DGHV, AGCD

Sam Hadow

December 14, 2025

Contents

1 Git repositories

- `https://git.hadow.fr/sam.hadow/dghv-python3`
  DGHV original sage implementation ported from Python2 to Python3.
- `https://git.hadow.fr/sam.hadow/homomorphic-encryption`
  DGHV Rust implementation.
- `https://git.hadow.fr/sam.hadow/approximate-gcd`
  AGCD solver using LLL (L^2 variant) in Rust.

2 AGCD (Approximate Greatest Common Divisor)

2.1 problem description

2.1.1 GCD problem:

We can solve the GCD problem in polynomial time. $\mathcal{O}(n^2)$ Using the Euclidian algorithm (with n the bitlength of $\max(a, b)$):

Algorithm 1 GCD using Euclidean Algorithm

```
function GCD(a, b)
    if b = 0 then
        return a
    else
        return GCD(b, a mod b)
    end if
end function
```

Or in $\mathcal{O}(M(n)\log n)$ using the binary GCD algorithm (best known algorithm) , with $M(n)$ the time complexity of the algorithm used for the multiplication of two n-bits integers. So the total complexity would be approximately $O(n\log^2 n)$ using Fürer's algorithm. [SZ04] [Für07]

Thus GCD is easy to compute.

2.1.2 AGCD problem

As soon as we introduce randomness with r_i the GCD problem becomes the AGCD problem which is hard (as long as we don't find an efficient algorithm to solve it).

The AGCD consists in finding the largest integer p such as p divides a set of integer $(x_1, x_2, \ldots, x_n)$ with
$x_i = q_i \cdot p + r_i$
r_i being a random noise
$p \mid \gcd(x_1 - r_1, x_2 - r_2, \ldots, x_n - r_n)$

2.2 Solving AGCD

2.2.1 brute force

First let's try with exhaustive search.

$x_0 - r_0 = q_0 \cdot p$ and $x_1 - r_1 = q_1 \cdot p$
Therefore $p|\gcd(x_0 - r_0, x_1 - r_1)$

With a bound 2^λ for the noise (λ being the security parameter in DGHV).
Thus we can guess r_0 and r_1 with $2^{2\lambda}$ GCD computation.

Using a time-memory trade-off [CNT12]:
With n near-multiples $x_j = p \cdot q_j + r_j$, with $q_j \in \left[0, \frac{2^\gamma}{p}\right)$, $r_j \in \left[0, 2^\lambda\right)$:
(γ being the bit length of the x_j)

1. Compute
$$y_j = \prod_{i=0}^{2^\lambda - 1} (x_j - i)$$
for $j = 1, \cdots, n$, using a product tree in $\tilde{O}(2^\lambda)$ time for each y_j. Since $x_j = r_j \pmod p$, $\forall j, p|y_j$.
2. Compute $g = \gcd(y_1, \ldots, y_n)$ with $n-1$ GCD computations on numbers of size $\tilde{O}(2^\lambda)$, in $\tilde{O}(n \cdot 2^\lambda)$ time.
3. With an overwhelming probability, $g = p \cdot k$, where all prime factors of k are smaller than a bound $B \approx 2^{\frac{n+1}{n-1}\lambda}$.
4. Compute the B-smooth part g' of g in $\tilde{O}(B)$ time and recover $p = g/g'$.

The attack achieves time complexity $\tilde{O}\left(2^{\frac{n+1}{n-1}\rho}\right)$, approaching $\tilde{O}(2^\rho)$ with enough samples, and memory complexity $\tilde{O}(2^\lambda)$.

So even with a time memory trade-off such an attack still has a time complexity of $\widetilde{O}(2^\lambda)$ and is not realistic with enough noise.

We also deduce the bit-length needed for the noise r_i in DGHV to prevent brute force attacks on the noise. at least 72 bits against civilian adversaries, at least 90 bits against government-level adversaries and at least 128 bits for long term security.

