Cohen: cryptosystem
fixing both of these problems.

1. Transform 1-bit encryption into multi-bit encryption by encrypting each bit separately. Use new randomness for each bit.

B-bit input message $m = (m_1, \ldots, m_B) \in \{0,1\}^B$.
For each $i \in \{1, \ldots, B\}$:
Generate $r_{i,1}, \ldots, r_{i,N} \in \{0,1\}$.

Ciphertext C:
$(-1)^{m_1}(r_{1,1}K_1 + \cdots + r_{1,N}K_N)$,
$\ldots$,
$(-1)^{m_B}(r_{B,1}K_1 + \cdots + r_{B,N}K_N)$.

2. Derandomize encryption, and reencrypt during decryption.

This is an example of "FO", the 1999 Fujisaki–Okamoto transform.

Derandomization: Generate r as cryptographic hash $H(m)$, using standard hash function H. (Watch out: Is m guessable?)

Decryption with reencryption:
1. Input C'. (Maybe $C' \neq C$.)
2. Decrypt to obtain m'.
3. Recompute $r' = H(m')$.
4. Recompute C'' from m', r'.
5. Abort if $C'' \neq C'$.

Subset-sum attacks

Attacker searches all possibilities for $(r_1, \ldots, r_N)$, checks $r_1 K_1 + \cdots + r_N K_N$ against $\pm C_1$.

This takes 2^N easy operations: e.g. 1024 operations for $N = 10$.

"This finds only one bit m_1."

— This is a problem in some applications. Should design encryption to leak *no* information.

— Also, can easily modify attack to find all bits of message.

Modified attack:
For each $(r_1, \ldots, r_N)$, look up $r_1 K_1 + \cdots + r_N K_N$ in hash table containing $\pm C_1, \pm C_2, \ldots, \pm C_B$.

Multi-target attack:
Apply this not just to B bits in one message, but all bits in all messages sent to this key.

Finding all bits in all messages: total 2^N operations.

Finding 1% of all bits in all messages, huge information leak: total $0.01 \cdot 2^N$ operations.

"We can stop attacks by taking $N = 128$, and changing keys every day, and applying all-or-nothing transform to each message."

— Standard subset-sum attacks take only $2^{N/2}$ operations to find $(r_1, \ldots, r_N) \in \{0, 1\}^N$ with $r_1 K_1 + \cdots + r_N K_N = C$.

Make hash table containing $C - r_{N/2+1} K_{N/2+1} - \cdots - r_N K_N$ for all $(r_{N/2+1}, \ldots, r_N)$.

Look up $r_1 K_1 + \cdots + r_{N/2} K_{N/2}$ in hash table for each $(r_1, \ldots, r_{N/2})$.

These attacks exploit linear structure of problem to convert one target C into many targets.

(Actually have $2B$ targets $\pm C_1, \ldots, \pm C_B$ for one message. Convert into $B^{1/2}2^{N/2}$ targets: total $B^{1/2}2^{N/2}$ operations to find all B bits. Also, maybe have more messages to attack.)

There are even more ways to exploit the linear structure.

1981 Schroeppel–Shamir: $2^{N/2}$ operations, space $2^{N/4}$.

2010 Howgrave-Graham–Joux: claimed $2^{0.311N}$ operations. 2011 May–Meurer correction: $2^{0.337N}$.

2011 Becker–Coron–Joux: $2^{0.291N}$ operations.

2016 Ozerov: $2^{0.287N}$ operations.

2019 Esser–May: claimed $2^{0.255N}$ operations, but withdrew claim.

2020 Bonnetain–Bricout–Schrottenloher–Shen: $2^{0.283N}$.

Quantum attacks: various papers.

Multi-target speedups: probably!

Variants of cryptosystem

2003 Regev: Cohen cryptosystem (without credit), but replace $(-1)^m(r_1K_1 + \cdots + r_NK_N)$ with $m(K_1/2) + r_1K_1 + \cdots + r_NK_N$.

