Optimization of LPN Solving Algorithms

Sonia Bogos    Serge Vaudenay

EPFL

08 December 2016
Now Hiring!

mailto: job_lasec@epfl.ch
Now Hiring!

mailto: job_lasec@epfl.ch
Motivation

- LPN can be defined as a noisy system of linear equations in the binary domain
Motivation

- LPN can be defined as a noisy system of linear equations in the binary domain
- believed to be quantum resistant
Motivation

- LPN can be defined as a noisy system of linear equations in the binary domain
- believed to be quantum resistant
- used in authentication protocols and cryptosystems
Motivation

- LPN can be defined as a noisy system of linear equations in the binary domain
- believed to be quantum resistant
- used in authentication protocols and cryptosystems
- special case of LWE, but its hardness is not proven so far
Motivation

- LPN can be defined as a noisy system of linear equations in the binary domain
- believed to be quantum resistant
- used in authentication protocols and cryptosystems
- special case of LWE, but its hardness is not proven so far

Best way to study its hardness is by improving the algorithms that solve it
Our Results

- analyse the existing LPN algorithms and study its building blocks
- improve the theory behind the covering code reduction
- optimise the order and the parameters used in LPN solving algorithms
- improve the best existing algorithms from ASIACRYPT’14 and EUROCRYPT’16
Outline

1. LPN
2. Code Reduction
3. Our Algorithm
4. Results
Outline

1. LPN
2. Code Reduction
3. Our Algorithm
4. Results
Learning Parity with Noise (LPN)

LPN Oracle
Learning Parity with Noise (LPN)

LPN Oracle

secret random vector $s$
Learning Parity with Noise (LPN)

LPN Oracle

secret random vector $s$

$c_1 = \langle v_1, s \rangle \oplus d_1$
Learning Parity with Noise (LPN)

LPN Oracle

secret random vector $s$

$c_1 = \langle v_1, s \rangle \oplus d_1$

random vector
Learning Parity with Noise (LPN)

\[ c_1 = \langle v_1, s \rangle \oplus d_1 \]
Learning Parity with Noise (LPN)

LPN Oracle

secret random vector $s$ \rightarrow (v_1, c_1)

c_1 = \langle v_1, s \rangle \oplus d_1

random vector

noise
Learning Parity with Noise (LPN)

LPN Oracle

secret random vector $s$

$c_2 = \langle v_2, s \rangle \oplus d_2$

(random vector)

noise

$(v_2, c_2)$
Learning Parity with Noise (LPN)

LPN Oracle

secret random vector $s$

$c_i = \langle v_i, s \rangle \oplus d_i$

random vector

noise

$(v_i, c_i)$
Learning Parity with Noise (LPN)

LPN Oracle

secret random vector $s$

$c_i = \langle v_i, s \rangle \oplus d_i$

random vector

noise

Definition (LPN)

Given independent queries from the LPN oracle, find the secret $s$. 
LPN Solving Algorithm

**Definition (LPN solving algorithm)**

We say that an algorithm $\mathcal{M}$ solves the LPN problem if

$$\Pr[\mathcal{M} \text{ recovers the secret } s] \geq \frac{1}{2},$$

The performance of $\mathcal{M}$ is measured by the running time $t$, memory $m$ and **number of queries** $n$ from the LPN oracle.

Define $\delta = \Pr[d_i = 0] - \Pr[d_i = 1]$ as the **noise bias**.
To recover a secret $s$ of $k$ bits:

- reduce to a secret $s'$ of $k' \leq k$ bits
- recover the secret $s'$
- update the queries & repeat the steps
To recover a secret $s$ of $k$ bits:

- reduce to a secret $s'$ of $k' \leq k$ bits through reduction techniques
- recover the secret $s'$
- update the queries & repeat the steps
General Structure

To recover a secret $s$ of $k$ bits:

- reduce to a secret $s'$ of $k' \leq k$ bits through **reduction techniques**
- recover the secret $s'$ through **solving techniques**
- update the queries & repeat the steps
To recover a secret $s$ of $k$ bits:

- reduce to a secret $s'$ of $k' \leq k$ bits through reduction techniques
- recover the secret $s'$ through solving techniques
- update the queries & repeat the steps until the entire $s$ is recovered
To recover a secret $s$ of $k$ bits:

- reduce to a secret $s'$ of $k' \leq k$ bits through **reduction techniques**
- recover the secret $s'$ through **solving techniques**
- update the queries & repeat the steps until the **entire $s$ is recovered**

\[
\text{LPN}_s \xrightarrow{\text{reduction}} \text{LPN}_{s_1} \rightarrow \ldots \rightarrow \text{LPN}_{s_i} \xrightarrow{\text{solve}} s_i
\]
To recover a secret $s$ of $k$ bits:
- reduce to a secret $s'$ of $k' \leq k$ bits through **reduction techniques**
- recover the secret $s'$ through **solving techniques**
- update the queries & repeat the steps until the entire $s$ is recovered

\[ \text{LPN}_s \xrightarrow{\text{reduction}} \text{LPN}_{s_1} \xrightarrow{\ldots} \text{LPN}_{s_i} \xrightarrow{\text{solve}} s_i \]

**Optimise the use of the reduction techniques**
Reduction Techniques

- `sparse-secret`
- `partition-reduce(b)`
- `xor-reduce(b)`
- `drop-reduce(b)`
- `code-reduce(k, k', params)`
- `guess-secret(b, w)`
Reduction Techniques

- \textit{sparse-secret}
- \textit{partition-reduce}(b)
- \textit{xor-reduce}(b)
- \textit{drop-reduce}(b)
- \textit{code-reduce}(k, k', \text{params})
- \textit{guess-secret}(b, w)
Reduction Techniques

- sparse-secret
- partition-reduce($b$)
- xor-reduce($b$)
- drop-reduce($b$)
- code-reduce($k, k', \text{params}$)
- guess-secret($b, w$)

Keep track of the:
- secret size
- number of queries
- noise bias
- secret bias
Reduction *sparse-secret*

\[
\begin{array}{ccc}
V_1 & \cdots & C_1 \\
V_2 & \cdots & C_2 \\
V_3 & \cdots & C_3 \\
V_4 & \cdots & C_2 \\
V_5 & \cdots & C_5 \\
V_6 & \cdots & C_6 \\
\vdots & \ddots & \vdots \\
V_{n-2} & \cdots & C_{n-2} \\
V_{n-1} & \cdots & C_{n-1} \\
V_n & \cdots & C_n \\
\end{array}
\]
Reduction *sparse-secret*

\[
\begin{array}{c|c}
\nu_1 & \cdots & \nu_n \\
\nu_2 & \cdots & \nu_{n-1} \\
\nu_3 & \cdots & \nu_{n-2} \\
\nu_4 & \cdots & \nu_{n-3} \\
\nu_5 & \cdots & \nu_{n-4} \\
\nu_6 & \cdots & \nu_{n-5} \\
\vdots & \ddots & \vdots \\
\nu_{n-2} & \cdots & \nu_1 \\
\nu_{n-1} & \cdots & \nu_2 \\
\nu_n & \cdots & \nu_3 \\
\end{array}
\]

\[
\begin{array}{c|c}
\nu_1 & \cdots & C_1 \\
\nu_2 & \cdots & C_2 \\
\nu_3 & \cdots & C_3 \\
\nu_4 & \cdots & C_2 \\
\nu_5 & \cdots & C_5 \\
\nu_6 & \cdots & C_6 \\
\vdots & \ddots & \vdots \\
\nu_{n-2} & \cdots & C_{n-2} \\
\nu_{n-1} & \cdots & C_{n-1} \\
\nu_n & \cdots & C_n \\
\end{array}
\]
Reduction *sparse-secret*

