LWE Estimator Documentation

Release 1.0

Martin R Albrecht

Jul 05, 2018

CHAPTER 1

Module Overview

This Sage module provides functions for estimating the concrete security of Learning with Errors instances.

The main intend of this estimator is to give designers an easy way to choose parameters resisting known attacks and to enable cryptanalysts to compare their results and ideas with other techniques known in the literature.

Usage Examples

```
sage: load("estimator.py")
sage: n, alpha, q = Param.Regev(128)
sage: costs = estimate_lwe(n, alpha, q)
usvp: rop:  2^57.7,  red:  2^57.7,  δ_0: 1.009214,  β:  102,  d:  357,  m:      228
 dec: rop:  2^61.5,  m:      229,  red:  2^61.5,  δ_0: 1.009595,  β:   93,  d:  357,␣
↪babai:  2^46.8,  babai_op:  2^61.9,  repeat:      293,  ϵ: 0.015625
dual: rop:  2^81.4,  m:      376,  red:  2^81.4,  δ_0: 1.008810,  β:  111,  d:  376,␣
↪|v|:  736.521,  repeat:  2^19.0,  ϵ: 0.003906
```

Online

You can run the estimator online using the Sage Math Cell server.

Coverage

At present the following algorithms are covered by this estimator.

- meet-in-the-middle exhaustive search
- Coded-BKW [C:GuoJohSta15]
- dual-lattice attack and small/sparse secret variant [EC:Albrecht17]
- lattice-reduction + enumeration [RSA:LinPei11]
- primal attack via uSVP [ICISC:AlbFitGop13,ACISP:BaiGal14]
- Arora-Ge algorithm [ICALP:AroGe11] using Gröbner bases [EPRINT:ACFP14]

Above, we use cryptobib-style bibtex keys as references.

Documentation

Documentation for the `estimator` is available here.

Evolution

This code is evolving, new results are added and bugs are fixed. Hence, estimations from earlier versions might not match current estimations. This is annoying but unavoidable at present. We recommend to also state the commit that was used when referencing this project.

We also encourage authors to let us know if their paper uses this code. In particular, we thrive to tag commits with those cryptobib ePrint references that use it. For example, this commit corresponds to this ePrint entry.

Contributions

Our intent is for this estimator to be maintained by the research community. For example, we encourage algorithm designers to add their own algorithms to this estimator and we are happy to help with that process.

More generally, all contributions such as bugfixes, documentation and tests are welcome. Please go ahead and submit your pull requests. Also, don't forget to add yourself to the list of contributors below in your pull requests.

At present, this estimator is maintained by Martin Albrecht. Contributors are:

- Martin Albrecht
- Florian Göpfert
- Cedric Lefebvre
- James Owen
- Rachel Player
- Markus Schmidt
- Sam Scott
- Fernando Virdia

Please follow PEP8 in your submissions. You can use flake8 to check for compliance. We use the following flake8 configuration (to allow longer line numbers and more complex functions):

```
[flake8]
max-line-length = 120
max-complexity = 16
ignore = E22,E241
```

Bugs

If you run into a bug, please open an issue on bitbucket. Also, please check first if the issue has already been reported.

Citing

If you use this estimator in your work, please cite

> Martin R. Albrecht, Rachel Player and Sam Scott. *On the concrete hardness of Learning with Errors*. Journal of Mathematical Cryptology. Volume 9, Issue 3, Pages 169–203, ISSN (Online) 1862-2984, ISSN (Print) 1862-2976 DOI: 10.1515/jmc-2015-0016, October 2015

A pre-print is available as

> Cryptology ePrint Archive, Report 2015/046, 2015. https://eprint.iacr.org/2015/046

A high-level overview of that work is given, for instance, in this talk.

Parameters from the Literature

The following estimates for various schemes from the literature illustrate the behaviour of the `estimator` . These estimates do not necessarily correspond to the claimed security levels of the respective schemes: for several parameter sets below the claimed security level by the designers' is lower than the complexity estimated by the `estimator` . This is usually because the designers anticipate potential future improvements to lattice-reduction algorithms and strategies. We recommend to follow the designers' decision. We intend to extend the estimator to cover these more optimistic (from an attacker's point of view) estimates in the future . . . pull requests welcome, as always.

New Hope

```
sage: load("estimator.py")
sage: n = 1024; q = 12289; stddev = sqrt(16/2); alpha = alphaf(sigmaf(stddev), q)
sage: _ = estimate_lwe(n, alpha, q, reduction_cost_model=BKZ.sieve)
usvp: rop: 2^313.1,  red: 2^313.1,  δ_0: 1.002094,  β:  968,  d: 2101,  m:     1076
 dec: rop: 2^410.0,  m:     1308,  red: 2^410.0,  δ_0: 1.001763,  β: 1213,  d: 2332, ␣
↪babai: 2^395.5,  babai_op: 2^410.6,  repeat:  2^25.2,  ϵ: 2^-23.0
dual: rop: 2^355.5,  m:     1239,  red: 2^355.5,  δ_0: 1.001884,  β: 1113,  repeat: 2^
↪307.0,  d: 2263,  c:        1
```

Frodo

```
sage: load("estimator.py")
sage: n = 752; q = 2^15; stddev = sqrt(1.75); alpha = alphaf(sigmaf(stddev), q)
sage: _ = estimate_lwe(n, alpha, q, reduction_cost_model=BKZ.sieve)
usvp: rop: 2^173.0,  red: 2^173.0,  δ_0: 1.003453,  β:  490,  d: 1448,  m:      695
 dec: rop: 2^208.3,  m:      829,  red: 2^208.3,  δ_0: 1.003064,  β:  579,  d: 1581, ␣
↪babai: 2^194.5,  babai_op: 2^209.6,  repeat:      588,  ϵ: 0.007812
dual: rop: 2^196.2,  m:      836,  red: 2^196.2,  δ_0: 1.003104,  β:  569,  repeat: 2^
↪135.0,  d: 1588,  c:        1
```

TESLA

```
sage: load("estimator.py")
sage: n = 804;  q = 2^31 - 19; alpha = sqrt(2*pi)*57/q; m = 4972
sage: _ = estimate_lwe(n, alpha, q, m=m, reduction_cost_model=BKZ.sieve)
usvp: rop: 2^129.3,  red: 2^129.3,  δ_0: 1.004461,  β:  339,  d: 1954,  m:     1149
 dec: rop: 2^144.9,  m:     1237,  red: 2^144.9,  δ_0: 1.004148,  β:  378,  d: 2041, ␣
↪babai: 2^130.9,  babai_op: 2^146.0,  repeat:       17,  ϵ: 0.250000
dual: rop: 2^139.4,  m:     1231,  red: 2^139.4,  δ_0: 1.004180,  β:  373,  repeat:  2^
↪93.0,  d: 2035,  c:        1
```

SEAL v2.2

```
sage: load("estimator.py")
sage: n = 2048; q = 2^60 - 2^14 + 1; alpha = 8/q; m = 2*n
sage: _ = estimate_lwe(n, alpha, q, secret_distribution=(-1,1), reduction_cost_
 ↪model=BKZ.sieve, m=m)
usvp: rop: 2^115.5,  red: 2^115.5,  δ_0: 1.004975,  β:  288,  d: 4013,  m:     1964
 dec: rop: 2^127.1,  m:  2^11.1,  red: 2^127.1,  δ_0: 1.004663,  β:  318,  d: 4237, ␣
 ↪babai: 2^114.8,  babai_op: 2^129.9,  repeat:        7,  ϵ: 0.500000
dual: rop: 2^118.4,  m:  2^11.0,  red: 2^118.4,  δ_0: 1.004864,  β:  298,  repeat:  2^
 ↪58.8,  d: 4090,  c:    3.909,  k:       30,  postprocess:       13
```

CHAPTER 2

Documentation README

Documentation for the `estimator` is available online. This documentation can be generated locally by running the following code in the lwe-estimator directory:

```
pipenv run make html
```

If documentation was previously generated locally, to ensure a full regeneration use:

```
pipenv run make clean && rm -fr doc/_apidoc
```

CHAPTER 3

estimator

Description

Cost estimates for solving LWE.