To make this work,
modify keygen to force $K_1 \in 2\mathbf{Z}$ and $(K_1 - u_1)/s \in 1 + 2\mathbf{Z}$.
Also be careful with u_i bounds.

2009 van Dijk–Gentry–Halevi–Vaikuntanathan: $K_i \in 2u_i + s\mathbf{Z}$;
$C = m + r_1K_1 + \cdots + r_NK_N$;
$m = (C \bmod s) \bmod 2$.
Be careful to take $s \in 1 + 2\mathbf{Z}$.

Homomorphic encryption

If u_i/s is small enough then 2009 DGHV system is homomorphic.

Take two ciphertexts:
$C = m + 2\epsilon + sq$,
$C' = m' + 2\epsilon' + sq'$
with small $\epsilon, \epsilon' \in \mathbf{Z}$.

$C + C' = m + m' + 2(\epsilon + \epsilon') + s(q + q')$. This decrypts to $m + m' \bmod 2$ if $\epsilon + \epsilon'$ is small.

$CC' = mm' + 2(\epsilon m' + \epsilon' m + 2\epsilon\epsilon') + s(\cdots)$. This decrypts to mm' if $\epsilon m' + \epsilon' m + 2\epsilon\epsilon'$ is small.

```
sage: N=10
sage: E=2^10
sage: Y=2^50
sage: X=2^80
sage: s=1+2*randrange(Y/4,Y/2)
sage: s
984887308997925
sage: u=[randrange(E)
....:     for i in range(N)]
sage: u
[247, 418, 365, 738, 123, 735,
 772, 209, 673, 47]
sage:
```

```
sage: K=[2*ui+s*randrange(
....:        ceil(-(X+2*ui)/s),
....:        floor((X-2*ui)/s)+1)
....:     for ui in u]
sage: K
[587473338058640662659869,
 -1111539179100720083770339,
 794301459533783434896055,
 68817802108374958901751,
 742362470968200823035396,
 1023345827831539515054795,
 -357168679398558876730006,
 1121421619119964601051443,
 -1109674862276222495587129,
 -235628937785003770523381]
```

```
sage: m=randrange(2)
sage: r=[randrange(2)
....:      for i in range(N)]
sage: C=m+sum(r[i]*K[i]
....:      for i in range(N))
sage: C
20940887487482472100167O3
sage: C%s
2703
sage: (C%s)%2
1
sage: m
1
sage:
```

```
sage: m2=randrange(2)
sage: r2=[randrange(2)
....:      for i in range(N)]
sage: C2=m2+sum(r2[i]*K[i]
....:      for i in range(N))
sage: C2
-5172235373798273727 0129
sage: C2%s
4971
sage: (C2%s)%2
1
sage: m2
1
sage:
```

```
sage: (C+C2)%s
7674
sage: (C*C2)%s
13436613
sage:
```

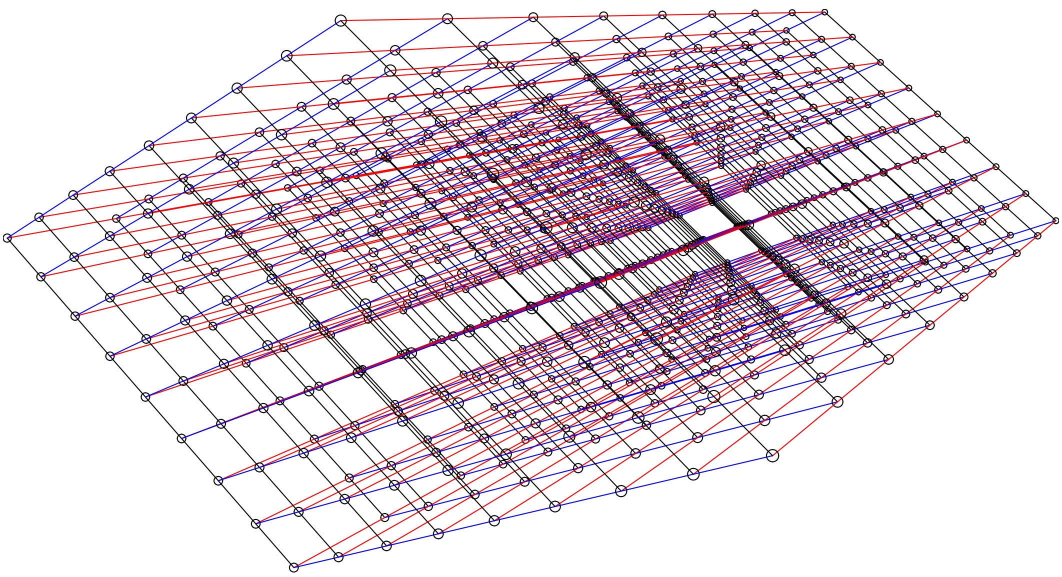
Because C mod s and C' mod s are small enough compared to s, have $C + C'$ mod $s = (C \bmod s) + (C' \bmod s)$ and CC' mod $s = (C \bmod s)(C' \bmod s)$.

