How to Generate Correlated Randomness from (variants of) LPN

Part I

Based on joint works with: Elette Boyle, Niv Gilboa, Yuval Ishai, Lisa Kohl, Srinivasan Raghuraman, Peter Rindal, Peter Scholl
How to Generate Correlated Randomness from (variants of) LPN

Part I

Stay tuned for part II :)

Based on joint works with: Elette Boyle, Niv Gilboa, Yuval Ishai, Lisa Kohl, Srinivasan Raghuraman, Peter Rindal, Peter Scholl
<table>
<thead>
<tr>
<th>What is Secure Computation?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
What is Secure Computation?

**Secure communication**

Goal: *communicating* a secret message

Output: Bob learns $m$

Security: Eve learns nothing
What is Secure Computation?

**Secure communication**
Goal: *communicating* a secret message

*Output:* Bob learns $m$

*Security:* Eve learns nothing

**Secure computation**
Goal: *computing* a (public) function on secret inputs $f_A(\cdot, \cdot), f_B(\cdot, \cdot)$

*Output:* Alice learns $f_A(x, y)$ and Bob learns $f_B(x, y)$

*Security:* Alice and Bob learn nothing else
What is Secure Computation?

Secure communication
Goal: communicating a secret message

Output: Bob learns $m$
Security: Eve learns nothing

Secure computation
Goal: computing a (public) function on secret inputs

Output: Alice learns $f_A(x, y)$ and Bob learns $f_B(x, y)$
Security: Alice and Bob learn nothing else

- It’s a more fine-grained approach to security: the function controls precisely what is learned (secure communication is all or nothing)
- It is much more demanding: now the adversary is internal (Alice must be protected against Bob, and Bob against Alice), and can influence the protocol!
Secure Computation from Oblivious Transfer

**Oblivious Transfer**
A minimal example of secure computation

\[(s_0, s_1)\]

**Output:** Bob learns \(s_b\)

**Security:** Alice does not learn \(b\), Bob does not learn \(s_{1-b}\).
Secure Computation from Oblivious Transfer

**Oblivious Transfer**
A minimal example of secure computation

\[(s_0, s_1) \quad \text{OT} \quad b\]

**Output:** Bob learns \(s_b\)

**Security:** Alice does not learn \(b\), Bob does not learn \(s_{1-b}\).

**Secure Computation for all functions**

\[f_A(\cdot, \cdot), f_B(\cdot, \cdot)\]

\[x \quad \text{GMW} \quad y\]

**Output:** Alice learns \(f_A(x, y)\) and Bob learns \(f_B(x, y)\)

**Security:** Alice and Bob learn nothing else.
Secure Computation from Oblivious Transfer

**Oblivious Transfer**
A minimal example of secure computation

\[(s_0, s_1) \rightarrow OT \rightarrow b\]

**Output:** Bob learns \(s_b\)

**Security:** Alice does not learn \(b\), Bob does not learn \(s_{1-b}\).

**Secure Computation for all functions**

\[f_A(\cdot, \cdot), f_B(\cdot, \cdot)\]

\[x \leftrightarrow \text{GMW} \leftrightarrow y\]

**Output:** Alice learns \(f_A(x, y)\) and Bob learns \(f_B(x, y)\)

**Security:** Alice and Bob learn nothing else

1. Use (additive) secret sharing
Secure Computation from Oblivious Transfer

Oblivious Transfer
A minimal example of secure computation

\((s_0, s_1)\)

Output: Bob learns \(s_b\)
Security: Alice does not learn \(b\), Bob does not learn \(s_{1-b}\).

Secure Computation for all functions

\(f_A(\cdot, \cdot), f_B(\cdot, \cdot)\)

Output: Alice learns \(f_A(x, y)\) and Bob learns \(f_B(x, y)\)
Security: Alice and Bob learn nothing else

1. Use (additive) secret sharing
2. Write the function as a circuit
Secure Computation from Oblivious Transfer

Oblivious Transfer
A minimal example of secure computation

Output: Bob learns $s_b$
Security: Alice does not learn $b$, Bob does not learn $s_{1-b}$.

Secure Computation for all functions

Output: Alice learns $f_A(x, y)$ and Bob learns $f_B(x, y)$
Security: Alice and Bob learn nothing else

1. Use (additive) secret sharing

2. Write the function as a circuit

3. Use OT to compute the gates

share($x, y) \implies \text{share(GATE}(x, y)$)

I’ll skip the details for now, but feel free to ask for them!
Secure Computation from Correlated Randomness

Suppose that, for some reason, the parties could already obtain the result of a random oblivious transfer prior to the protocol:
Secure Computation from Correlated Randomness

Suppose that, for some reason, the parties could already obtain the result of a random oblivious transfer prior to the protocol:

Then the parties can use it to perform an arbitrary oblivious transfer!