Supported Secret Distributions

The following distributions for the secret are supported:

- `"normal"` : normal form instances, i.e. the secret follows the noise distribution (alias: `True`)
- `"uniform"` : uniform mod q (alias: `False`)
- `(a,b)` : uniform in the interval `[a,...,b]`
- `((a,b),h)` : exactly `h` components are `[a,...,b]\{0\}` , all other components are zero

Classes

BKZ	Cost estimates for BKZ.
Cost ([data])	Algorithms costs.
Logging	Control level of detail being printed.
OrderedDict (*args, **kwds)	Dictionary that remembers insertion order
Param	Namespace for processing LWE parameter sets.
SDis	Distributions of Secrets.
partial	partial(func, *args, **keywords) - new function with partial application

estimator.BKZ

class `estimator.` **`BKZ`**

Cost estimates for BKZ.

Methods

ADPS16 (beta, d[, B, mode])	Runtime estimation from *[ADPS16]*.

Continued on next page

Table 3.2 – continued from previous page

BDGL16 (beta, d[, B])	Runtime estimation given β and assuming sieving is used to realise the SVP oracle.
CheNgu12 (beta, d[, B])	Runtime estimation given β and assuming *[CheNgu12]* estimates are correct.
LLL (d[, B])	Runtime estimation for LLL algorithm
LaaMosPol14 (beta, d[, B])	Runtime estimation for quantum sieving.
LinPei11 (beta, d[, B])	Runtime estimation assuming the Lindner-Peikert model in elementary operations.
enum (beta, d[, B])	Runtime estimation given β and assuming *[CheNgu12]* estimates are correct.
lp (beta, d[, B])	Runtime estimation assuming the Lindner-Peikert model in elementary operations.
qsieve (beta, d[, B])	Runtime estimation for quantum sieving.
sieve (beta, d[, B])	Runtime estimation given β and assuming sieving is used to realise the SVP oracle.
svp_repeat (beta, d)	Return number of SVP calls in BKZ-β

estimator.BKZ.ADPS16

static BKZ. **ADPS16** (*beta*, *d*, *B=None*, *mode='classical'*)

Runtime estimation from *[ADPS16]*.

Parameters

- **beta** – block size
- **n** – LWE dimension $n > 0$
- **B** – bit-size of entries

EXAMPLE:

```
sage: from estimator import BKZ, Param, dual, partial
sage: cost_model = partial(BKZ.ADPS16, mode="paranoid")
sage: dual(*Param.LindnerPeikert(128), reduction_cost_model=cost_model)
    rop:   2^37.3
      m:       346
    red:   2^37.3
delta_0: 1.008209
   beta:       127
      d:       346
    |v|:  284.363
 repeat:      2000
epsilon: 0.062500
```

estimator.BKZ.BDGL16

static BKZ. **BDGL16** (*beta*, *d*, *B=None*)

Runtime estimation given β and assuming sieving is used to realise the SVP oracle.

param beta block size

param n LWE dimension $n > 0$

param B bit-size of entries

estimator.BKZ.CheNgu12

static BKZ. **CheNgu12**(*beta*, *d*, *B=None*)

Runtime estimation given β and assuming *[CheNgu12]* estimates are correct.

Parameters

- **beta** – block size
- **n** – LWE dimension $n > 0$
- **B** – bit-size of entries

The constants in this function were derived as follows based on Table 4 in *[CheNgu12]*:

```
sage: dim = [100, 110, 120, 130, 140, 150, 160, 170, 180, 190, 200, 210, 220,␣
↪230, 240, 250]
sage: nodes = [39.0, 44.0, 49.0, 54.0, 60.0, 66.0, 72.0, 78.0, 84.0, 96.0, 99.
↪0, 105.0, 111.0, 120.0, 127.0, 134.0]  # noqa
sage: times = [c + log(200,2).n() for c in nodes]
sage: T = zip(dim, nodes)
sage: var("a,b,c,k")
(a, b, c, k)
sage: f = a*k*log(k, 2.0) + b*k + c
sage: f = f.function(k)
sage: f.subs(find_fit(T, f, solution_dict=True))
k |--> 0.270188776350190*k*log(k) - 1.0192050451318417*k + 16.10253135200765
```

The estimation

2^(0.270188776350190*beta*log(beta) - 1.0192050451318417*beta + 16.10253135200765)

is of the number of enumeration nodes, hence we need to multiply by the number of cycles to process one node. This cost per node is typically estimated as 100 [FPLLL].

estimates (Full Version). 2012. http://www.di.ens.fr/~ychen/research/Full_BKZ.pdf

Available at https://github.com/fplll/fplll

estimator.BKZ.LLL

static BKZ. **LLL**(*d*, *B=None*)

Runtime estimation for LLL algorithm

Parameters

- **d** – lattice dimension
- **B** – bit-size of entries

estimator.BKZ.LaaMosPol14

static BKZ. **LaaMosPol14**(*beta*, *d*, *B=None*)

Runtime estimation for quantum sieving.

Parameters

- **beta** – block size
- **n** – LWE dimension $n > 0$

- **B** – bit-size of entries

estimator.BKZ.LinPei11

static BKZ. **LinPei11** (*beta*, *d*, *B=None*)

Runtime estimation assuming the Lindner-Peikert model in elementary operations.

Parameters

- **beta** – block size
- **d** – lattice dimension
- **B** – bit-size of entries

estimator.BKZ.enum

static BKZ. **enum** (*beta*, *d*, *B=None*)

Runtime estimation given β and assuming *[CheNgu12]* estimates are correct.

Parameters

- **beta** – block size
- **n** – LWE dimension $n > 0$
- **B** – bit-size of entries

The constants in this function were derived as follows based on Table 4 in *[CheNgu12]*:

```
sage: dim = [100, 110, 120, 130, 140, 150, 160, 170, 180, 190, 200, 210, 220,␣
↪230, 240, 250]
sage: nodes = [39.0, 44.0, 49.0, 54.0, 60.0, 66.0, 72.0, 78.0, 84.0, 96.0, 99.
↪0, 105.0, 111.0, 120.0, 127.0, 134.0]  # noqa
sage: times = [c + log(200,2).n() for c in nodes]
sage: T = zip(dim, nodes)
sage: var("a,b,c,k")
(a, b, c, k)
sage: f = a*k*log(k, 2.0) + b*k + c
sage: f = f.function(k)
sage: f.subs(find_fit(T, f, solution_dict=True))
k |--> 0.270188776350190*k*log(k) - 1.0192050451318417*k + 16.10253135200765
```

The estimation

2^(0.270188776350190*beta*log(beta) - 1.0192050451318417*beta + 16.10253135200765)

is of the number of enumeration nodes, hence we need to multiply by the number of cycles to process one node. This cost per node is typically estimated as 100 [FPLLL].

estimates (Full Version). 2012. http://www.di.ens.fr/~ychen/research/Full_BKZ.pdf

Available at https://github.com/fplll/fplll

estimator.BKZ.lp

static BKZ. **lp** (*beta*, *d*, *B=None*)

Runtime estimation assuming the Lindner-Peikert model in elementary operations.

Parameters

- **beta** – block size
- **d** – lattice dimension
- **B** – bit-size of entries

estimator.BKZ.qsieve

static BKZ.**qsieve**(*beta*, *d*, *B=None*)

Runtime estimation for quantum sieving.

Parameters

- **beta** – block size
- **n** – LWE dimension $n > 0$
- **B** – bit-size of entries

estimator.BKZ.sieve

static BKZ.**sieve**(*beta*, *d*, *B=None*)

Runtime estimation given β and assuming sieving is used to realise the SVP oracle.

param beta block size

param n LWE dimension $n > 0$

param B bit-size of entries

estimator.BKZ.svp_repeat

static BKZ.**svp_repeat**(*beta*, *d*)

Return number of SVP calls in BKZ-β

Parameters

- **beta** – block size
- **d** – dimension

Note: loosely based on experiments in [PhD:Chen13]

Attributes

GSA

estimator.BKZ.GSA

BKZ.**GSA** = <Mock name='mock()' id='139783166715728'>

estimator.Cost

class `estimator.` **`Cost`** (*data=None*, ***kwds*)

Algorithms costs.