2.2.2 Lattice attack

Let's say we have $x_0 = q_0 \cdot p + r_0$ and $x_1 = q_1 \cdot p + r_1$, we can write:

$$\begin{aligned} q_0 \cdot x_1 - q_1 \cdot x_0 &= q_0 \cdot (q_1 \cdot p + r_1) - q_1 \cdot (q_0 \cdot p + r_0) \\ &= q_0 \cdot q_1 \cdot p + q_0 \cdot r_1 - q_1 \cdot q_0 \cdot p - q_1 \cdot r_0 \\ &= q_0 \cdot r_1 - q_1 \cdot r_0 \end{aligned}$$

And we also know that $q_0 \cdot x_1 - q_1 \cdot x_0$ is much smaller than any x_i because r_i are small noise terms, while x_i are dominated by $q_i \cdot p$. The issue with only 2 terms is we don't know q_0 and q_1.
Let's generalise in a matrix form this observation but with n samples instead of just 2:

$$\mathbf{M} = \begin{pmatrix} 2^{\lambda+1} & x_1 & x_2 & \cdots & x_n \\ & -x_0 & & & \\ & & -x_0 & & \\ & & & \ddots & \\ & & & & -x_0 \end{pmatrix}$$

Let's also write a vector $\mathbf{q} = (q_0, q_1, q_2, \cdots, q_n)$
We can then write:

$$\begin{aligned} q \cdot M &= (q_0, q_1, \cdots, q_t) \cdot M \\ &= (q_0 \cdot 2^{\lambda+1}, q_0 \cdot x_1 - q_1 \cdot x_0, \cdots, q_0 \cdot x_n - q_n \cdot x_0) \\ &= (q_0 \cdot 2^{\lambda+1}, q_0 \cdot r_1 - q_1 \cdot r_0, \cdots, q_0 \cdot r_n - q_n \cdot r_0) \end{aligned}$$

Let's define the lattice:

$$\mathcal{L} = \left\{ \mathbf{q} \cdot \mathbf{M} \;\middle|\; \mathbf{q} \in \mathbb{Z}^{n+1} \right\}$$

This lattice contains the set of all integer-linear combinations of the rows of the matrix M. In particular it contains the vector:

$$(q_0, q_1, \cdots, q_n) \cdot \mathbf{M} = \left(q_0 \cdot 2^{\lambda+1},\ q_0 r_1 - q_1 r_0,\ \cdots,\ q_0 \cdot r_n - q_n \cdot r_0 \right)$$

Which is exactly the vector we're looking for, and is a vector with small coefficients compared to every x_i. Therefore we can reduce the AGCD (Approximate Greatest Common Divisor) problem to an instance of the SVP (Shortest Vector Problem) in a lattice. And finding short vectors in lattices is assumed to be hard as we didn't find an efficient algorithm to solve this problem yet. DGHV security is based on the fact that AGCD problem is a computationally hard problem.

From there to find p, we can extract the first element from the shortest vector, find q_0, and then we can write:
$x_0 = p \cdot q_0 + r_0$, therefore $r_0 = x_0 \mod q_0$ and $p = \frac{x_0 - r_0}{q_0}$

2.3 SVP (Shortest Vector Problem)

Now that we reduced the AGCD problem to an instance of SVP, how do we solve this problem? Unfortunately SVP is not known to be NP-complete, and for specific scenarios, the exact version of this problem is NP-Hard. [Din02] [Ajt98]

However in practice, we can solve an approximation of SVP. We have 2 algorithms for this:

- LLL (Lenstra-Lenstra-Lovas) Algorithm [LLL82]
- BKZ (Block Korkine-Zolotarev) Algorithm [Yas21]

2.3.1 LLL (Lenstra-Lenstra-Lovas) Algorithm

LLL Algorithm given a basis for a lattice (M in our case) returns a reduced basis where the first vector is relatively short and nearly orthogonal to the rest. This first vector is likely (but not guaranteed) to be the one we're looking for (from which we can extract $q_0 \cdot 2^{\lambda+1}$).
LLL is a heuristic algorithm and therefore we don't have any guarantee to solve the initial AGCD problem every time. But from empirical observations, we get "good enough" results against DGHV with "bad" security parameters.