(Simple) protocol:

- If $a = b$ and Bob gets $(s_0 \oplus r_0, s_1 \oplus r_1)$, he can get $s_b = s_a$, since he knows only $r_b = r_a$.
- If $a = 1 - b$ and Bob gets $(s_0 \oplus r_1, s_1 \oplus r_0)$, he again gets $s_b$, since he knows only $r_{1-b}$.
- Bob simply tells Alice whether $a = b$ (leaks nothing since $a$ is random!), and Alice sends the appropriate pair.
Secure Computation from Correlated Randomness

Suppose that, for some reason, the parties could already obtain the result of a *random* oblivious transfer prior to the protocol:

Given *many* random OTs, one can compute an arbitrary function! The protocol is

- Information-theoretically secure
- Very fast: only three bits exchanged per OT! (In practice, this means 6 bits / AND gate, and 0 / XOR gate)

This is the *correlated randomness model*: fast, information-theoretically secure computation given access to a (trusted) source of correlated random coins.

Then the parties can use it to perform an *arbitrary* oblivious transfer!

(Simple) protocol:

- If $a = b$ and Bob gets $(s_0 \oplus r_0, s_1 \oplus r_1)$, he can get $s_b = s_a$, since he knows only $r_b = r_a$.
- If $a = 1 - b$ and Bob gets $(s_0 \oplus r_0, s_1 \oplus r_0)$, he again gets $s_b$, since he knows only $r_1 - b$.
- Bob simply tells Alice whether $a = b$ (leaks nothing since $a$ is random!), and Alice sends the appropriate pair.
Secure Computation from Correlated Randomness

Suppose that, for some reason, the parties could already obtain the result of a random oblivious transfer prior to the protocol:

\[(r_0, r_1) \rightarrow (a, r_a)\]

Random OT

\[(s_0, s_1) \rightarrow b\]

Then the parties can use it to perform an arbitrary oblivious transfer!

(Simple) protocol:

- If \(a = b\) and Bob gets \((s_0 \oplus r_0, s_1 \oplus r_1)\), he can get \(s_b = s_a\), since he knows only \(r_b = r_a\).
- If \(a = 1 - b\) and Bob gets \((s_0 \oplus r_1, s_1 \oplus r_0)\), he again gets \(s_b\), since he knows only \(r_1 - b\).
- Bob simply tells Alice whether \(a = b\) (leaks nothing since \(a\) is random!), and Alice sends the appropriate pair.

Given many random OTs, one can compute an arbitrary function! The protocol is:

- Information-theoretically secure
- Very fast: only three bits exchanged per OT! (In practice, this means 6 bits / AND gate, and 0 / XOR gate)

This is the correlated randomness model: fast, information-theoretically secure computation given access to a (trusted) source of correlated random coins.

The natural question:

Can we efficiently generate (securely) large amounts of correlated randomness?

Perhaps the most fundamental question in secure computation!
Secure Computation from Correlated Randomness

Suppose that, for some reason, the parties could already obtain the result of a random oblivious transfer prior to the protocol:

\[(r_0, r_1), (a, r_a)\]

Random OT

\[(s_0, s_1), b\]

Then the parties can use it to perform an arbitrary oblivious transfer!

(Simple) protocol:
- If \(a = b\) and Bob gets \((s_0 \oplus r_0, s_1 \oplus r_1)\), he can get \(s_b = s_a\), since he knows only \(r_b = r_a\).
- If \(a = 1 - b\) and Bob gets \((s_0 \oplus r_1, s_1 \oplus r_0)\), he again gets \(s_b\), since he knows only \(r_{1-b}\).
- Bob simply tells Alice whether \(a = b\) (leaks nothing since \(a\) is random!), and Alice sends the appropriate pair.

Given many random OTs, one can compute an arbitrary function! The protocol is
- Information-theoretically secure
- Very fast: only three bits exchanged per OT! (In practice, this means 6 bits / AND gate, and 0 / XOR gate)

This is the correlated randomness model: fast, information-theoretically secure computation given access to a (trusted) source of correlated random coins.

The natural question:
Can we efficiently generate (securely) large amounts of correlated randomness?

Perhaps the most fundamental question in secure computation!

This talk:
Can we compress correlated randomness?

Turns out to be just the right way to ask the previous question.
A source of secret *correlated* randomness is an extremely useful resource in secure protocols:
Correlated Randomness in Cryptography

A source of secret correlated randomness is an extremely useful resource in secure protocols:

Correlation: $R_A = R_B$

(Equality correlation)

One-time pad

IT

C
A source of secret correlated randomness is an extremely useful resource in secure protocols:

**One-time pad**

- Correlation: $R_A = R_B$
- (Equality correlation)

**GMW**

- Correlation: $R_A + R_B = S_A \land S_B$
- (Oblivious transfer correlation)
A source of secret correlated randomness is an extremely useful resource in secure protocols:

**One-time pad**

Correlation: \( R_A = R_B \)  
(Equality correlation)

**GMW**

Correlation: \( R_A + R_B = S_A \land S_B \)  
(Oblivious transfer correlation)

In the computational world, can we *compress* correlated randomness?
A source of secret *correlated* randomness is an extremely useful resource in secure protocols:

**One-time pad**

$R_A = R_B$

*Correlation: $R_A = R_B$*

*(Equality correlation)*

**GMW**

$R_A + R_B = S_A \land S_B$

*Correlation: $R_A + R_B = S_A \land S_B$*

*(Oblivious transfer correlation)*

Equality correlations can be *compressed* using a PRG:

$R_A = \text{PRG}(\text{seed}_A)$

$R_B = \text{PRG}(\text{seed}_B)$
Correlated Randomness in Cryptography

A source of secret *correlated* randomness is an extremely useful resource in secure protocols:

**One-time pad**

Correlation: $R_A = R_B$

(Equality correlation)

**GMW**

Correlation: $R_A + R_B = S_A \land S_B$

(Oblivious transfer correlation)

Equality correlations can be *compressed* using a PRG:

$R_A = \text{PRG}(\text{seed}_A)$

$R_B = \text{PRG}(\text{seed}_B)$

Can OT correlations be *compressed* using a PCG?

$\text{Gen}(1^\lambda)$

$(R_A, S_A)$

Expand($i, \text{seed}_i$)

$(R_B, S_B)$
Secure Computation with Silent Preprocessing

**Pseudorandom correlation generator:** Gen$(1^λ)$ → $(\text{seed}_A, \text{seed}_B)$ such that (1) $(\text{Expand}(A, \text{seed}_A), \text{Expand}(B, \text{seed}_B))$ looks like $n$ samples from the target correlation, and (2) $\text{Expand}(A, \text{seed}_A)$ looks ‘random conditioned on satisfying the correlation with $\text{Expand}(B, \text{seed}_B)$’ to Bob (similar property w.r.t. Alice).
Secure Computation with Silent Preprocessing

**Pseudorandom correlation generator:** $\text{Gen}(1^\lambda) \rightarrow (\text{seed}_A, \text{seed}_B)$ such that 1) $(\text{Expand}(A, \text{seed}_A), \text{Expand}(B, \text{seed}_B))$ looks like $n$ samples from the target correlation, and 2) $\text{Expand}(A, \text{seed}_A)$ looks ‘random conditioned on satisfying the correlation with $\text{Expand}(B, \text{seed}_B)$’ to Bob (similar property w.r.t. Alice).

**One-time short interaction**

Interactive protocol with short communication and computation; Alice and Bob store a small seed afterwards.

**Preprocessing phase**

**Online phase**
Secure Computation with Silent Preprocessing

**Pseudorandom correlation generator:** \( \text{Gen}(1^\lambda) \rightarrow (\text{seed}_A, \text{seed}_B) \) such that (1) \( (\text{Expand}(A, \text{seed}_A), \text{Expand}(B, \text{seed}_B)) \) looks like \( n \) samples from the target correlation, and (2) \( \text{Expand}(A, \text{seed}_A) \) looks ‘random conditioned on satisfying the correlation with \( \text{Expand}(B, \text{seed}_B) \)’ to Bob (similar property w.r.t. Alice).

**One-time short interaction**

Interactive protocol with short communication and computation; Alice and Bob store a small seed afterwards.

**‘Silent’ computation**

The bulk of the preprocessing phase is offline: Alice and Bob stretch their seeds into large pseudorandom correlated strings.

**Preprocessing phase**

**Online phase**
Secure Computation with Silent Preprocessing

Pseudorandom correlation generator: \( \text{Gen}(1^\lambda) \rightarrow (\text{seed}_A, \text{seed}_B) \) such that (1) \((\text{Expand}(A, \text{seed}_A), \text{Expand}(B, \text{seed}_B))\) looks like \(n\) samples from the target correlation, and (2) \(\text{Expand}(A, \text{seed}_A)\) looks 'random conditioned on satisfying the correlation with \(\text{Expand}(B, \text{seed}_B)\)' to Bob (similar property w.r.t. Alice).

One-time short interaction

Interactive protocol with short communication and computation; Alice and Bob store a small seed afterwards.

'Silent' computation

The bulk of the preprocessing phase is offline: Alice and Bob stretch their seeds into large pseudorandom correlated strings.

Non-cryptographic

Alice and Bob consume the preprocessing material in a fast, non-cryptographic online phase.
A quick reminder of what we want: Gen generates short correlated seeds which can be locally expanded into pseudorandom instances of a target correlation.

**A construction from LPN**

0. Rewriting the ‘many OTs correlation’

Oblivious transfer correlation: $\overrightarrow{w}_A + \overrightarrow{w}_B = \overrightarrow{u} \ast \overrightarrow{v}$
A quick reminder of what we want: Gen generates short correlated seeds which can be locally expanded into pseudorandom instances of a target correlation.

Oblivious transfer correlation: \( \overrightarrow{w_A} + \overrightarrow{w_B} = \overrightarrow{u} \star \overrightarrow{v} \)

Because \( s_b \oplus s_0 = b \cdot (s_0 \oplus s_1) \), Hence \( \overrightarrow{u} \) are the selection bits, \( \overrightarrow{w_B} \) are the \( s_0 \)'s, \( \overrightarrow{w_A} \) are the outputs, and \( \overrightarrow{v} \) allows to recover the \( s_1 \)'s.
Pseudorandom Correlation Generators - Walkthrough

A quick reminder of what we want: Gen generates short correlated seeds which can be locally expanded into pseudorandom instances of a target correlation.