\[
\begin{align*}
&k \\
&\begin{array}{ccc}
  & v_1 & \cdots & c_1 \\
  & v_2 & \cdots & c_2 \\
  & v_3 & \cdots & c_3 \\
  & v_4 & \cdots & c_2 \\
  & v_5 & \cdots & c_5 \\
  & v_6 & \cdots & c_6 \\
  \cdots & v_{n-2} & \cdots & c_{n-2} \\
  & v_{n-1} & \cdots & c_{n-1} \\
  & v_n & \cdots & c_n \\
\end{array}
\end{align*}
\]
Reduction *sparse-secret*

\[
k
\]

\[
\begin{array}{|c|c|}
\hline
v_1 & \cdots & c_1 \\
\hline
v_2 & \cdots & c_2 \\
\hline
v_3 & \cdots & c_3 \\
\hline
v_4 & \cdots & c_2 \\
\hline
v_5 & \cdots & c_5 \\
\hline
v_6 & \cdots & c_6 \\
\hline
\ldots & \ldots & \ldots \\
\hline
v_{n-2} & \cdots & c_{n-2} \\
\hline
v_{n-1} & \cdots & c_{n-1} \\
\hline
v_n & \cdots & c_n \\
\hline
\end{array}
\]

\[
c_i = \langle v_i, s \rangle \oplus d_i
\]
Reduction *sparse-secret*

\[ c_i = \langle v_i, s \rangle \oplus d_i \]

<table>
<thead>
<tr>
<th>( v_i )</th>
<th>( s )</th>
<th>( d_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 1 0 1</td>
<td>( \cdots )</td>
<td>0</td>
</tr>
<tr>
<td>1 1 1 0 1 0 0</td>
<td>( \cdots )</td>
<td>0</td>
</tr>
<tr>
<td>0 0 0 1 1 1 0</td>
<td>( \cdots )</td>
<td>0</td>
</tr>
<tr>
<td>0 0 0 0 0 1 0</td>
<td>( \cdots )</td>
<td>1</td>
</tr>
<tr>
<td>0 0 1 1 0 0 1</td>
<td>( \cdots )</td>
<td>1</td>
</tr>
<tr>
<td>1 0 1 0 0 1 1</td>
<td>( \cdots )</td>
<td>0</td>
</tr>
<tr>
<td>0 0 1 1 0 1 1</td>
<td>( \cdots )</td>
<td>0</td>
</tr>
<tr>
<td>1 0 1 0 1 1 0</td>
<td>( \cdots )</td>
<td>1</td>
</tr>
<tr>
<td>1 0 0 1 1 0 0</td>
<td>( \cdots )</td>
<td>1</td>
</tr>
</tbody>
</table>
Reduction *sparse-secret*

\[ c_i = \langle v_i, s \rangle \oplus d_i \]

Change the distribution of the secret
Reduction *sparse-secret*

\[ c_i = \langle v_i, s \rangle \oplus d_i \]

Change the distribution of the secret

- from \( s \) being uniformly distributed
- to an \( s \) where each bit has the same distribution as the noise
Reduction *sparse-secret*

\[ c'_i = \langle v'_i, s' \rangle \oplus d_i \]

| 1 0 1 0 1 0 1 | \cdots | 0 |
| 1 1 1 0 1 0 0 | \cdots | 0 |
| 0 0 0 1 1 1 0 | \cdots | 0 |
| 0 0 0 0 0 1 0 | \cdots | 1 |
| 0 0 1 1 0 0 1 | \cdots | 1 |
| 1 0 1 0 0 1 1 | \cdots | 0 |
| \cdots | \cdots | \cdots |
| 0 0 1 1 0 1 1 | \cdots | 0 |
| 1 0 1 0 1 1 0 | \cdots | 1 |
| 1 0 0 1 1 0 0 | \cdots | 1 |

Change the distribution of the secret
- from \( s \) being uniformly distributed
- to an \( s \) where each bit has the same distribution as the noise