Algorithm 2 original LLL Algorithm

Require: A basis $(b_1, \ldots, b_d)$ and $\delta \in \left(\frac{1}{4}, 1\right)$
Ensure: An LLL-reduced basis with factor $(\delta, 1/2)$
Compute the rational Gram-Schmidt Orthogonalization: all the $\mu_{i,j}$ and $r_{i,i}$
$\kappa \leftarrow 2$
while $\kappa \leq d$ **do**
    Size-reduce the vector b_κ using the size-reduction algorithm (Alg. 3), which updates the Gram-Schmidt Orthogonalization
    $\kappa' \leftarrow \kappa$
    while $\kappa \geq 2$ **and** $\delta \cdot r_{\kappa-1,\kappa-1} > r_{\kappa',\kappa'} + \sum_{i=\kappa-1}^{\kappa'-1} \mu_{\kappa',i}^2 r_{i,i}$ **do**
        $\kappa \leftarrow \kappa - 1$
    end while
    Insert the vector $b_{\kappa'}$ right before b_κ and update the Gram-Schmidt Orthogonalization accordingly
    $\kappa \leftarrow \kappa + 1$
end while
return $(b_1, \ldots, b_d)$

Algorithm 3 Size-Reduction Algorithm

Require: A basis $(b_1, \ldots, b_d)$, its Gram-Schmidt Orthogonalization, and an index κ
Ensure: The basis where b_κ is size-reduced and the updated Gram-Schmidt Orthogonalization
for $i = \kappa - 1$ down to 1 **do**
    $b_\kappa \leftarrow b_\kappa - \mu_{\kappa,i} \cdot b_i$
end for
Update the Gram-Schmidt Orthogonalization accordingly

Here we don't use the original LLL algorithm but the $\mathbf{L^2}$ algorithm. [NS09]
With the LLL algorithm the worst time complexity when using only integers makes it problematic on large lattices (which can be the case with AGCD), variants of the original LLL algorithm exist, like using floating point arithmetic instead of big integer arithmetic. However due to rounding errors the algorithm becomes unstable and might not converge.
Instead the $\mathbf{L^2}$ algorithm is guaranteed to terminate in polynomial time while still giving a LLL-reduced base (same quality of reduction as with the original LLL). It uses a Classical Floating-point Algorithm more numerically stable, and a lazy size reduction to avoid amplifying rounding errors. Below are the algorithms used:

Algorithm 4 $\mathbf{L^2}$ Algorithm

Require: A valid pair (δ, η) (as in Theorem 2), a basis $(b_1, \ldots, b_d)$, and an fp-precision $p + 1$
Ensure: An L3-reduced basis with factor pair (δ, η)
    Variables: An integral matrix G, floating-point numbers $\bar{r}_{i,j}$, $\bar{\mu}_{i,j}$, and $\bar{s}_i$
Compute exactly $G = G(b_1, \ldots, b_d)$
$\bar{\delta} \leftarrow ((\delta + 1)/2)$, $\bar{r}_{1,1} \leftarrow \langle b_1, b_1 \rangle$, $\kappa \leftarrow 2$
while $\kappa \leq d$ **do**
    Size-reduce the vector b_κ using (Alg. 5), updating the fp-Gram-Schmidt Orthogonalization
    $\kappa' \leftarrow \kappa$
    while $\kappa \geq 2$ **and** $(\bar{\delta} \cdot \bar{r}_{\kappa-1,\kappa-1}) > \bar{s}_{\kappa-1}^{(\kappa')}$ **do**
        $\kappa \leftarrow \kappa - 1$
    end while
    for $i = 1$ **to** $\kappa - 1$ **do**
        $\bar{\mu}_{\kappa,i} \leftarrow \bar{\mu}_{\kappa',i}$, $\bar{r}_{\kappa,i} \leftarrow \bar{r}_{\kappa',i}$, $\bar{r}_{\kappa,\kappa} \leftarrow \bar{s}_\kappa^{(\kappa')}$
    end for
    Insert the vector $b_{\kappa'}$ right before b_κ and update G accordingly
    $\kappa \leftarrow \kappa + 1$
end while
return $(b_1, \ldots, b_d)$

Algorithm 5 Lazy Size-Reduction Algorithm ($\mathbf{L^2}$)

Require: A factor $\eta > \frac{1}{2}$, an fp-precision $\ell + 1$, an index κ, a basis $(\mathbf{b}_1, \ldots, \mathbf{b}_d)$ with its Gram matrix, and floating-point numbers $\bar{r}_{i,j}$ and $\bar{\mu}_{i,j}$ for $j \leq i < \kappa$