Because $s_b \oplus s_0 = b \cdot (s_0 \oplus s_1)$, Hence $\vec{u}$ are the selection bits, $\vec{w}_B$ are the $s_0$'s, $\vec{w}_A$ are the outputs, and $\vec{v}$ allows to recover the $s_1$'s.

Oblivious transfer correlation: $\vec{w}_A + \vec{w}_B = \vec{u} \star \vec{v}$

A construction from LPN

0. Rewriting the ‘many OTs correlation’

1. Reduction to subfield-VOLE

[IKNP03]: subfield vector-OLE correlation + correlation-robust hash functions gives (pseudorandom) OT correlations.

Subfield vector-OLE

$$(\vec{u}, \vec{w}_A) \in \mathbb{F}_2^n \times \mathbb{F}_2^n \quad (x, \vec{w}_B) \in \mathbb{F}_2^i \times \mathbb{F}_2^n$$

$$\vec{w}_A + \vec{w}_B = x \cdot \vec{u}$$
A quick reminder of what we want: Gen generates short correlated seeds which can be locally expanded into pseudorandom instances of a target correlation.

Oblivious transfer correlation: \( \overrightarrow{w}_A + \overrightarrow{w}_B = \overrightarrow{u} \star \overrightarrow{v} \)

Because \( s_b \oplus s_0 = b \cdot (s_0 \oplus s_1) \), Hence \( \overrightarrow{u} \) are the selection bits, \( \overrightarrow{w}_B \) are the \( s_0 \)'s, \( \overrightarrow{w}_A \) are the outputs, and \( \overrightarrow{v} \) allows to recover the \( s_1 \)'s.

A construction from LPN

0. Rewriting the ‘many OTs correlation’
1. Reduction to subfield-VOLE

[IKNP03]: subfield vector-OLE correlation + correlation-robust hash functions gives (pseudorandom) OT correlations.
A quick reminder of what we want: Gen generates short correlated seeds which can be locally expanded into pseudorandom instances of a target correlation.

A construction from LPN

0. Rewriting the ‘many OTs correlation’
1. Reduction to subfield-VOLE
2. Constructing a PCG for subfield-VOLE

Three steps:

1. Construction for a random unit vector $\vec{u}$ from puncturable pseudorandom functions
2. Construction for a random $t$-sparse vector $\vec{u}$ via $t$ parallel repetitions of (1)
3. Construction for a pseudorandom vector $\vec{u}$ using dual-LPN

New target
Pseudorandom Correlation Generators - Walkthrough

1. Construction for a random unit vector $\vec{u}$ from puncturable pseudorandom functions

   $seed_A = (x, K)$
   $seed_B = (\vec{u}, (\alpha : u_\alpha = 1))$

\[ (\vec{u}, \vec{w}_A) \in \mathbb{F}_2^n \times \mathbb{F}_{2^i}^n \]
\[ (x, \vec{w}_B) \in \mathbb{F}_{2^i}^n \times \mathbb{F}_2^n \]
\[ \vec{w}_A + \vec{w}_B = x \cdot \vec{u} \]

Three steps:

0. Rewriting the ‘many OTs correlation’
1. Reduction to subfield-VOLE
2. Constructing a PCG for subfield-VOLE

A construction from LPN
Pseudorandom Correlation Generators - Walkthrough

1. Construction for a random unit vector $\vec{u}$ from puncturable pseudorandom functions
   
   $\text{seed}_A = (x, \vec{w}_A)$, $\text{seed}_B = (\vec{u}, \vec{w}_B)$

0. Rewriting the ‘many OTs correlation’

1. Reduction to subfield-VOLE
2. Constructing a PCG for subfield-VOLE

   Three steps:

   1. Construction for a random unit vector $\vec{u}$ from puncturable pseudorandom functions

   A construction from LPN

   \[ (\alpha : u_\alpha = 1) \]

   \[ (1 : \lambda) \]

   \[ (\vec{w}_A, \vec{w}_B) \in \mathbb{F}_2^n \times \mathbb{F}_2^n \]

   \[ (x, \vec{w}_B) \in \mathbb{F}_2^n \times \mathbb{F}_2^n \]

   \[ \vec{w}_A + \vec{w}_B = x \cdot \vec{u} \]
Pseudorandom Correlation Generators - Walkthrough

1. Construction for a random unit vector \( \vec{u} \) from puncturable pseudorandom functions

\[ \text{seed}_A = (x, K), \quad \text{seed}_B = (\vec{u}), \]

(\( \alpha : u_\alpha = 1 \))

---

A construction from LPN

0. Rewriting the ‘many OTs correlation’
1. Reduction to subfield-VOLE
2. Constructing a PCG for subfield-VOLE

Three steps:

1. Construction for a random unit vector \( \vec{u} \) from puncturable pseudorandom functions

\[ (\vec{u}, \vec{w}_A) \in \mathbb{F}_2^n \times \mathbb{F}_2^{n\alpha} \]
\[ (x, \vec{w}_B) \in \mathbb{F}_2^{n\alpha} \times \mathbb{F}_2^n \]
\[ \vec{w}_A + \vec{w}_B = x \cdot \vec{u} \]
Pseudorandom Correlation Generators - Walkthrough

1. Construction for a random unit vector $\vec{u}$ from puncturable pseudorandom functions

\[
\text{seed}_A = (x, K) \quad \text{seed}_B = (\vec{u}),
\]

0. Rewriting the ‘many OTs correlation’

1. Reduction to subfield-VOLE

2. Constructing a PCG for subfield-VOLE

Three steps:

1. Construction for a random unit vector $\vec{u}$ from puncturable pseudorandom functions

A construction from LPN

\[
\begin{align*}
\text{seed}_A &= (x, K) \\
\text{seed}_B &= (\vec{u}),
\end{align*}
\]

\[
\begin{align*}
\text{Gen}(1^\lambda) &\quad \text{Expand}(i, \text{seed}_i) \quad \text{seed}_B \\
(\vec{u}, \vec{w}_A) &\in \mathbb{F}_2^n \times \mathbb{F}_2^{n_i} \\
(x, \vec{w}_B) &\in \mathbb{F}_2^i \times \mathbb{F}_2^n \\
\vec{w}_A + \vec{w}_B &= x \cdot \vec{u}
\end{align*}
\]
Pseudorandom Correlation Generators - Walkthrough

1. Construction for a random unit vector $\vec{u}$ from puncturable pseudorandom functions

```
seed_A = (x, K)  seed_B = (\vec{u}, \alpha \cdot (\alpha : u_\alpha = 1) \oplus x)
```

0. Rewriting the ‘many OTs correlation’

1. Reduction to subfield-VOLE
2. Constructing a PCG for subfield-VOLE

**Three steps:**

1. Construction from LPN

```
(\vec{u}, \vec{w}) \in \mathbb{F}_2^n \times \mathbb{F}_2^n
(x, \vec{w}) = (\vec{u}^\top x) \in \mathbb{F}_2 \times \mathbb{F}_2^n
\vec{w}_A + \vec{w}_B = x \cdot \vec{u}
```

GGM
Pseudorandom Correlation Generators - Walkthrough

1. Construction for a random unit vector $\vec{u}$ from puncturable pseudorandom functions

$(\alpha : u_\alpha = 1)$

$seed_A = (x, K) \quad seed_B = (\vec{u}, \underbrace{\vdots}_{\alpha}, \underbrace{\oplus x})$

$\vec{w}_A \leftarrow \text{FullEval}(K) \quad \vec{w}_B \leftarrow \text{Insert}(x \oplus F_K(\alpha), \text{FullEval}(K_{\{\alpha\}}))$

A construction from LPN

0. Rewriting the ‘many OTs correlation’
1. Reduction to subfield-VOLE
2. Constructing a PCG for subfield-VOLE

Three steps:

Construction for a random unit vector $\vec{u}$ from puncturable pseudorandom functions

$seed_A = (x, K) \quad seed_B = (\vec{u}, \underbrace{\vdots}_{\alpha}, \underbrace{\oplus x})$

$\vec{w}_A \leftarrow \text{FullEval}(K) \quad \vec{w}_B \leftarrow \text{Insert}(x \oplus F_K(\alpha), \text{FullEval}(K_{\{\alpha\}}))$

1. Reduction to subfield-VOLE
2. Constructing a PCG for subfield-VOLE
Pseudorandom Correlation Generators - Walkthrough

**Construction for a random t-sparse vector \( \overrightarrow{u} \)**
via \( t \) parallel repetitions of (1)

\[
\text{seed}_A = (x, K^1) \quad \text{seed}_B = \left( \overrightarrow{u}_1, K_\{\alpha_1\}^1, F_{K^1}(\alpha_1) \oplus x \right)
\]

\[
K^2
\]

\[
\vdots
\]

\[
K^t
\]

• Write \( \overrightarrow{u} \) as a sum of \( t \) unit vectors \( \overrightarrow{u}_1 \cdots \overrightarrow{u}_t \)
• Apply the previous construction \( t \) times (with the same \( x \))
• After expansion, the parties locally sum their shares:

\[
\left( \bigoplus_{i=1}^{t} \overrightarrow{w}_A^i \right) \oplus \left( \bigoplus_{i=1}^{t} \overrightarrow{w}_B^i \right) = x \cdot \left( \bigoplus_{i=1}^{t} \overrightarrow{u}_i \right) = x \cdot \overrightarrow{u}
\]

**A construction from LPN**

0. Rewriting the ‘many OTs correlation’
1. Reduction to subfield-VOLE
2. Constructing a PCG for subfield-VOLE

Three steps:

1. Construction for a random unit vector \( \overrightarrow{u} \)
   from puncturable pseudorandom functions

2. Construction for a random t-sparse vector \( \overrightarrow{u} \)
   via \( t \) parallel repetitions of (1)
Construction for a pseudorandom vector $\vec{u}$ using dual-LPN