Complexity: \( O(\min_{\chi \in \mathbb{N}}(k(n - k) \left\lceil \frac{k}{\chi} \right\rceil + k^3 + k\chi 2^\chi)) \)
Reduction $\text{xor-reduce}$

$$c_i = \langle v_i, s \rangle \oplus d_i$$

Find collisions on a window of $b$ bits
Reduction *xor-reduce*

\[ c_i = \langle v_i, s \rangle \oplus d_i \]

Find collisions on a window of $b$ bits

- group queries in equivalence classes
- xor each pair of queries from the same equivalence class
Reduction *xor-reduce*

\[ c_i \oplus c_j = \langle v_i \oplus v_j, s \rangle \oplus d_i \oplus d_j \]

Find collisions on a window of \( b \) bits

- group queries in equivalence classes
- xor each pair of queries from the same equivalence class

Complexity: \( O(k \cdot \max(n, \frac{n(n-1)}{2^{b+1}})) \)
Find collisions on a window of $b$ bits
- group queries in equivalence classes
- xor each pair of queries from the same equivalence class

Complexity: $O(k \cdot \max(n, \frac{n(n-1)}{2^{b+1}}))$

When $n \approx 1 + 2^{b+1}$, the number of queries stay constant
Reduction \textit{drop-reduce}

\[ c_i = \langle v_i, s \rangle \oplus d_i \]

Keep only the queries with 0 on a window of \( b \) bits
Reduction *drop-reduce*

\[
c_i = \langle v_i, s \rangle \oplus d_i
\]

Keep only the queries with 0 on a window of \(b\) bits
Reduction \textit{drop-reduce}

\[ c_i = \langle v_i, s \rangle \oplus d_i \]

Keep only the queries with 0 on a window of \( b \) bits
**Reduction** *drop-reduce*

\[
c_i = \langle v_i, s \rangle \oplus d_i
\]

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Keep only the queries with 0 on a window of \( b \) bits

Complexity: \( O(n(1 + \frac{1}{2} + \ldots + \frac{1}{2^{b-1}})) \)
Reduction \textit{code-reduce}

\[
k \begin{bmatrix}
1 & 0 & 1 & 0 & 1 & 0 & 1 & \cdots & 0 \\
1 & 1 & 1 & 0 & 1 & 0 & 0 & \cdots & 0 \\
0 & 0 & 0 & 1 & 1 & 1 & 0 & \cdots & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & \cdots & 1 \\
0 & 0 & 1 & 1 & 0 & 0 & 1 & \cdots & 1 \\
1 & 0 & 1 & 0 & 0 & 0 & 1 & \cdots & 0 \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
0 & 0 & 1 & 1 & 0 & 1 & 1 & \cdots & 0 \\
1 & 0 & 1 & 0 & 1 & 1 & 0 & \cdots & 1 \\
1 & 0 & 0 & 1 & 1 & 0 & 0 & \cdots & 1 
\end{bmatrix}
\]

\[c_i = \langle v_i, s \rangle \oplus d_i\]

Introduced at ASIACRYPT’14 [GJL]

Use a linear code $C[k, k', D]$ with generator matrix $G$, where $g = g' G \in C$

Approximate each vector $v_i$ to the nearest neighbour in the code $C$
Reduction code-reduce

\[
\begin{align*}
  k & \quad \begin{array}{cccccccc}
  1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & \\
  1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & \\
  0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & \\
  0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & \\
  0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & \\
  1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & \\
  \cdots & & \cdots & & \cdots & & \cdots & & \\
  \end{array} \\
  n & \\
  0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & \\
  1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & \\
  1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & \\
  \cdots & & \cdots & & \cdots & & \cdots & & \\
  \end{align*}
\]

\[c_i = \langle v_i, s \rangle \oplus d_i = \langle g, s \rangle \oplus \langle v_i - g, s \rangle \oplus d_i = \langle g' G, s \rangle \oplus \langle v_i - g, s \rangle \oplus d_i = \langle g', sG^T \rangle \oplus \langle v_i - g, s \rangle \oplus d_i\]