Ensure: Floating-point numbers $\bar{r}_{\kappa,j}$, $\bar{\mu}_{\kappa,j}$, and $\bar{s}_j^{(\kappa)}$ for $j \leq \kappa$, and a basis $(\mathbf{b}_1, \ldots, \mathbf{b}_{\kappa-1}, \mathbf{b}'_\kappa, \mathbf{b}_{\kappa+1}, \ldots, \mathbf{b}_d)$ with its Gram matrix, where:
$\mathbf{b}'_\kappa = \mathbf{b}_\kappa - \sum_{i<\kappa} x_i \cdot \mathbf{b}_i$ for some $x_i \in \mathbb{Z}$ and $|\langle \mathbf{b}'_\kappa, \mathbf{b}_i^* \rangle| \leq \eta \|\mathbf{b}_i^*\|^2$ for any $i < \kappa$

1: $\bar{\eta} \leftarrow \diamond\left(\diamond(\eta) + \frac{1}{2}\right)/2$
2: Compute $\bar{r}_{\kappa,j}$, $\bar{\mu}_{\kappa,j}$, $\bar{s}_j^{(\kappa)}$ using a floating point approximation with $i = \kappa$ in (Alg. 6)
3: **if** $\max_{j<\kappa} |\bar{\mu}_{\kappa,j}| \leq \bar{\eta}$ **then**
4: **return**
5: **else**
6: **for** $i = \kappa - 1$ **down to** 1 **do**
7: $X_i \leftarrow \bar{\mu}_{\kappa,i}$
8: **for** $j = 1$ **to** $i - 1$ **do**
9: $\bar{\mu}_{\kappa,j} \leftarrow \bar{\mu}_{\kappa,j} - (X_i \cdot \bar{\mu}_{i,j})$
10: **end for**
11: **end for**
12: $\mathbf{b}_\kappa \leftarrow \mathbf{b}_\kappa - \sum_{i=1}^{\kappa-1} X_i \cdot \mathbf{b}_i$, update $G(\mathbf{b}_1, \ldots, \mathbf{b}_d)$ accordingly
13: **Goto** step 2
14: **end if**

Orthogonalization is performed vector by vector with the following algorithm:

Algorithm 6 Gram-Schmidt Orthogonalization Algorithm

Require: Gram matrix of $(b_1, \ldots, b_d)$
Ensure: A vector as defined by the Gram-Schmidt Orthogonalization
1: **for** $j = 1$ to $i - 1$ **do**
2: $r_{i,j} \leftarrow \langle b_i, b_j \rangle$
3: **for** $k = 1$ to $j - 1$ **do**
4: $r_{i,j} \leftarrow r_{i,j} - \mu_{j,k} \cdot r_{i,k}$
5: **end for**
6: $\mu_{i,j} \leftarrow \frac{r_{i,j}}{r_{j,j}}$
7: **end for**
8: $s_1^{(i)} \leftarrow \|b_i\|^2$
9: **for** $j = 2$ to i **do**
10: $s_j^{(i)} \leftarrow s_{j-1}^{(i)} - \mu_{i,j-1} \cdot r_{i,j-1}$
11: **end for**
12: $r_{i,i} \leftarrow s_i^{(i)}$

We have 2 parameters in $\mathbf{L^2}$:

- $\delta \in (0.25, 1)$: Lovász parameter, or reduction quality constant.
- $\eta \in \left(\frac{1}{2}, \sqrt{\delta}\right)$: the size reduction parameter.

$\mathbf{L^2}$ is guaranteed to run in polynomial time with these bounds for the parameters. The proven time complexity is $O\big(d^2 n(d + \log B) \log B \cdot M(d)\big)$. with:

- d: dimension of the lattice
- n: dimension of a vector (all the vectors are in $\mathbb{R}^n$)
- $\log B$: the bit-length of the input vectors used
- $M(d)$: the time complexity to multiply two d-bit numbers
  There is a $M(d)$ term and not a $M(\log B)$ term in the complexity because the operations are mostly between two coefficients (bit-length d), or a coefficient and a vector (which corresponds to multiple operations of complexity $O\big(M(d)\big)$), the costly multiplications happen when initializing the Gram matrix and are therefore much rarer.