The LPN assumption - primal

A construction from LPN

0. Rewriting the ‘many OTs correlation’

1. Reduction to subfield-VOLE

2. Constructing a PCG for subfield-VOLE

   Three steps:

   1. Construction for a random unit vector $\vec{u}$ from puncturable pseudorandom functions

   2. Construction for a random $t$-sparse vector $\vec{u}$ via $t$ parallel repetitions of (1)

   3. Construction for a pseudorandom vector $\vec{u}$ using dual-LPN

Same seeds as in step (2)
Pseudorandom Correlation Generators - Walkthrough

Construction for a pseudorandom vector $\vec{u}$ using dual-LPN

The LPN assumption - primal

A construction from LPN

0. Rewriting the ‘many OTs correlation’
1. Reduction to subfield-VOLE
2. Constructing a PCG for subfield-VOLE
   Three steps:
   
   - Construction for a random unit vector $\vec{u}$ from puncturable pseudorandom functions
   - Construction for a random $t$-sparse vector $\vec{u}$ via $t$ parallel repetitions of (1)
   - Construction for a pseudorandom vector $\vec{u}$ using dual-LPN
Pseudorandom Correlation Generators - Walkthrough

2. Constructing a PCG for subfield-VOLE

Three steps:

1. Reduction to subfield-VOLE
2. Constructing a PCG for subfield-VOLE

Three steps:

1. Construction for a random unit vector $\vec{u}$ from puncturable pseudorandom functions

2. Construction for a random $t$-sparse vector $\vec{u}$ via $t$ parallel repetitions of (1)

3. Construction for a pseudorandom vector $\vec{u}$ using dual-LPN

A construction from LPN

0. Rewriting the ‘many OTs correlation’
Pseudorandom Correlation Generators - Walkthrough

Construction for a pseudorandom vector $\vec{u}$ using dual-LPN

1. Construction for a random unit vector $\vec{u}$ from puncturable pseudorandom functions

2. Construction for a random $t$-sparse vector $\vec{u}$ via $t$ parallel repetitions of (1)

3. Construction for a pseudorandom vector $\vec{u}$ using dual-LPN

A construction from LPN

0. Rewriting the ‘many OTs correlation’

1. Reduction to subfield-VOLE

2. Constructing a PCG for subfield-VOLE

Three steps:

The LPN assumption - primal

Parity-check matrix of $G$

Random matrix

Short secret

Sparse noise

$H \cdot \left( \begin{pmatrix} x \end{pmatrix}, \begin{pmatrix} w_A \\ w_B \end{pmatrix} \right) \approx \$
Pseudorandom Correlation Generators - Walkthrough

### Construction for a pseudorandom vector $\vec{u}$ using dual-LPN

1. **Reduction to subfield-VOLE**
   - Construction for a random unit vector $\vec{u}$ from puncturable pseudorandom functions

2. **Constructing a PCG for subfield-VOLE**
   - Construction for a random $t$-sparse vector $\vec{u}$ via $t$ parallel repetitions of (1)

3. **Construction for a pseudorandom vector $\vec{u}$ using dual-LPN**

### The LPN assumption - dual

- $H \cdot \vec{u} \approx $ $\vec{w}$
  - Random matrix
  - Sparse noise
Pseudorandom Correlation Generators - Walkthrough

2. Constructing a PCG for subfield-VOLE

1. Reduction to subfield-VOLE

2. Construction for a random unit vector from puncturable pseudorandom functions $\vec{u}$

3. Construction for a random $t$-sparse vector via parallel repetitions of (1) $t\vec{u}$

3. Construction for a pseudorandom vector using dual-LPN $\vec{u}$

\[ H \cdot \vec{x} \approx \vec{s} \]

The LPN assumption - dual

\[ H \cdot \vec{x} \approx \vec{s} \]

Random matrix

Sparse noise

Dual Version: Syndrome Decoding

Problem: find $s$

Problem: find $e$

Remember, from Benny’s talk on Monday
Pseudorandom Correlation Generators - Walkthrough

Construction for a pseudorandom vector $\vec{u}$ using dual-LPN

1. Reduction to subfield-VOLE
2. Constructing a PCG for subfield-VOLE

Three steps:

1. Construction for a random unit vector $\vec{u}$ from puncturable pseudorandom functions
2. Construction for a random $t$-sparse vector $\vec{u}$ via $t$ parallel repetitions of (1)
3. Construction for a pseudorandom vector $\vec{u}$ using dual-LPN

Rewriting the ‘many OTs correlation’

$H \cdot \vec{w}_A + H \cdot \vec{w}_B = H \cdot (x \cdot \vec{u}) = x \cdot (H \cdot \vec{u})$

Pseudorandom under the LPN assumption
Pseudorandom Correlation Generators - Efficiently?

\[ \overrightarrow{w}_A + \overrightarrow{w}_B = x \cdot \overrightarrow{u} \]

- \( \lambda \) is a security parameter, \( t \) is an LPN noise parameter, \( n \) is the vector length.
- Converted to \( n \) pseudorandom OTs via a correlation-robust hash function.
Placing the Assumption in the LPN Landscape

Remember this slide, shamefully stolen from Benny’s talk on Monday?
Placing the Assumption in the LPN Landscape

Remember this slide, shamefully stolen from Benny’s talk on Monday?