Introduced at ASIACRYPT’14 [GJL]

Use a linear code \( C[k, k', D] \) with generator matrix \( G \), where \( g = g' G \in C \)

Approximate each vector \( v_i \) to the nearest neighbour in the code \( C \)
Reduction code-reduce

\[
\begin{aligned}
\left\{ \begin{array}{c}
k' \\
1 1 0 1 \cdots 0 \\
0 1 0 0 \cdots 0 \\
0 1 1 1 \cdots 0 \\
1 0 1 0 \cdots 1 \\
0 0 1 0 \cdots 1 \\
0 0 1 1 \cdots 0 \\
\vdots \\
1 1 0 1 \cdots 0 \\
1 0 1 1 \cdots 1 \\
0 0 1 0 \cdots 1 \\
\end{array} \right.
\end{aligned}
\]

\[
c_i = \langle v_i, s \rangle \oplus d_i
= \langle g, s \rangle \oplus \langle v_i - g, s \rangle \oplus d_i
= \langle g' G, s \rangle \oplus \langle v_i - g, s \rangle \oplus d_i
= \langle g', sG^T \rangle \oplus \langle v_i - g, s \rangle \oplus d_i
\]

Introduced at ASIACRYPT'14 [GJL]

Use a linear code \( C[k, k', D] \) with generator matrix \( G \), where \( g = g' G \in C \)

Approximate each vector \( v_i \) to the nearest neighbour in the code \( C \)

Complexity: \( O(k \cdot n) \)
Define

\[ f(x) = \sum_i 1_{v_i = x} (-1)^{\langle v_i, s \rangle} \oplus d_i \]

and apply the Walsh Hadamard Transform (WHT) to obtain

\[ \hat{f}(\nu) = \sum_x (-1)^{\langle \nu, x \rangle} f(x) = \sum_i (-1)^{\langle v_i, s + \nu \rangle} \oplus d_i \]

|\(\hat{f}(s)\)| is large; In order to be the largest value in the table of \(\hat{f}\), we require certain amount of queries

Complexity: \(O(k2^k \frac{\log_2 n + 1}{2} + kn)\), when WHT is applied for a secret of \(k\) bits on \(n\) queries
Outline

1. LPN
2. Code Reduction
3. Our Algorithm
4. Results
Bias of the Code Reduction

For *code-reduce* we have

\[ c_i = \langle v_i, s \rangle \oplus d_i = \langle g', s' \rangle \oplus \langle v_i - g, s \rangle \oplus d_i \]
Bias of the Code Reduction

For \textit{code-reduce} we have

\[ c_i = \langle v_i, s \rangle \oplus d_i = \langle g', s' \rangle \oplus \langle v_i - g, s \rangle \oplus d_i \]

\[ bc = E((-1)^{\langle v_i - g, s \rangle}) = \sum_{e \in \{0,1\}^k} \Pr[v_i - g = e] E((-1)^{\langle e, s \rangle}) \]

\[ = E \left( \delta^\text{HW}(v_i - g) \right), \]

where \( \delta_s \) is the secret bias.
Bias of the Code Reduction

For code-reduce we have

\[ c_i = \langle v_i, s \rangle \oplus d_i = \langle g', s' \rangle \oplus \langle v_i - g, s \rangle \oplus d_i \]

\[ bc = E((-1)^{\langle v_i - g, s \rangle}) = \sum_{e \in \{0,1\}^k} \Pr[v_i - g = e]E((-1)^{\langle e, s \rangle}) \]

\[ = E(\delta_s^{\text{HW}(v_i - g)}) , \]

where \( \delta_s \) is the secret bias

We analyse:
- perfect codes
- quasi-perfect codes
- random codes
Perfect Codes

- Repetition code \([k, 1, \frac{k-1}{2}]\) with \(k\) odd

\[
bc = \sum_{w=0}^{\frac{k-1}{2}} \frac{1}{2^{k-1}} \binom{k}{w} \delta_s^w
\]