Parameters **`data`** – we call `OrderedDict(data)`

`__init__` (*data=None*, ***kwds*)

Parameters **`data`** – we call `OrderedDict(data)`

Methods

__init__ ([data])	**param data** we call `OrderedDict(data)`
combine (right[, base])	Combine `left` and `right` .
filter (keys)	Return new ordered dictinonary from dictionary restricted to the keys.
reorder (first)	Return a new ordered dict from the key:value pairs in dictinonary but reordered such that the `first` keys come first.
repeat (times[, select, lll])	Return a report with all costs multiplied by *times*.
str ([keyword_width, newline, round_bound, ...])	**param keyword_width** keys are printed with this width
values ()	

estimator.Cost.__init__

`Cost.` **`__init__`** (*data=None*, ***kwds*)

Parameters **`data`** – we call `OrderedDict(data)`

estimator.Cost.combine

`Cost.` **`combine`** (*right*, *base=None*)

Combine `left` and `right` .

Parameters

- **`left`** – cost dictionary
- **`right`** – cost dictionary
- **`base`** – add entries to `base`

estimator.Cost.filter

`Cost.` **`filter`** (*keys*)

Return new ordered dictinonary from dictionary restricted to the keys.

Parameters

- **dictionary** – input dictionary
- **keys** – keys which should be copied (ordered)

estimator.Cost.reorder

Cost. **reorder** (*first*)

Return a new ordered dict from the key:value pairs in dictinonary but reordered such that the `first` keys come first.

Parameters

- **dictionary** – input dictionary
- **first** – keys which should come first (in order)

EXAMPLE:

```
sage: from estimator import Cost
sage: d = Cost([("a",1),("b",2),("c",3)]); d
a:         1
b:         2
c:         3

sage: d.reorder( ["b","c","a"])
b:         2
c:         3
a:         1
```

estimator.Cost.repeat

Cost. **repeat** (*times*, *select=None*, *lll=None*)

Return a report with all costs multiplied by *times*.

Parameters

- **d** – a cost estimate
- **times** – the number of times it should be run
- **select** – toggle which fields ought to be repeated and which shouldn't
- **lll** – if set amplify lattice reduction times assuming the LLL algorithm suffices and costs `lll`

Returns a new cost estimate

We maintain a local dictionary which decides if an entry is multiplied by *times* or not. For example, δ would not be multiplied but "#bop" would be. This check is strict such that unknown entries raise an error. This is to enforce a decision on whether an entry should be multiplied by *times* if the function *report* reports on is called *times* often.

EXAMPLE:

```
sage: from estimator import Param, dual
sage: n, alpha, q = Param.Regev(128)

sage: dual(n, alpha, q).repeat(2^10)
```

```
                rop:   2^91.4
                  m:   2^18.6
                red:   2^91.4
            delta_0: 1.008810
               beta:      111
                  d:      376
                 |v|:  736.521
             repeat:   2^29.0
            epsilon: 0.003906

sage: dual(n, alpha, q).repeat(1)
            rop:   2^81.4
              m:      376
            red:   2^81.4
        delta_0: 1.008810
           beta:      111
              d:      376
             |v|:  736.521
         repeat:   2^19.0
        epsilon: 0.003906
```

estimator.Cost.str

Cost.**str**(*keyword_width=None*, *newline=None*, *round_bound=2048*, *compact=False*, *unicode=True*)

Parameters

- **keyword_width** – keys are printed with this width
- **newline** – insert a newline
- **round_bound** – values beyond this bound are represented as powers of two
- **compact** – do not add extra whitespace to align entries
- **unicode** – use unicode to shorten representation

EXAMPLE:

```
sage: from estimator import Cost
sage: s = Cost({"delta_0":5, "bar":2})
sage: print(s)
bar: 2, delta_0: 5

sage: s = Cost([(u"delta_0", 5), ("bar",2)])
sage: print(s)
delta_0: 5, bar: 2
```

estimator.Cost.values

Cost.**values**()

estimator.Logging

class estimator. **Logging**
Control level of detail being printed.

Methods

set_level (lvl[, loggers])	Set logging level

estimator.Logging.set_level

static Logging. **set_level** (*lvl*, *loggers=None*)
Set logging level

Parameters

- **lvl** – one of *CRITICAL*, *ERROR*, *WARNING*, *INFO*, *DEBUG*, *NOTSET*
- **loggers** – one of *Logging.loggers*, if *None* all loggers are used.

Attributes

CRITICAL
DEBUG
ERROR
INFO
NOTSET
WARNING
detail_logger
logger
loggers
plain_logger

estimator.Logging.CRITICAL

Logging. **CRITICAL = 50**

estimator.Logging.DEBUG

Logging. **DEBUG = 10**

estimator.Logging.ERROR

Logging. **ERROR = 40**

estimator.Logging.INFO

Logging. **INFO = 20**

estimator.Logging.NOTSET

Logging. **NOTSET = 0**

estimator.Logging.WARNING

Logging. **WARNING = 30**

estimator.Logging.detail_logger

Logging. **detail_logger = <logging.StreamHandler object>**

estimator.Logging.logger

Logging. **logger = 'guess'**

estimator.Logging.loggers

Logging. **loggers = ('binsearch', 'repeat', 'guess')**

estimator.Logging.plain_logger

Logging. **plain_logger = <logging.StreamHandler object>**

estimator.OrderedDict

class estimator. **OrderedDict** (**args*, ***kwds*)

Dictionary that remembers insertion order

Initialize an ordered dictionary. The signature is the same as regular dictionaries, but keyword arguments are not recommended because their insertion order is arbitrary.

__init__ (**args*, ***kwds*)

Initialize an ordered dictionary. The signature is the same as regular dictionaries, but keyword arguments are not recommended because their insertion order is arbitrary.

Methods

__init__ (*args, **kwds)	Initialize an ordered dictionary.
clear (() -> None. Remove all items from od.)	
copy (() -> a shallow copy of od)	

Continued on next page

Table 3.7 – continued from previous page

fromkeys ((S[, ...)	If not specified, the value defaults to None.
get ((k[,d]) -> D[k] if k in D, ...)	
has_key ((k) -> True if D has a key k, else False)	
items (() -> list of (key, value) pairs in od)	
iteritems ()	od.iteritems -> an iterator over the (key, value) pairs in od
iterkeys (() -> an iterator over the keys in od)	
itervalues ()	od.itervalues -> an iterator over the values in od
keys (() -> list of keys in od)	
pop ((k[,d]) -> v, ...)	value. If key is not found, d is returned if given, otherwise KeyError
popitem (() -> (k, v), ...)	Pairs are returned in LIFO order if last is true or FIFO order if false.
setdefault ((k[,d]) -> od.get(k,d), ...)	
update (([E, ...)	If E present and has a .keys() method, does: for k in E: D[k] = E[k]
values (() -> list of values in od)	
viewitems (...)	
viewkeys (...)	
viewvalues (...)	

estimator.OrderedDict.__init__

OrderedDict.**__init__**(**args*, ***kwds*)

Initialize an ordered dictionary. The signature is the same as regular dictionaries, but keyword arguments are not recommended because their insertion order is arbitrary.

estimator.OrderedDict.clear

OrderedDict.**clear**() → None. Remove all items from od.

estimator.OrderedDict.copy

OrderedDict.**copy**() → a shallow copy of od

estimator.OrderedDict.fromkeys

classmethod OrderedDict.**fromkeys**(*S*[, *v*]) → New ordered dictionary with keys from S.