We also deduce from the time complexity of a $\mathbf{L^2}$ reduction the key-length which should be used in DGHV.
The noise bit-length γ must be large enough to hide the secret p against lattice-based AGCD attacks like the one here. Intuitively, if the noise is too small, lattice algorithms can efficiently find short vectors revealing p.
The complexity of lattice reduction grows roughly quadratically in the lattice dimension d and in $\log B$, where $B \sim 2^\gamma$, which implies $\log B = \gamma$.
To achieve λ-bit security, lattice reduction should require time at least 2^λ.
Since the lattice dimension d grows roughly like η, and bit-length $B \sim 2^\gamma$, the time complexity $O\left(d^2 n(d+\gamma)\gamma \cdot M(d)\right)$ should be at least quadratic in η.
This implies $\gamma \geq (\eta^2 \log \lambda)$ to prevent lattice-based attacks.

The parameter τ is the number of public key elements (which corresponds to the lattice dimension since each public key element can be used to build the initial lattice basis). Since LLL is a heuristic algorithm, one needs enough samples x_i for a successful lattice-based attack.
As mentioned in DGHV paper, the leftover hash lemma states that, to extract nearly uniform randomness, the number of samples must exceed the entropy by a margin of $\omega(\log \lambda)$. [HILL99]
Since the noise bit-length is γ, we need $\tau \geq \gamma + \omega(\log \lambda)$ to prevent lattice-based attacks exploiting a lack of randomness in the public key.

Parallelization:

Side note on parallelization. Parallelizing LLL (or $\mathbf{L^2}$ here) is unfortunately not possible here because the outer loop (while $k \leq d$) is inherently sequential, each step depending on the previous basis computed. We could parallelize the operations on the vectors (dot products, addition, multiplication by a scalar and subtraction). But the overhead of thread scheduling outweighs the gain and it turns out the reduction is slower with parallelism.
As a reference, here was the time to reduce a lattice of dimension 100 (CPU: i5-8400, 6 cores, 3.8GHz):

- with parallelization (6 cores): 1m8.282s
- without parallelization: 0m14.155s

2.3.2 BKZ (Block Korkine-Zolotarev) Algorithm

BKZ is another heuristic algorithm giving an approximate solution for the Shortest Vector Problem with better guarantees than LLL. It gives much shorter and more orthogonal vectors. However it runs in exponential time instead of polynomial time.
BKZ extends LLL by processing the lattice basis in blocks of size β. For each block, it finds the shortest vector (solving SVP) and updates the basis.
BKZ therefore requires a choice of algorithm to solve SVP on each block, preferably an exact rather than approximate SVP algorithm performing an exhaustive search to find the shortest non-zero vector (like the Schnorr–Euchner enumeration algorithm). These algorithms are however computationally expensive and the one used significantly impacts the time complexity of BKZ. But with an exact SVP solver, the total time complexity of BKZ is exponential and therefore not really possible to use in practice in a lattice-based attack.

In the code, a BKZ algorithm was implemented using $\mathbf{L^2}$ as the underlying SVP solver, but the solution's quality is the same as when using $\mathbf{L^2}$ directly and the added overhead when using BKZ makes the reduction slower (about 2 times slower). Therefore, using $\mathbf{L^2}$ as the underlying SVP solver for BKZ is not interesting.