- Golay code \([23, 12, 7]\)

\[
bc = 2^{-11} \sum_{w=0}^{3} \binom{23}{w} \delta_s^w
\]

- Hamming code \([2^\ell - 1, 2^\ell - \ell, 3]\)

\[
bc = 2^{-\ell} \sum_{w=0}^{1} \binom{2^\ell - 1}{w} \delta_s^w
\]
Optimal Concatenated Code

Not every code $C[k, k', D]$ is perfect or quasi-perfect.
Optimal Concatenated Code

Not every code $C[k, k', D]$ is perfect or quasi-perfect

$\Downarrow$

Concatenate codes
Optimal Concatenated Code

Not every code $C[k, k', D]$ is perfect or quasi-perfect

$\Downarrow$

Concatenate codes

Take the $C[k, k', D]$ code as the concatenation of $C_1[k - \ell, k' - \ell', D_1]$ and $C_2[\ell, \ell', D_2]$ with $bc = bc_1 \cdot bc_2$
Optimal Concatenated Code

Not every code $C[k, k', D]$ is perfect or quasi-perfect

\[\downarrow\]

Concatenate codes

Take the $C[k, k', D]$ code as the concatenation of $C_1[k - \ell, k' - \ell', D_1]$ and $C_2[\ell, \ell', D_2]$ with $bc = bc_1 \cdot bc_2$

Computation:

- compute the biases for perfect, quasi-perfect and random codes
- for each $[k, k', D]$, check if $bc[k, k', D] < bc[k - \ell, k' - \ell', D_1] \cdot bc[\ell, \ell', D_2]$
Outline

1. LPN
2. Code Reduction
3. Our Algorithm
4. Results
LPN Solving Automaton

LPN solving algorithms = chains of reductions + WHT

\[
\text{LPN}_s \xrightarrow{\text{reduction}} \text{LPN}_{s_1} \xrightarrow{\ldots} \text{LPN}_{s_i} \xrightarrow{\text{solve}} s_i
\]
LPN Solving Automaton

LPN solving algorithms = chains of reductions + WHT

\[ \text{LPN}_s \xrightarrow{\text{reduction}} \text{LPN}_{s_1} \xrightarrow{\ldots} \text{LPN}_{s_i} \xrightarrow{\text{solve}} s_i \]

Diagram:

- Initial state: 1
- Sparse-secret (2) XOR-reduce WHT
- Code-reduce (3) XOR-reduce WHT
- Accepting state (4) XOR-reduce WHT
Graph of Reduction Chains

Construct a graph of all possible reduction chains

- the vertex stores the secret size and the number of queries
- the edge stores the bias change for a reduction
Graph of Reduction Chains

Construct a graph of all possible reduction chains

- the vertex stores the secret size and the number of queries
- the edge stores the bias change for a reduction

Find the reductions that optimize the bias
Graph of Reduction Chains

Construct a graph of all possible reduction chains

- the vertex stores the secret size and the number of queries
- the edge stores the bias change for a reduction

Find the reductions that optimize the bias

The time complexity of a chain is the sum of the complexities of each reduction step + cost of WHT
Graph of Reduction Chains

Construct a graph of all possible reduction chains

- the vertex stores the secret size and the number of queries
- the edge stores the bias change for a reduction

Find the reductions that optimize the bias

The **time complexity** of a chain is the sum of the complexities of each reduction step + cost of WHT

Use **max-complexity** as an approximation for the time complexity
Graph of Reduction Chains

Construct a graph of all possible reduction chains

- the vertex stores the secret size and the number of queries
- the edge stores the bias change for a reduction

Find the reductions that optimize the bias

The **time complexity** of a chain is the sum of the complexities of each reduction step + cost of WHT

Use **max-complexity** as an approximation for the time complexity

Find the chain with the smallest max-complexity and compute its total time complexity
Graph of Reduction Chains