We use $O(n)$ samples and noise $\lambda/n$. Therefore, we have exp. security in $\lambda$, and we do not view $n$ as the security parameter anymore (since it is our target number of OTs).
Pseudorandom Correlation Generators - Efficiently?

\[ \overrightarrow{w}_A + \overrightarrow{w}_B = x \cdot \overrightarrow{u} \]

\( |\text{seed}_A| \approx \lambda \cdot t \quad \text{and} \quad |\text{seed}_B| \approx \lambda \cdot t \cdot \log n \)

- \( \lambda \) is a security parameter, \( t \) is an LPN noise parameter, \( n \) is the vector length.
- Converted to \( n \) pseudorandom OTs via a correlation-robust hash function.

Wrapping-up

Is this really efficient?

The expansion of the PCG boils down to the computation of

\[ H \cdot \overrightarrow{e} \]

Big random matrix

Where \( \overrightarrow{e} \) is a very sparse vector, and (the shares of) the entries of \( x \cdot \overrightarrow{e} \) can be computed individually in log-time.
Pseudorandom Correlation Generators - Efficiently?

**Wrapping-up**

\[ \overrightarrow{w}_A + \overrightarrow{w}_B = x \cdot \overrightarrow{u} \]

\[ |\text{seed}_A| \approx \lambda \cdot t \]

\[ |\text{seed}_B| \approx \lambda \cdot t \cdot \log n \]

- \( \lambda \) is a security parameter, \( t \) is an LPN noise parameter, \( n \) is the vector length.
- Converted to \( n \) pseudorandom OTs via a correlation-robust hash function.

**Is this really efficient?**

The expansion of the PCG boils down to the computation of

\[ H \cdot \langle x \cdot \overrightarrow{e} \rangle \]

Where \( \overrightarrow{e} \) is a very sparse vector, and (the shares of) the entries of \( x \cdot \overrightarrow{e} \) can be computed individually in log-time.

Computing \( H \cdot \langle x \cdot \overrightarrow{e} \rangle \) takes a time quadratic in \( n \)...

But remember that \( n \) is the number of OTs we want: it’s easily in the millions or billions.

This is nowhere near practical!
Pseudorandom Correlation Generators - Efficiently?

Wrapping-up

\[ \overrightarrow{w}_A + \overrightarrow{w}_B = x \cdot \overrightarrow{u} \]

| seed\textsubscript{A} | \approx \lambda \cdot t  \\  
| seed\textsubscript{B} | \approx \lambda \cdot t \cdot \log n

\begin{itemize}
  \item $\lambda$ is a security parameter, $t$ is an LPN noise parameter, $n$ is the vector length.
  \item Converted to $n$ pseudorandom OTs via a correlation-robust hash function.
\end{itemize}

Is this really efficient?

The expansion of the PCG boils down to the computation of

\[ H \cdot \langle x \cdot \overrightarrow{e} \rangle \]

Where $\overrightarrow{e}$ is a very sparse vector, and (the shares of) the entries of $x \cdot \overrightarrow{e}$ can be computed individually in log-time.

\textbf{This is nowhere near practical!}

We need to use variants of LPN, where multiplication by $H$ is (much) faster, ideally linear-time.
We want: computing $H \cdot \_ \_ \_ \_ \_$ is fast, and the code generated by $G$ is LPN-friendly.
Pseudorandom Correlation Generators - Efficiently?

We want: computing $H \cdot \cdot$ is fast, and the code generated by $G$ is LPN-friendly.

Candidate from the literature: quasi-cyclic codes (i.e., a code such that a cyclic shift by $s$ of a codeword is still a codeword, for some value $s$).

- **Resistant against LPN attacks:** highly plausible ✓ (was used in the design of several NIST proposals, e.g. BIKE, HQC, and LEDA, and are considered well studied)

- **Fast multiplication:** not too bad due to Fast Fourier Transform, $O(n \cdot \log n)$ ✓
We want: computing $H \cdot G$ is fast, and the code generated by $G$ is LPN-friendly.

Candidate from the literature: quasi-cyclic codes (i.e., a code such that a cyclic shift by $s$ of a codeword is still a codeword, for some value $s$).

- **Resistant against LPN attacks**: highly plausible (was used in the design of several NIST proposals, e.g. BIKE, HQC, and LEDA, and are considered well studied)
- **Fast multiplication**: not too bad due to Fast Fourier Transform, $O(n \cdot \log n)$

$O(n \cdot \log n)$ is not too bad, but when $n$ is huge, as in our scenario, it still gives a significant slowdown... Unfortunately, no existing well-understood ‘LPN-friendly’ candidate has linear time multiplication by $H$. So… What do we do?