2.4 Results

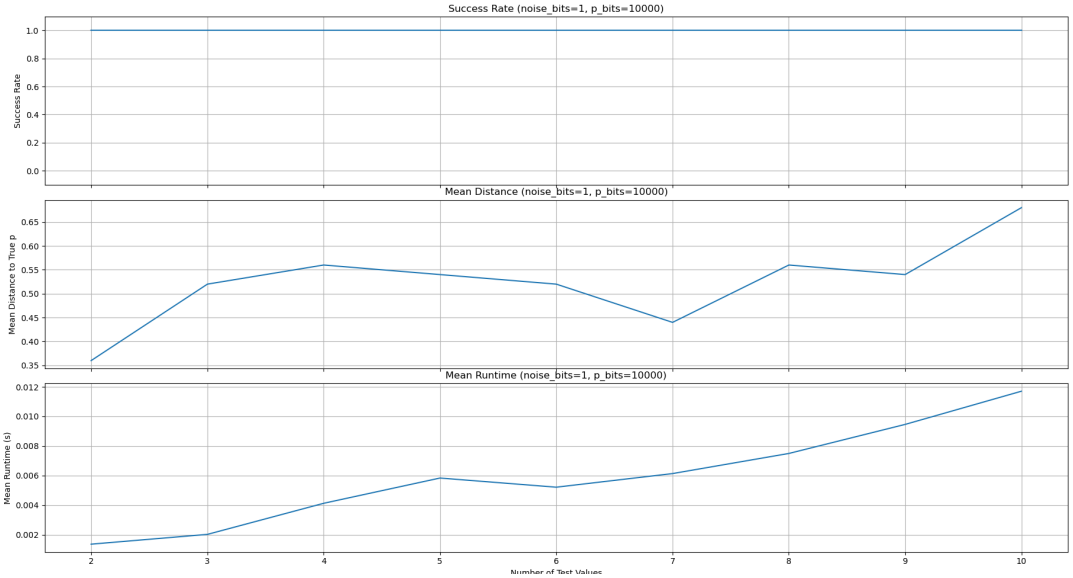

Figure 1: p 10000 bits, 1 noise bit

With no noise we can see that our lattice-based attack recover the true p $\pm$ 1 every time.

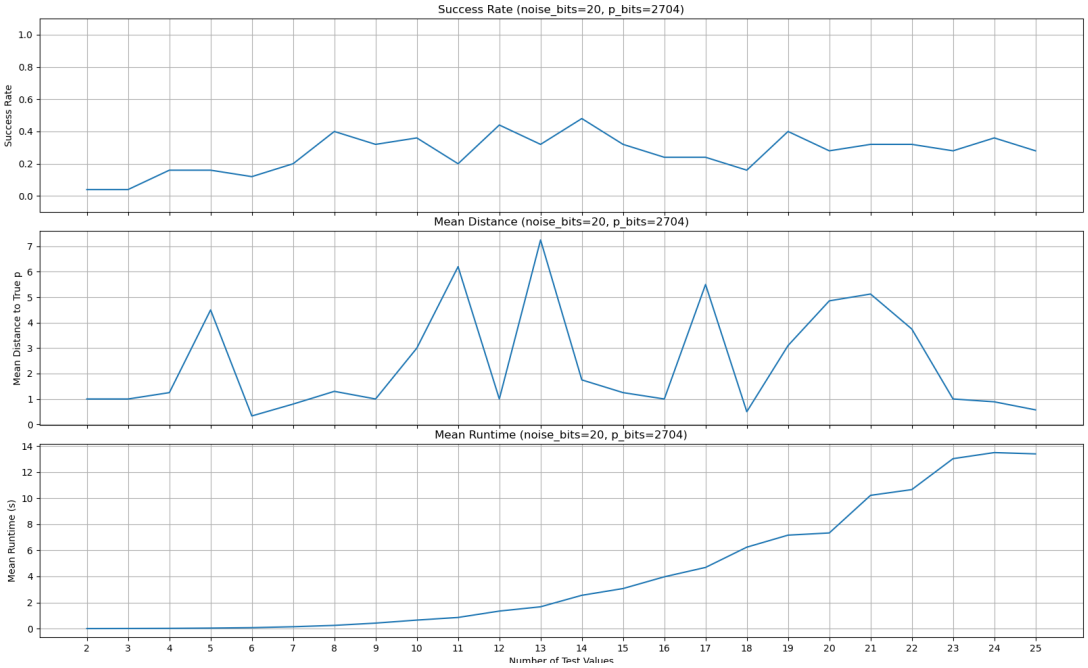

Figure 2: p 2704 bits, 20 noise bits

With a key of 2704 bits, 20 bits of noise and a few samples, we manage to recover the true p almost 40% of the time on average and with a distance to this true p of less than 10 (an exhaustive search would then be used).