Find the chain with the smallest max-complexity and compute its total time complexity
Outline

1. LPN
2. Code Reduction
3. Our Algorithm
4. Results
## Results

<table>
<thead>
<tr>
<th>$(k, \tau)$</th>
<th>ASIACRYPT’14 [GJL]</th>
<th>EUROCRYPT’16 [ZJW]</th>
<th>our results</th>
</tr>
</thead>
<tbody>
<tr>
<td>(512, 0.125)</td>
<td>81.90</td>
<td>80.09</td>
<td>78.84</td>
</tr>
<tr>
<td>(532, 0.125)</td>
<td>88.62</td>
<td>82.17</td>
<td>81.02</td>
</tr>
<tr>
<td>(592, 0.125)</td>
<td>97.71</td>
<td>89.32</td>
<td>87.57</td>
</tr>
</tbody>
</table>

Table: Logarithmic time complexity to solve LPN (in bit operations)

- $k$ - secret size
- $\tau$ - noise level
### Results

<table>
<thead>
<tr>
<th>$\tau$</th>
<th>32</th>
<th>48</th>
<th>64</th>
<th>100</th>
<th>256</th>
<th>512</th>
<th>768</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.05</td>
<td>13.89</td>
<td>14.52</td>
<td>16.04</td>
<td>20.47</td>
<td>36.75</td>
<td>57.77</td>
<td>76.63</td>
</tr>
<tr>
<td>0.1</td>
<td>15.04</td>
<td>18.58</td>
<td>21.58</td>
<td>27.61</td>
<td>46.75</td>
<td>73.68</td>
<td>98.97</td>
</tr>
<tr>
<td>0.125</td>
<td>15.66</td>
<td>19.29</td>
<td>22.94</td>
<td>28.91</td>
<td>49.90</td>
<td>78.85</td>
<td>105.89</td>
</tr>
<tr>
<td>0.2</td>
<td>17.01</td>
<td>21.25</td>
<td>24.42</td>
<td>32.06</td>
<td>56.31</td>
<td>89.04</td>
<td>121.04</td>
</tr>
<tr>
<td>0.25</td>
<td>18.42</td>
<td>22.34</td>
<td>26.86</td>
<td>32.94</td>
<td>59.47</td>
<td>94.66</td>
<td>127.35</td>
</tr>
</tbody>
</table>

**Table**: Logarithmic time complexity to solve LPN
<table>
<thead>
<tr>
<th>$\tau$</th>
<th>32</th>
<th>48</th>
<th>64</th>
<th>100</th>
<th>256</th>
<th>512</th>
<th>768</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.05</td>
<td>13.89</td>
<td>14.52</td>
<td>16.04</td>
<td>20.47</td>
<td>36.75</td>
<td>57.77</td>
<td>76.63</td>
</tr>
<tr>
<td>0.1</td>
<td>15.04</td>
<td>18.58</td>
<td>21.58</td>
<td>27.61</td>
<td>46.75</td>
<td>73.68</td>
<td>98.97</td>
</tr>
<tr>
<td>0.125</td>
<td>15.66</td>
<td>19.29</td>
<td>22.94</td>
<td>28.91</td>
<td>49.90</td>
<td>78.85</td>
<td>105.89</td>
</tr>
<tr>
<td>0.2</td>
<td>17.01</td>
<td>21.25</td>
<td>24.42</td>
<td>32.06</td>
<td>56.31</td>
<td>89.04</td>
<td>121.04</td>
</tr>
<tr>
<td>0.25</td>
<td>18.42</td>
<td>22.34</td>
<td>26.86</td>
<td>32.94</td>
<td>59.47</td>
<td>94.66</td>
<td>127.35</td>
</tr>
</tbody>
</table>

**Table:** Logarithmic time complexity to solve LPN
Conclusion

- Create an algorithm that automatizes the LPN solving algorithms
- Improve the best existing results
- New reduction techniques can be integrated later on

Thank you for your kind attention!