If not specified, the value defaults to None.

estimator.OrderedDict.get

OrderedDict.**get**(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

estimator.OrderedDict.has_key

OrderedDict.**has_key**(*k*) → True if D has a key k, else False

estimator.OrderedDict.items

OrderedDict. **items** ( ) → list of (key, value) pairs in od

estimator.OrderedDict.iteritems

OrderedDict. **iteritems** ( )
od.iteritems -> an iterator over the (key, value) pairs in od

estimator.OrderedDict.iterkeys

OrderedDict. **iterkeys** ( ) → an iterator over the keys in od

estimator.OrderedDict.itervalues

OrderedDict. **itervalues** ( )
od.itervalues -> an iterator over the values in od

estimator.OrderedDict.keys

OrderedDict. **keys** ( ) → list of keys in od

estimator.OrderedDict.pop

OrderedDict. **pop** ($k[, d]$) → v, remove specified key and return the corresponding value. If key is not found, d is returned if given, otherwise KeyError is raised.

estimator.OrderedDict.popitem

OrderedDict. **popitem** ( ) → (k, v), return and remove a (key, value) pair.
Pairs are returned in LIFO order if last is true or FIFO order if false.

estimator.OrderedDict.setdefault

OrderedDict. **setdefault** ($k[, d]$) → od.get(k,d), also set od[k]=d if k not in od

estimator.OrderedDict.update

OrderedDict. **update** ($[E]$, **F) → None. Update D from mapping/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

estimator.OrderedDict.values

OrderedDict. **values** ( ) → list of values in od

estimator.OrderedDict.viewitems

OrderedDict. **viewitems** ( ) → a set-like object providing a view on od's items

estimator.OrderedDict.viewkeys

OrderedDict. **viewkeys** ( ) → a set-like object providing a view on od's keys

estimator.OrderedDict.viewvalues

OrderedDict. **viewvalues** ( ) → an object providing a view on od's values

estimator.Param

class estimator. **Param**

Namespace for processing LWE parameter sets.

Methods

LindnerPeikert (n[, m, dict])	**param n** LWE dimension $n > 0$
Regev (n[, m, dict])	**param n** LWE dimension $n > 0$
dict (lwe)	Return dictionary consisting of n, α, q and samples given an LWE instance object.
preprocess (n, alpha, q[, ...])	Check if parameters n, α, q are sound and return correct types.
tuple (lwe)	Return (n, α, q) given an LWE instance object.

estimator.Param.LindnerPeikert

static Param. **LindnerPeikert** (*n*, *m=None*, *dict=False*)

Parameters

- **n** – LWE dimension $n > 0$
- **m** – number of LWE samples $m > 0$
- **dict** – return a dictionary

estimator.Param.Regev

static Param. **Regev** (*n*, *m=None*, *dict=False*)

Parameters

- **n** – LWE dimension $n > 0$

- **m** – number of LWE samples $m > 0$
- **dict** – return a dictionary

estimator.Param.dict

static Param. **dict** (*lwe*)

Return dictionary consisting of n, α, q and samples given an LWE instance object.

Parameters **lwe** – LWE object

Returns "n": n, "alpha": α, "q": q, "samples": samples

Return type dictionary

estimator.Param.preprocess

static Param. **preprocess** (*n*, *alpha*, *q*, *success_probability=None*, *prec=None*, *m=<Mock name='mock()' id='139783166714704'>*)

Check if parameters n, α, q are sound and return correct types. Also, if given, the soundness of the success probability and the number of samples is ensured.

estimator.Param.tuple

static Param. **tuple** (*lwe*)

Return (n, α, q) given an LWE instance object.

Parameters **lwe** – LWE object

Returns (n, α, q)

estimator.SDis

class estimator. **SDis**

Distributions of Secrets.

Methods

bounds (secret_distribution)	Return bounds of secret distribution
is_bounded_uniform (secret_distribution)	Return true if the secret is bounded uniform (sparse or not).
is_small (secret_distribution)	Return true if the secret distribution is small
is_sparse (secret_distribution)	Return true if the secret distribution is sparse
is_ternary (secret_distribution)	Return true if the secret is ternary (sparse or not)
mean (secret_distribution[, q, n])	Mean of the secret per component.
nonzero (secret_distribution, n)	Return number of non-zero elements or None
variance (secret_distribution[, alpha, q, n])	Variance of the secret per component.

estimator.SDis.bounds

static SDis. **bounds** (*secret_distribution*)

Return bounds of secret distribution

Parameters **secret_distribution** – distribution of secret, see module level documentation for details

EXAMPLES:

```
sage: from estimator import SDis
sage: SDis.bounds(False)
Traceback (most recent call last):
...
ValueError: Cannot extract bounds for secret.

sage: SDis.bounds(True)
Traceback (most recent call last):
...
ValueError: Cannot extract bounds for secret.

sage: SDis.bounds(((-1, 1), 64))
(-1, 1)

sage: SDis.bounds(((-3, 3), 64))
(-3, 3)

sage: SDis.bounds((-3, 3))
(-3, 3)
```

estimator.SDis.is_bounded_uniform

static SDis. **is_bounded_uniform** (*secret_distribution*)

Return true if the secret is bounded uniform (sparse or not).

Parameters **secret_distribution** – distribution of secret, see module level documentation for details

EXAMPLES:

```
sage: from estimator import SDis
sage: SDis.is_bounded_uniform(False)
False

sage: SDis.is_bounded_uniform(True)
False

sage: SDis.is_bounded_uniform(((-1, 1), 64))
True

sage: SDis.is_bounded_uniform(((-3, 3), 64))
True

sage: SDis.is_bounded_uniform((-3, 3))
True
```

Note: This function requires the bounds to be of opposite sign, as scaling code does not handle the other case.

estimator.SDis.is_small

static SDis. **is_small** (*secret_distribution*)

Return true if the secret distribution is small

Parameters **secret_distribution** – distribution of secret, see module level documentation for details

EXAMPLES:

```
sage: from estimator import SDis
sage: SDis.is_small(False)
False

sage: SDis.is_small(True)
True

sage: SDis.is_small(((-1, 1), 64))
True

sage: SDis.is_small(((-3, 3), 64))
True

sage: SDis.is_small((-3, 3))
True
```

estimator.SDis.is_sparse

static SDis. **is_sparse** (*secret_distribution*)

Return true if the secret distribution is sparse

Parameters **secret_distribution** – distribution of secret, see module level documentation for details

EXAMPLES:

```
sage: from estimator import SDis
sage: SDis.is_sparse(True)
False

sage: SDis.is_sparse(((-1, 1), 64))
True

sage: SDis.is_sparse(((-3, 3), 64))
True

sage: SDis.is_sparse((-3, 3))
False
```

estimator.SDis.is_ternary

static SDis. **is_ternary** (*secret_distribution*)
Return true if the secret is ternary (sparse or not)

Parameters **secret_distribution** – distribution of secret, see module level documentation for details

EXAMPLES:

```
sage: from estimator import SDis
sage: SDis.is_ternary(False)
False

sage: SDis.is_ternary(True)
False

sage: SDis.is_ternary(((-1, 1), 64))
True

sage: SDis.is_ternary((-1, 1))
True

sage: SDis.is_ternary(((-3, 3), 64))
False

sage: SDis.is_ternary((-3, 3))
False
```

estimator.SDis.mean

static SDis. **mean** (*secret_distribution*, *q=None*, *n=None*)
Mean of the secret per component.

Parameters

- **secret_distribution** – distribution of secret, see module level documentation for details
- **n** – only used for sparse secrets

EXAMPLE:

```
sage: from estimator import SDis
sage: SDis.mean(True)
0

sage: SDis.mean(False, q=10)
0

sage: SDis.mean(((-3,3)))
0

sage: SDis.mean(((-3,3),64), n=256)
0

sage: SDis.mean(((-3,2)))
-1/2
```

```
sage: SDis.mean(((-3,2),64), n=256)
-3/20
```

estimator.SDis.nonzero

static SDis. **nonzero** (*secret_distribution*, *n*)

Return number of non-zero elements or None

Parameters

- **secret_distribution** – distribution of secret, see module level documentation for details
- **n** – LWE dimension $n > 0$

estimator.SDis.variance

static SDis. **variance** (*secret_distribution*, *alpha=None*, *q=None*, *n=None*)

Variance of the secret per component.