Refinements: add more noise to ciphertexts, bootstrap (2009 Gentry) to control noise, etc.

Lattices

This is a lettuce:

This is a lattice:



Lattices, mathematically

Assume that $V_1, \ldots, V_D \in \mathbf{R}^N$
are $\mathbf{R}$-linearly independent,
i.e., $\mathbf{R}V_1 + \cdots + \mathbf{R}V_D =$
$\{r_1V_1 + \cdots + r_DV_D : r_1, \ldots, r_D \in \mathbf{R}\}$
is a D-dimensional vector space.

$\mathbf{Z}V_1 + \cdots + \mathbf{Z}V_D =$
$\{r_1V_1 + \cdots + r_DV_D : r_1, \ldots, r_D \in \mathbf{Z}\}$
is a rank-D length-N **lattice**.

$V_1, \ldots, V_D$
is a **basis** of this lattice.

Short vectors in lattices

Given $V_1, V_2, \ldots, V_D \in \mathbf{Z}^N$,
what is shortest vector
in $L = \mathbf{Z}V_1 + \cdots + \mathbf{Z}V_D$?

0.

"SVP: shortest-vector problem":
What is shortest nonzero vector?

1982 Lenstra–Lenstra–Lovász
(LLL) algorithm runs in poly time,
computes a nonzero vector in L
with length at most $2^{D/2}$ times
length of shortest nonzero vector.
Typically $\approx 1.02^D$ instead of $2^{D/2}$.

Subset-sum lattices

One way to find $(r_1, \ldots, r_N)$ where $C = r_1 K_1 + \cdots + r_N K_N$:

Choose λ. Define
$V_0 = (-C, 0, 0, \ldots, 0)$,
$V_1 = (K_1, \lambda, 0, \ldots, 0)$,
$V_2 = (K_2, 0, \lambda, \ldots, 0)$,
$\ldots$,
$V_N = (K_N, 0, 0, \ldots, \lambda)$.

Define $L = \mathbf{Z}V_0 + \cdots + \mathbf{Z}V_N$.
L contains the short vector
$V_0 + r_1 V_1 + \cdots + r_N V_N =$
$(0, r_1\lambda, \ldots, r_N\lambda)$.

LLL is fast but almost never finds this short vector in L.

1991 Schnorr–Euchner “BKZ” algorithm spends more time than LLL finding shorter vectors in any lattice. Many subsequent time-vs.-shortness improvements.

2012 Schnorr–Shevchenko claim that modern form of BKZ solves subset-sum problems faster than 2011 Becker–Coron–Joux.

Is this true? Open: What’s the exponent of this algorithm?

Lattice attacks on DGHV keys

Recall $K_i = 2u_i + sq_i \approx sq_i$.
Each u_i is small: $u_i < E$.
Note $q_j K_i - q_i K_j = 2q_j u_i - 2q_i u_j$.