With a few tests with 104 samples the attack took more than a minute but the true p was recovered (with a distance of 1 at most). The attack however quickly becomes hard to do with a lot of samples as the computation time increases quadratically. But with only 20 bits of noise, an attack is still possible in a reasonable time with a good chance of success with more than 100 samples. With enough noise (52 bits with a 2704-bits key as recommended) the attack would need much more samples and possibly take a few weeks on a personal computer to have a good chance of success.

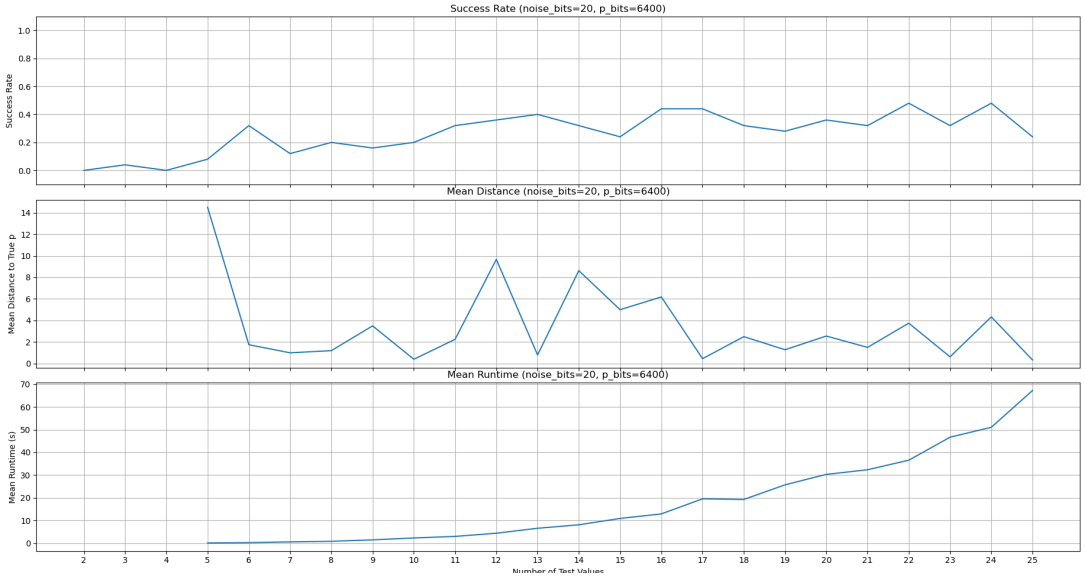

Figure 3: p 6400 bits, 20 noise bits

With 6400-bits keys and 20 bits of noise we manage to recover the key approximately as often as when the key bit-length was 2704, though the attack becomes about 5 times slower. We notice from empirical observations that the noise influence the chance of success (with the same amount of samples). And the key length doesn't directly influence the chance of success, it only makes the attack slower. But a longer key is needed when we add more noise in DGHV.

With 80 bits of noise and a key of 6400 bits (security of 80 bit), not a single success was registered even with up to 200 samples. In order to have a success we would theoretically need so many samples that the attack wouldn't be realistic on a personal computer.

With the correct parameters used for the noise we can achieve the desired level of security. And in DGHV the size of the key then depends on the noise bit-length to have a fully homomorphic encryption (it must be possible to evaluate the squashed decryption circuit homomorphically for the bootstrapping).

3 DGHV

[vDGHV10]

3.1 DGHV security parameters

λ is the security parameter. Other parameters depend on λ.
$\lambda = 80$ represents approximately 80-bits of security.

- $\rho = \omega(\log \lambda)$, noise bit-length
  Noise is necessary to protect against brute-force attacks.
- $\eta \geq \rho \cdot \Theta(\lambda \log^2 \lambda)$, private key p bit-length
  In order to have the squashed decryption circuit (bootstrapping necessary for FHE).
- $\gamma = \omega(\eta^2 \log \lambda)$, bit-length of the noise in each element of the public key
  Must be big enough to hide p and protect against lattice-based attacks (AGCD problem).
- $\tau \geq \gamma + \omega(\log \lambda)$, number of elements x_i in the public key
  In order to use the leftover hash lemma in the reduction to approximate GCD.

For the bit length of the noise q we should have $q \cdot q \geq \eta$ to hide the secret key. So we can use the same size for both noises (r and q).