Parameters

- **secret_distribution** – distribution of secret, see module level documentation for details
- **alpha** – only used for normal form LWE
- **q** – only used for normal form LWE
- **n** – only used for sparse secrets

EXAMPLE:

```
sage: from estimator import SDis
sage: SDis.variance(True, 8./2^15, 2^15).sqrt().n()
3.19...

sage: SDis.variance((-3,3), 8./2^15, 2^15)
4

sage: SDis.variance(((-3,3),64), 8./2^15, 2^15, n=256)
7/6

sage: SDis.variance((-3,2))
35/12

sage: SDis.variance(((-3,2),64), n=256)
371/400

sage: SDis.variance((-1,1), 8./2^15, 2^15)
2/3

sage: SDis.variance(((-1,1),64), 8./2^15, 2^15, n=256)
1/4
```

Note: This function assumes that the bounds are of opposite sign, and that the distribution is centred around zero.

estimator.partial

class estimator. **partial**
partial(func, *args, **keywords) - new function with partial application of the given arguments and keywords.

__init__ ( )
x.__init__(...) initializes x; see help(type(x)) for signature

Attributes

args	tuple of arguments to future partial calls
func	function object to use in future partial calls
keywords	dictionary of keyword arguments to future partial calls

estimator.partial.args

partial. **args**
tuple of arguments to future partial calls

estimator.partial.func

partial. **func**
function object to use in future partial calls

estimator.partial.keywords

partial. **keywords**
dictionary of keyword arguments to future partial calls

Functions

alphaf (sigma, q[, sigma_is_stddev])	Gaussian width σ, modulus q $\rightarrow$ noise rate α
amplify (target_success_probability, ...[, ...])	Return the number of trials needed to amplify current *success_probability* to
amplify_sigma (target_advantage, sigma, q)	Amplify distinguishing advantage for a given σ and q
arora_gb (n, alpha, q[, secret_distribution, ...])	Arora-GB as described in [AroGe11,ACFP14]_
betaf (delta)	Compute block size β from root-Hermite factor δ_0.
binary_search (f, start, stop, param[, predicate])	Searches for the best value in the interval [start,stop] depending on the given predicate.
binary_search_minimum (f, start, stop, param)	Return minimum of f if f is convex.

Continued on next page

Table 3.11 – continued from previous page

bkw_coded (n, alpha, q[, ...])	Coded-BKW as described in [C:GuoJohSta15]
delta_0f (beta)	Compute root-Hermite factor δ_0 from block size β.
drop_and_solve (f, n, alpha, q[, ...])	Solve instances of dimension n-k with increasing k using f and pick parameters such that cost is minimised.
dual_scale (n, alpha, q, secret_distribution)	Estimate cost of solving LWE by finding small *(y,x/c)* such that *y A c x mod q* as
enumeration_cost (n, alpha, q, ...[, ...])	Estimates the cost of performing enumeration.
estimate_lwe (n[, alpha, q, ...])	Highlevel-function for estimating security of LWE parameter sets
gb_cost (n, D[, omega])	Estimate the complexity of computing a Gröbner basis.
guess_and_solve (f, n, alpha, q, ...[, ...])	Guess components of the secret.
lattice_reduction_cost (cost_model, delta_0, d)	Return cost dictionary for returning vector of norm‘ δ_0^d Vol(Λ)^{1/d}‘ using provided lattice reduction algorithm.
lattice_reduction_opt_m (n, q, delta)	Return the (heuristically) optimal lattice dimension *m*
mitm (n, alpha, q[, secret_distribution, m, ...])	Meet-in-the-Middle attack as described in *[AlbPlaSco15]*
primal_usvp (n, alpha, q[, ...])	Estimate cost of solving LWE using primal attack (uSVP version)
reduction_default_cost (beta, d[, B])	Runtime estimation given β and assuming *[CheNgu12]* estimates are correct.
rinse_and_repeat (f, n, alpha, q[, ...])	Find best trade-off between success probability and running time.
sieve_or_enum (func)	Take minimum of sieving or enumeration for lattice-based attacks.
sigmaf (stddev)	standard deviation → Gaussian width parameter σ
stddevf (sigma)	Gaussian width parameter σ → standard deviation
success_probability_drop (n, h, k[, fail, ...])	Probability that k randomly sampled components have fail non-zero components amongst them.
switch_modulus (f, n, alpha, q, ...)	**param f** run f

estimator.alphaf

estimator. **alphaf** (*sigma*, *q*, *sigma_is_stddev=False*)

Gaussian width σ, modulus q → noise rate α

Parameters

- **sigma** – Gaussian width parameter (or standard deviation if sigma_is_stddev is set)
- **q** – modulus *0 < q*
- **sigma_is_stddev** – if set then *sigma* is interpreted as the standard deviation

Returns $\alpha = \sigma$/q or $\sigma \cdot$sqrt(2π)/q depending on *sigma_is_stddev*

estimator.amplify

estimator. **amplify** (*target_success_probability*, *success_probability*, *majority=False*)

Return the number of trials needed to amplify current *success_probability* to *target_success_probability*

Parameters

- **target_success_probability** – targeted success probability < 1
- **success_probability** – targeted success probability < 1
- **majority** – if *True* amplify a deicsional problem, not a computational one if *False* then we assume that we can check solutions, so one success suffices

Returns number of required trials to amplify

estimator.amplify_sigma

estimator. **amplify_sigma** (*target_advantage*, *sigma*, *q*)

Amplify distinguishing advantage for a given σ and q

Parameters

- **target_advantage** –
- **sigma** – (lists of) Gaussian width parameters
- **q** – modulus *0 < q*

estimator.arora_gb

estimator. **arora_gb** (*n*, *alpha*, *q*, *secret_distribution=True*, *m=<Mock name='mock()' id='139783166714704'>*, *success_probability=0.99*, *omega=2*)

Arora-GB as described in [AroGe11,ACFP14]_

Parameters

- **n** – LWE dimension *n > 0*
- **alpha** – noise rate *0 α < 1*, noise will have standard deviation *αq/sqrt{2π}*
- **q** – modulus *0 < q*
- **secret_distribution** – distribution of secret, see module level documentation for details
- **m** – number of LWE samples *m > 0*
- **success_probability** – targeted success probability < 1
- **omega** – linear algebra constant

estimator.betaf

estimator. **betaf** (*delta*)

Compute block size β from root-Hermite factor δ_0.

estimator.binary_search

estimator. **binary_search** (*f*, *start*, *stop*, *param*, *predicate=<function <lambda>>*, **arg*, ***kwds*)

Searches for the best value in the interval [start,stop] depending on the given predicate.

Parameters

- **start** – start of range to search
- **stop** – stop of range to search (exclusive)
- **param** – the parameter to modify when calling *f*
- **predicate** – comparison is performed by evaluating `predicate(current,best)`

estimator.binary_search_minimum

estimator. **binary_search_minimum** (*f*, *start*, *stop*, *param*, *extract=<function <lambda>>*, **arg*, ***kwds*)

Return minimum of *f* if *f* is convex.

Parameters

- **start** – start of range to search
- **stop** – stop of range to search (exclusive)
- **param** – the parameter to modify when calling *f*
- **extract** – comparison is performed on `extract(f(param=?,*args,**kwds))`

estimator.bkw_coded

estimator. **bkw_coded** (*n*, *alpha*, *q*, *secret_distribution=True*, *m=<Mock name='mock()' id='139783166714704'>*, *success_probability=0.99*)

Coded-BKW as described in [C:GuoJohSta15]

Parameters

- **n** – LWE dimension $n > 0$
- **alpha** – noise rate $0 \alpha < 1$, noise will have standard deviation $\alpha q/sqrt{2\pi}$
- **q** – modulus $0 < q$
- **success_probability** – targeted success probability < 1
- **samples** – the number of available samples

EXAMPLE:

```
sage: from estimator import Param, bkw_coded
sage: n, alpha, q = Param.Regev(64)
sage: bkw_coded(n, alpha, q)
     rop:   2^50.9
       m:   2^39.6
     mem:   2^39.6
       b:         3
      t1:         2
      t2:        10
       l:         2
    ncod:        53
    ntop:         0
   ntest:         6
```

estimator.delta_0f

estimator. **delta_0f** (*beta*)

Compute root-Hermite factor δ_0 from block size β.

estimator.drop_and_solve

estimator. **drop_and_solve** (*f*, *n*, *alpha*, *q*, *secret_distribution=True*, *success_probability=0.99*, *postprocess=True*, *decision=True*, *rotations=False*, ***kwds*)

Solve instances of dimension `n-k` with increasing `k` using `f` and pick parameters such that cost is minimised.