Define
$V_1 = (E, K_2, K_3, \ldots, K_N)$;
$V_2 = (0, -K_1, 0, \ldots, 0)$;
$V_3 = (0, 0, -K_1, \ldots, 0)$;
$\ldots$;
$V_N = (0, 0, 0, \ldots, -K_1)$.

Define $L = \mathbf{Z}V_1 + \cdots + \mathbf{Z}V_N$.
L contains $q_1 V_1 + \cdots + q_N V_N =$
$(q_1 E, q_1 K_2 - q_2 K_1, \ldots) =$
$(q_1 E, 2q_1 u_2 - 2q_2 u_1, \ldots)$.

```
sage: V=matrix.identity(N)
sage: V=-K[0]*V
sage: Vtop=copy(K)
sage: Vtop[0]=E
sage: V[0]=Vtop
sage: q0=V.LLL()[0][0]/E
sage: q0
596487875
sage: round(K[0]/q0)
984887308997925
sage: s
984887308997925
sage:
```

```
sage: V[0]
(1024,
 -1111539179100720083770339,
 794301459533783434896055,
 688178021083749589017751,
 742362470968200823035396,
 1023345827831539515054795,
 -3571686793985588767300006,
 1121421619119964601051443,
 -1109674862276222495587129,
 -2356289377850037705233381)
sage: V[1]
(0, -5874733380586406626598 69,
 0, 0, 0, 0, 0, 0, 0, 0)
sage:
```

```
sage: V.LLL()[0]
(610803584000, 1056189937254,
 37030242384, 845898454698,
 -225618319442, 363547143644,
 1100126026284, -313150978512,
 1359463649048, 174256676348)
sage: q=[Ki//s for Ki in K]
sage: q[0]*E
610803584000
sage: q[0]*K[1]-q[1]*K[0]
1056189937254
sage: q[0]*K[9]-q[9]*K[0]
174256676348
sage:
```

2009 DGHV analysis: can choose key sizes where these lattice attacks fail.

2011 Coron–Mandal–Naccache–Tibouchi: reduce key sizes by modifying DGHV. "This shows that fully homomorphic encryption can be implemented with a simple scheme."

e.g. all attacks take $\geq 2^{72}$ cycles with public keys only 802MB.

2012 Chen–Nguyen: faster attack. Need bigger DGHV/CMNT keys.

Big attack surfaces are dangerous

1991 Chaum–van Heijst–Pfitzmann: choose p sensibly; define $C(x, y) = 4^x 9^y \bmod p$ for suitable ranges of x and y.

Simple, beautiful, structured. Very easy security reduction: finding C collision implies computing a discrete logarithm.

Typical exaggerations: C is "provably secure"; C is "cryptographically collision-free"; "security follows from rigorous mathematical proofs".

Security losses in C include
1922 Kraitchik (index calculus);
1986 Coppersmith–Odlyzko–
Schroeppel (NFS predecessor);
1993 Gordon (general DL NFS);
1993 Schirokauer (faster NFS);
1994 Shor (quantum poly time);
many subsequent attack speedups
from people who care about
pre-quantum security.

C is very bad cryptography.
No matter what user's cost limit
is, obtain better security with
"unstructured" compression-
function designs such as BLAKE.

For public-key encryption:
Some mathematical structure seems to be unavoidable, but pursuing simple structures often leads to security disasters.

Pre-quantum example: DH is simpler than ECDH, but DH has suffered many more security losses than ECDH. State-of-the-art DH attacks are very complicated.

2013 Barbulescu–Gaudry–Joux–Thomé: pre-quantum quasi-poly break of small-characteristic DH.

The state-of-the-art attacks against Cohen's cryptosystem are much more complicated than the cryptosystem is. Scary!

Lattice-based cryptosystems are advertised as "algorithmically simple", consisting mainly of "linear operations on vectors". Attacks exploit this structure!

For efficiency, lattice-based cryptosystems usually have features that expand the attack surface even more: e.g., rings and decryption failures.