Table 1: Concrete parameter sets for different security levels

security λ	Sk η (bit-length)	noise ρ (bit-length)	γ ($\times 10^6$) (bit-length)	τ Pk (elements)	Pk (size)
52	2704	52	0.83	572	59.3 MB
62	3844	62	4.20	2110	1.1 GB
72	5184	72	19.35	7659	18.5 GB
80	6400	80	35.9	28 125	126.2 GB
128	16 384	128	343	102 656	4.4 TB

As we can see from these parameter sets, DGHV is hard to use in practice if we want 128-bits of security as the public key becomes very large and hard to transfer over the network. For the ciphertexts they become about 10^4 times bigger than the cleartexts with 128-bits of security increasing storage costs in the cloud and slowing down computations. When evaluating large circuits, the bootstrapping time, especially with a big λ isn't negligible either.

3.2 Symmetric scheme

In the symmetric scheme we have the following operations:
encryption: $c = m + p \cdot q + 2 \cdot r$
decryption: $m = (c \bmod p) \bmod 2$

With:

- m: the clear bit
- c: the ciphertext
- p: the private key (odd-integer of bit-length η)
- q: a random noise (bit-length γ)
- r: a random noise (bit-length ρ)

3.3 Asymmetric scheme

In the asymmetric scheme we need a public key first. To generate a public key we use a table of τ elements. Each element x_i is a bit 0 encrypted with the private key p.
$\forall i, x_i = p \cdot q_i + 2 \cdot r_i$
And the largest element x_0 must be odd.

Encryption with a public key:

$$c = (m + 2 \cdot r + \sum_{i=1}^{\tau-1}(b_i \cdot x_i)) \bmod x0$$

With:

- x_0: The largest public element.
- $\forall i, b_i \in \{0, 1\}$, each one randomly chosen. (Boolean variable for if we use the public subkey x_i or not)

The decryption is the same operation as in the symmetric scheme.

References

[Ajt98] Miklós Ajtai. The shortest vector problem in ℓ_2 is NP-hard for randomized reductions (extended abstract). In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing (STOC '98)*, pages 10–19. ACM, 1998.

[CNT12] Jean-Sébastien Coron, David Naccache, and Mehdi Tibouchi. Public key compression and modulus switching for fully homomorphic encryption over the integers. In David Pointcheval and Thomas Johansson, editors, *Advances in Cryptology – EUROCRYPT 2012*, pages 446–464, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[Din02] Irit Dinur. Approximating svp∞ to within almost-polynomial factors is np-hard. *Theoretical Computer Science*, 285(1):55–71, 2002. Algorithms and Complexity.

[Für07] Martin Fürer. Faster integer multiplication. In *Proceedings of the 39th Annual ACM Symposium on Theory of Computing (STOC)*, pages 57–66, San Diego, California, USA, 2007. ACM.

[HILL99] Johan HÅstad, Russell Impagliazzo, Leonid A. Levin, and Michael Luby. A pseudorandom generator from any one-way function. *SIAM Journal on Computing*, 28(4):1364–1396, 1999.

[LLL82] A. K. Lenstra, H. W. Lenstra, and L. Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261:515–534, 1982.

[NS09] Phong Q. Nguyen and Damien Stehlé. An lll algorithm with quadratic complexity. *SIAM Journal on Computing*, 39(3):874–903, 2009.

[SZ04] Damien Stehlé and Paul Zimmermann. A binary recursive gcd algorithm. *LORIA/INRIA Lorraine*, 2004. `{stehle,zimmerma}@loria.fr`.

[vDGHV10] Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. Fully homomorphic encryption over the integers. In Henri Gilbert, editor, *Advances in Cryptology – EUROCRYPT 2010*, pages 24–43, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[Yas21] Masaya Yasuda. A survey of solving svp algorithms and recent strategies for solving the svp challenge. In Tsuyoshi Takagi, Masato Wakayama, Keisuke Tanaka, Noboru Kunihiro, Kazufumi Kimoto, and Yasuhiko Ikematsu, editors, *International Symposium on Mathematics, Quantum Theory, and Cryptography*, pages 189–207, Singapore, 2021. Springer Singapore.