Parameters

- **f** – attack estimate function
- **n** – LWE dimension $n > 0$
- **alpha** – noise rate $0 < \alpha < 1$, noise will have standard deviation $\alpha q/sqrt{2\pi}$
- **q** – modulus $0 < q$
- **secret_distribution** – distribution of secret, see module level documentation for details
- **success_probability** – targeted success probability < 1
- **postprocess** – check against shifted distributions
- **decision** – the underlying algorithm solves the decision version or not

EXAMPLE:

```
sage: from estimator import drop_and_solve, dual_scale, primal_usvp, partial
sage: q = next_prime(2^30)
sage: n, alpha = 512, 8/q
sage: primald = partial(drop_and_solve, primal_usvp)
sage: duald = partial(drop_and_solve, dual_scale)

sage: duald(n, alpha, q, secret_distribution=((-1,1), 64))
                 rop:   2^55.7
                   m:       478
                 red:   2^55.1
             delta_0: 1.009807
                beta:        89
              repeat:   2^14.2
                   d:       920
                   c:     8.387
                   k:        70
         postprocess:         8

sage: duald(n, alpha, q, secret_distribution=((-3,3), 64))
                 rop:   2^59.6
                   m:       517
                 red:   2^59.5
             delta_0: 1.009403
                beta:        97
              repeat:   2^18.2
                   d:       969
                   c:     3.926
```

```
                  k:       60
        postprocess:        6

sage: kwds = {"use_lll":False, "postprocess":False}
sage: duald(n, alpha, q, secret_distribution=((-1,1), 64), **kwds)
                rop:   2^69.8
                  m:      521
                red:   2^69.8
            delta_0: 1.008953
               beta:      107
             repeat:      257
                  d:     1033
                  c:    9.027
                  k:        0
        postprocess:        0

sage: duald(n, alpha, q, secret_distribution=((-3,3), 64), **kwds)
                rop:   2^74.5
                  m:      560
                red:   2^74.5
            delta_0: 1.008668
               beta:      114
             repeat:  524.610
                  d:     1068
                  c:    4.162
                  k:        4
        postprocess:        0

sage: duald(n, alpha, q, secret_distribution=((-3,3), 64), rotations=True, **kwds)
Traceback (most recent call last):
  ...
ValueError: Rotations are only support as part of the primal-usvp attack on NTRU.

sage: primald(n, alpha, q, secret_distribution=((-3,3), 64), rotations=True,␣
↪**kwds)
                rop:   2^51.1
                red:   2^51.1
            delta_0: 1.010046
               beta:       84
                  d:      914
                  m:      445
             repeat: 1.509286
                  k:       44
        postprocess:        0

sage: primald(n, alpha, q, secret_distribution=((-3,3), 64), rotations=False,␣
↪**kwds)
                rop:   2^58.0
                red:   2^58.0
            delta_0: 1.009350
               beta:       98
                  d:     1003
                  m:      494
             repeat: 1.708828
                  k:        4
        postprocess:        0
```

This function is based on:

estimator.dual_scale

estimator. **dual_scale** (*n*, *alpha*, *q*, *secret_distribution*, *m=<Mock name='mock()' id='139783166714704'>*, *success_probability=0.99*, *reduction_cost_model=<function CheNgu12>*, *c=None*, *use_lll=True*)

Estimate cost of solving LWE by finding small *(y,x/c)* such that *y A c x mod q* as described in [EC:Abrecht17]

Parameters

- **n** – LWE dimension *n > 0*
- **alpha** – noise rate *0 α < 1*, noise will have standard deviation *αq/sqrt{2π}*
- **q** – modulus *0 < q*
- **secret_distribution** – distribution of secret, see module level documentation for details
- **m** – number of LWE samples *m > 0*
- **success_probability** – targeted success probability < 1
- **reduction_cost_model** – cost model for lattice reduction
- **c** – explicit constant *c*
- **use_lll** – use LLL calls to produce more small vectors

EXAMPLES:

```
sage: from estimator import Param, dual_scale

sage: dual_scale(*Param.Regev(256), secret_distribution=(-1,1))
        rop:  2^105.4
          m:      286
        red:  2^105.4
    delta_0: 1.006698
       beta:      181
     repeat:   2^69.0
          d:      542
          c:   31.271

sage: dual_scale(*Param.Regev(256), secret_distribution=(-1,1), m=200)
        rop:  2^109.7
          m:      200
        red:  2^109.7
    delta_0: 1.006543
       beta:      188
     repeat:   2^74.0
          d:      456
          c:   31.271

sage: dual_scale(*Param.Regev(256), secret_distribution=((-1,1), 64))
        rop:   2^96.4
          m:      257
        red:   2^96.4
    delta_0: 1.006998
       beta:      168
     repeat:   2^59.0
          d:      513
          c:   51.065
```

estimator.enumeration_cost

estimator.**enumeration_cost**(*n*, *alpha*, *q*, *success_probability*, *delta_0*, *m*, *clocks_per_enum=35119.872820389246*)

Estimates the cost of performing enumeration.

Parameters

- **n** – LWE dimension $n > 0$
- **alpha** – noise rate $0 < \alpha < 1$, noise will have standard deviation $\alpha q/\sqrt{2\pi}$
- **q** – modulus $0 < q$
- **success_probability** – target success probability
- **delta_0** – root-Hermite factor $\delta_0 > 1$
- **m** – number of LWE samples $m > 0$
- **clocks_per_enum** – the log of the number of clock cycles needed per enumeration

estimator.estimate_lwe

estimator.**estimate_lwe**(*n*, *alpha=None*, *q=None*, *secret_distribution=True*, *m=<Mock name='mock()' id='139783166714704'>*, *reduction_cost_model=<function CheNgu12>*, *skip=('mitm', 'arora-gb', 'bkw')*)

Highlevel-function for estimating security of LWE parameter sets

Parameters

- **n** – LWE dimension $n > 0$
- **alpha** – noise rate $0 < \alpha < 1$, noise will have standard deviation $\alpha q/\sqrt{2\pi}$
- **q** – modulus $0 < q$
- **m** – number of LWE samples $m > 0$
- **secret_distribution** – distribution of secret, see module level documentation for details
- **reduction_cost_model** – use this cost model for lattice reduction
- **skip** – skip these algorithms

EXAMPLE:

```
sage: from estimator import estimate_lwe, Param, BKZ
sage: d = estimate_lwe(*Param.Regev(128))
usvp: rop:  2^57.7,  red:  2^57.7,  δ_0: 1.009214,  β:  102,  d:  357,  m:
↪228
 dec: rop:  2^61.5,  m:      229,  red:  2^61.5,  δ_0: 1.009595,  β:   93,  d:
↪357,  babai:  2^46.8,  ...
dual: rop:  2^81.4,  m:      376,  red:  2^81.4,  δ_0: 1.008810,  β:  111,  d:
↪376,  |v|:  736.521,  ...

sage: d = estimate_lwe(**Param.LindnerPeikert(256, dict=True))
usvp: rop: 2^137.8,  red: 2^137.8,  δ_0: 1.005788,  β:  229,  d:  594,  m:
↪337
 dec: rop: 2^142.9,  m:      334,  red: 2^142.9,  δ_0: 1.006061,  β:  212,  d:
↪590,  babai: 2^128.5,  ...
```

```
dual: rop: 2^166.0,  m:      368,  red: 2^166.0,  δ_0: 1.005479,  β:  249, ␣
↪repeat: 2^131.0,  d:  624,  ...

sage: d = estimate_lwe(*Param.LindnerPeikert(256), secret_distribution=(-1,1))
usvp: rop: 2^103.2,  red: 2^103.2,  δ_0: 1.006744,  β:  179,  d:  506,  m:  ␣
↪249
 dec: rop: 2^142.9,  m:      334,  red: 2^142.9,  δ_0: 1.006061,  β:  212,  d: ␣
↪590,  babai: 2^128.5,  ...
dual: rop: 2^112.3,  m:      268,  red: 2^112.3,  δ_0: 1.006445,  β:  192, ␣
↪repeat:  2^76.5,  d:  508,  ...

sage: d = estimate_lwe(*Param.LindnerPeikert(256), secret_distribution=(-1,1),␣
↪reduction_cost_model=BKZ.sieve)
usvp: rop:  2^80.7,  red:  2^80.7,  δ_0: 1.006744,  β:  179,  d:  506,  m:  ␣
↪249
 dec: rop: 2^111.8,  m:      369,  red: 2^111.8,  δ_0: 1.005423,  β:  253,  d: ␣
↪625,  babai:  2^97.0,  ...
dual: rop:  2^90.6,  m:      284,  red:  2^90.6,  δ_0: 1.006065,  β:  212, ␣
↪repeat:  2^53.5,  d:  524,  ...

sage: d = estimate_lwe(n=100, alpha=8/2^20, q=2^20, skip="arora-gb")
mitm: rop: 2^329.2,  m:       23,  mem: 2^321.5
usvp: rop:  2^32.0,  red:  2^32.0,  δ_0: 1.013310,  β:   40,  d:  141,  m:  ␣
↪40
 dec: rop:  2^33.7,  m:      156,  red:  2^33.7,  δ_0: 1.021398,  β:   40,  d: ␣
↪256,  babai:        1, ...
dual: rop:  2^35.3,  m:      311,  red:  2^35.3,  δ_0: 1.014423,  β:   40,  d: ␣
↪311,  |v|:  2^12.9,  ...
 bkw: rop:  2^56.8,  m:  2^43.5,  mem:  2^44.5,  b:   2,  t1:   5,  t2:  18,  l: ␣
↪ 1,  ncod:  84,  ...
```

estimator.gb_cost

`estimator.` **`gb_cost`** (*n*, *D*, *omega=2*)

Estimate the complexity of computing a Gröbner basis.

Parameters

- **`n`** – number of variables *n > 0*
- **`D`** – tuple of *(d,m)* pairs where *m* is number polynomials and *d* is a degree
- **`omega`** – linear algebra exponent, i.e. matrix-multiplication costs *O(n^ω)* operations.

estimator.guess_and_solve

`estimator.` **`guess_and_solve`** (*f*, *n*, *alpha*, *q*, *secret_distribution*, *success_probability=0.99*, ***kwds*)

Guess components of the secret.

Parameters

- **`f`** –
- **`n`** – LWE dimension *n > 0*
- **`alpha`** – noise rate *0 α < 1*, noise will have standard deviation *αq/sqrt{2π}*

- **q** – modulus $0 < q$
- **secret_distribution** – distribution of secret, see module level documentation for details
- **success_probability** – targeted success probability < 1

EXAMPLE:

```
sage: from estimator import guess_and_solve, dual_scale, partial
sage: q = next_prime(2^30)
sage: n, alpha = 512, 8/q
sage: dualg = partial(guess_and_solve, dual_scale)
sage: dualg(n, alpha, q, secret_distribution=((-1,1), 64))
        rop:   2^64.1
          m:       530
        red:   2^64.1
    delta_0: 1.008803
       beta:       111
     repeat:   2^21.6
          d:      1042
          c:     9.027
          k:         0
```

estimator.lattice_reduction_cost

estimator. **lattice_reduction_cost** (*cost_model*, *delta_0*, *d*, *B=None*)

Return cost dictionary for returning vector of norm‘ δ_0^d Vol(Λ)^{1/d}‘ using provided lattice reduction algorithm.

Parameters

- **lattice_reduction_estimate** –
- **delta_0** – root-Hermite factor $\delta_0 > 1$
- **d** – lattice dimension
- **B** – bit-size of entries

estimator.lattice_reduction_opt_m

estimator. **lattice_reduction_opt_m** (*n*, *q*, *delta*)

Return the (heuristically) optimal lattice dimension m

Parameters

- **n** – LWE dimension $n > 0$
- **q** – modulus $0 < q$
- **delta** – root Hermite factor δ_0

estimator.mitm

estimator.**mitm**(*n*, *alpha*, *q*, *secret_distribution=True*, *m=<Mock name='mock()' id='139783166714704'>*, *success_probability=0.99*)

Meet-in-the-Middle attack as described in *[AlbPlaSco15]*

Parameters

- **n** – LWE dimension $n > 0$
- **alpha** – noise rate $0 < \alpha < 1$, noise will have standard deviation $\alpha q/\sqrt{2\pi}$
- **q** – modulus $0 < q$
- **secret_distribution** – distribution of secret, see module level documentation for details
- **m** – number of LWE samples $m > 0$
- **success_probability** – targeted success probability < 1

estimator.primal_usvp

estimator.**primal_usvp**(*n*, *alpha*, *q*, *secret_distribution=True*, *m=<Mock name='mock()' id='139783166714704'>*, *success_probability=0.99*, *reduction_cost_model=<function CheNgu12>*, ***kwds*)

Estimate cost of solving LWE using primal attack (uSVP version)

Parameters

- **n** – LWE dimension $n > 0$
- **alpha** – noise rate $0 < \alpha < 1$, noise will have standard deviation $\alpha q/\sqrt{2\pi}$
- **q** – modulus $0 < q$
- **secret_distribution** – distribution of secret, see module level documentation for details
- **m** – number of LWE samples $m > 0$
- **success_probability** – targeted success probability < 1
- **reduction_cost_model** – cost model for lattice reduction

EXAMPLES:

```
sage: from estimator import primal_usvp, Param, BKZ
sage: n, alpha, q = Param.Regev(256)

sage: primal_usvp(n, alpha, q)
            rop:  2^158.6
            red:  2^158.6
        delta_0: 1.005374
           beta:      257
              d:      704
              m:      447

sage: primal_usvp(n, alpha, q, secret_distribution=True, m=n)
            rop:  2^208.1
            red:  2^208.1
```

```
        delta_0: 1.004628
           beta:      321
              d:      513
              m:      256

sage: primal_usvp(n, alpha, q, secret_distribution=False, m=2*n)
            rop:  2^208.1
            red:  2^208.1
        delta_0: 1.004628
           beta:      321
              d:      513
              m:      512

sage: primal_usvp(n, alpha, q, reduction_cost_model=BKZ.sieve)
        rop:  2^103.9
        red:  2^103.9
    delta_0: 1.005374
       beta:      257
          d:      704
          m:      447

sage: primal_usvp(n, alpha, q)
            rop:  2^158.6
            red:  2^158.6
        delta_0: 1.005374
           beta:      257
              d:      704
              m:      447

sage: primal_usvp(n, alpha, q, secret_distribution=(-1,1), m=n)
            rop:   2^88.5
            red:   2^88.5
        delta_0: 1.007317
           beta:      156
              d:      492
              m:      235

sage: primal_usvp(n, alpha, q, secret_distribution=((-1,1), 64))
            rop:   2^80.0
            red:   2^80.0
        delta_0: 1.007723
           beta:      142
              d:      461
              m:      204
```

estimator.reduction_default_cost

estimator. **reduction_default_cost** (*beta*, *d*, *B=None*)

Runtime estimation given β and assuming *[CheNgu12]* estimates are correct.

Parameters

- **beta** – block size
- **n** – LWE dimension $n > 0$
- **B** – bit-size of entries

The constants in this function were derived as follows based on Table 4 in *[CheNgu12]*:

```
sage: dim = [100, 110, 120, 130, 140, 150, 160, 170, 180, 190, 200, 210, 220,␣
↪230, 240, 250]
sage: nodes = [39.0, 44.0, 49.0, 54.0, 60.0, 66.0, 72.0, 78.0, 84.0, 96.0, 99.0,␣
↪105.0, 111.0, 120.0, 127.0, 134.0]  # noqa
sage: times = [c + log(200,2).n() for c in nodes]
sage: T = zip(dim, nodes)
sage: var("a,b,c,k")
(a, b, c, k)
sage: f = a*k*log(k, 2.0) + b*k + c
sage: f = f.function(k)
sage: f.subs(find_fit(T, f, solution_dict=True))
k |--> 0.270188776350190*k*log(k) - 1.0192050451318417*k + 16.10253135200765
```

The estimation

2^(0.270188776350190*beta*log(beta) - 1.0192050451318417*beta + 16.10253135200765)

is of the number of enumeration nodes, hence we need to multiply by the number of cycles to process one node. This cost per node is typically estimated as 100 [FPLLL].

estimates (Full Version). 2012. http://www.di.ens.fr/~ychen/research/Full_BKZ.pdf

Available at https://github.com/fplll/fplll

estimator.rinse_and_repeat

estimator. **rinse_and_repeat** (*f*, *n*, *alpha*, *q*, *success_probability=0.99*, *m=<Mock name='mock()' id='139783166714704'>*, *optimisation_target=u'red'*, *decision=True*, *repeat_select=None*, **args*, ***kwds*)

Find best trade-off between success probability and running time.

Parameters

- **f** – a function returning a cost estimate
- **n** – LWE dimension $n > 0$
- **alpha** – noise rate $0 < \alpha < 1$, noise will have standard deviation $\alpha q/sqrt{2\pi}$
- **q** – modulus $0 < q$
- **success_probability** – targeted success probability < 1
- **optimisation_target** – which value to minimise
- **decision** – set if f solves a decision problem, unset for search problems
- **repeat_select** – passed through to `cost_repeat` as parameter `select`
- **samples** – the number of available samples

estimator.sieve_or_enum

estimator. **sieve_or_enum** (*func*)

Take minimum of sieving or enumeration for lattice-based attacks.

Parameters **func** – a lattice-reduction based estimator

estimator.sigmaf

estimator. **sigmaf** (*stddev*)

standard deviation → Gaussian width parameter σ

Parameters **sigma** – standard deviation

EXAMPLE:

```
sage: from estimator import stddevf, sigmaf
sage: n = 64.0
sage: sigmaf(stddevf(n))
64.000...
```

estimator.stddevf

estimator. **stddevf** (*sigma*)

Gaussian width parameter σ → standard deviation

Parameters **sigma** – Gaussian width parameter σ

EXAMPLE:

```
sage: from estimator import stddevf
sage: n = 64.0
sage: stddevf(n)
25.532...
```

estimator.success_probability_drop

estimator. **success_probability_drop** (*n*, *h*, *k*, *fail=0*, *rotations=False*)

Probability that k randomly sampled components have `fail` non-zero components amongst them.

Parameters

- **n** – LWE dimension $n > 0$
- **h** – number of non-zero components
- **k** – number of components to ignore
- **fail** – we tolerate `fail` number of non-zero components amongst the k ignored components
- **rotations** – consider rotations of the basis to exploit ring structure (NTRU only)

estimator.switch_modulus

estimator. **switch_modulus** (*f*, *n*, *alpha*, *q*, *secret_distribution*, **args*, ***kwds*)

Parameters

- **f** – run f
- **n** – LWE dimension $n > 0$

- **alpha** – noise rate *0 α < 1*, noise will have standard deviation $\alpha q/sqrt\{2\pi\}$
- **q** – modulus *0 < q*
- **secret_distribution** – distribution of secret, see module level documentation for details

CHAPTER 4

Index

Bibliography

[ADPS16] Edem Alkim, Léo Ducas, Thomas Pöppelmann, & Peter Schwabe (2016). Post-quantum key exchange - A New Hope. In T. Holz, & S. Savage, 25th USENIX Security Symposium, USENIX Security 16 (pp. 327–343). USENIX Association.

[BDGL16] Becker, A., Ducas, L., Gama, N., & Laarhoven, T. (2016). New directions in nearest neighbor searching with applications to lattice sieving. In SODA 2016, (pp. 10–24).

[CheNgu12] Yuanmi Chen and Phong Q. Nguyen. BKZ 2.0: Better lattice security

[FPLLL] The FPLLL development team. fplll, a lattice reduction library. 2016.

[CheNgu11] Chen, Y., & Nguyen, P. Q. (2011). BKZ 2.0: better lattice security estimates. In D. H. Lee, & X. Wang, ASIACRYPT~2011 (pp. 1–20). : Springer, Heidelberg.

[LaaMosPol14] Thijs Laarhoven, Michele Mosca, & Joop van de Pol. Finding shortest lattice vectors faster using quantum search. Cryptology ePrint Archive, Report 2014/907, 2014. https://eprint.iacr.org/2014/907.

[Laarhoven15] Laarhoven, T. (2015). Search problems in cryptography: from fingerprinting to lattice sieving (Doctoral dissertation). Eindhoven University of Technology. http://repository.tue.nl/837539

[LinPei11] Lindner, R., & Peikert, C. (2011). Better key sizes (and attacks) for LWE-based encryption. In A. Kiayias, CT-RSA~2011 (pp. 319–339). : Springer, Heidelberg.

[CheNgu12] Yuanmi Chen and Phong Q. Nguyen. BKZ 2.0: Better lattice security

[FPLLL] The FPLLL development team. fplll, a lattice reduction library. 2016.

[LinPei11] Lindner, R., & Peikert, C. (2011). Better key sizes (and attacks) for LWE-based encryption. In A. Kiayias, CT-RSA~2011 (pp. 319–339). : Springer, Heidelberg.

[LaaMosPol14] Thijs Laarhoven, Michele Mosca, & Joop van de Pol. Finding shortest lattice vectors faster using quantum search. Cryptology ePrint Archive, Report 2014/907, 2014. https://eprint.iacr.org/2014/907.

[Laarhoven15] Laarhoven, T. (2015). Search problems in cryptography: from fingerprinting to lattice sieving (Doctoral dissertation). Eindhoven University of Technology. http://repository.tue.nl/837539

[BDGL16] Becker, A., Ducas, L., Gama, N., & Laarhoven, T. (2016). New directions in nearest neighbor searching with applications to lattice sieving. In SODA 2016, (pp. 10–24).

[ACFP14] Albrecht, M. R., Cid, C., Jean-Charles Faug‘ere, & Perret, L. (2014). Algebraic algorithms for LWE.

[AroGe11] Arora, S., & Ge, R. (2011). New algorithms for learning in presence of errors. In L. Aceto, M. Henzinger, & J. Sgall, ICALP 2011, Part~I (pp. 403–415). : Springer, Heidelberg.

[GuoJohSta15] Guo, Q., Johansson, T., & Stankovski, P. (2015). Coded-BKW: solving LWE using lattice codes. In R. Gennaro, & M. J. B. Robshaw, CRYPTO~2015, Part~I (pp. 23–42). : Springer, Heidelberg.

[Albrecht17] Albrecht, M. R. (2017). On dual lattice attacks against small-secret LWE and parameter choices in helib and SEAL. In J. Coron, & J. B. Nielsen, EUROCRYPT 2017, Part II (pp. 103–129).

[Albrecht17] Albrecht, M. R. (2017). On dual lattice attacks against small-secret LWE and parameter choices in helib and SEAL. In J. Coron, & J. B. Nielsen, EUROCRYPT} 2017, Part {II (pp. 103–129).

[AlbPlaSco15] Albrecht, M. R., Player, R., & Scott, S. (2015). On the concrete hardness of Learning with Errors. Journal of Mathematical Cryptology, 9(3), 169–203.

[USENIX:ADPS16] Alkim, E., Léo Ducas, Thomas Pöppelmann, & Schwabe, P. (2015). Post-quantum key exchange - a new hope.

[BaiGal14] Bai, S., & Galbraith, S. D. (2014). Lattice decoding attacks on binary LWE. In W. Susilo, & Y. Mu, ACISP 14 (pp. 322–337). : Springer, Heidelberg.

[CheNgu12] Yuanmi Chen and Phong Q. Nguyen. BKZ 2.0: Better lattice security

[FPLLL] The FPLLL development team. fplll, a lattice reduction library. 2016.

e

Symbols

A

B

C

D

E

F

G

H

I

K

L