We try to understand what makes a code ‘LPN-friendly’, and we craft our own!
## Security of (variants of) LPN - Linear Tests

A tremendous number of attacks on LPN have been published…

### Gaussian Elimination attacks
- Standard gaussian elimination
- Blum-Kalai-Wasserman [J.ACM:BKW03]
- Sample-efficient BKW [A-R:Lyu05]
- Pooled Gauss [CRYPTO:EKM17]
- Well-pooled Gauss [CRYPTO:EKM17]
- Leveil-Fouque [SCN:LF06]
- Covering codes [JC:GJL19]
- Covering codes+ [BT:15]
- Covering codes++ [BV:AC16]
- Covering codes+++ [EC:ZJW16]

### Information Set Decoding Attacks
- Prange’s algorithm [Prange62]
- Stern’s variant [ICIT:Stern88]
- Finiasz and Sendrier’s variant [AC:FS09]
- BJMM variant [EC:BJMM12]
- May-Ozerov variant [EC:MO15]
- Both-May variant [QIC:BM18]
- MMT variant [AC:MMT11]
- Well-pooled MMT [CRYPTO:EKM17]
- BLP variant [CRYPTO:BLP11]

### Other Attacks
- Generalized birthday [CRYPTO:Wag02]
- Improved GBA [Kirchner11]
- Linearization [EC:BM97]
- Linearization 2 [INDO:Saa07]
- Low-weight parity-check [Zichron17]
- Low-deg approx [ITCS:ABGKR17]

### Statistical Decoding Attacks
- Jabri’s attack [ICCC:Jab01]
- Overbeck’s variant [ACISP:Ove06]
- FKI’s variant [Trans.IT:FKI06]
- Debris-Tillich variant [ISIT:DT17]
A tremendous number of attacks on LPN have been published...

**Crucial observation:** *all* these attacks fit in the same framework, the *linear test framework.* (*)

### Game

1. Send $G$ to $\hat{A}$

2. $\hat{A}$ returns a test vector $\vec{v}'$ computed from $G$ in unbounded time

### Check

The adversary wins in the distribution induced by

$$\mathbf{G} \cdot \vec{v}' + \vec{0}$$

(over a random choice of secret and sparse noise) is non-negligibly biased.
A tremendous number of attacks on LPN have been published...

**Crucial observation:** all these attacks fit in the same framework, the *linear test framework.* (*)

**Game**
1. Send $H$ to $\mathcal{A}$
2. $\mathcal{A}$ returns a *test vector* $\vec{v}$ computed from $H$ in unbounded time

**Check**

The adversary wins in the distribution induced by $H \cdot \vec{v}$ computed from $H$ in unbounded time (over a random choice of secret and sparse noise) is non-negligibly *biased.*
**Crucial observation:** all these attacks fit in the same framework, the *linear test framework*. (*)

### Game

1. Send $H$ to $A$

2. $A$ returns a test vector $\vec{v}$ computed from $H$ in unbounded time

### Check

The adversary wins in the distribution induced by $\langle H, \vec{v} \rangle$ (over a random choice of secret and sparse noise) is non-negligibly biased.

### Linear Test Framework

A tremendous number of attacks on LPN have been published...

(*): highly structured algebraic codes (e.g. Reed-Solomon) are a different beast

### Gaussian Elimination attacks
- Standard gaussian elimination
- Blum-Kalai-Wasserman [J.ACM:BKW03]
- Sample-efficient BKW [A.R:Lyu05]
- Pooled Gauss [CRYPTO:EKM17]
- Well-pooled Gauss [CRYPTO:EKM17]
- Levieil-Fouque [SCN:LF06]
- Covering codes [JC:GJL19]
- Covering codes+ [BTV15]
- Covering codes++ [BV:AC16]
- Covering codes+++ [EC:ZJW16]

### Information Set Decoding Attacks
- Prange’s algorithm [Prange62]
- Stern’s variant [ICIT:Stern88]
- Finiasz and Sendrier’s variant [AC:FS09]
- BJMM variant [EC:BJMM12]
- May-Ozerov variant [EC:MO15]
- Both-May variant [PQC:BM18]
- MMT variant [AC:MMT11]
- Well-pooled MMT [CRYPTO:EKM17]
- BLP variant [CRYPTO:BLP11]

### Other Attacks
- Generalized birthday [CRYPTO:Wag02]
- Improved GBA [Kirchner11]
- Linearization [EC:BM97]
- Linearization 2 [INDO:Saa07]
- Low-weight parity-check [Zichron17]
- Low-deg approx [ITCS:ABGKR17]
The adversary wins in the distribution induced by

\[ \vec{v} \cdot \left( G \cdot \mu + \right) \]

(over a random choice of secret and sparse noise) is non-negligibly *biased*. 
The adversary wins in the distribution induced by
\[(\overrightarrow{v} \cdot G + \text{Induced by the noise vector})\]
(over a random choice of secret and sparse noise) is non-negligibly biased.

We have a sum of two distributions:
- Induced by the \textit{codeword}
- Induced by the \textit{noise vector}
A Sufficient Condition to Withstand all Linear Tests

The adversary wins in the distribution induced by
\[ \vec{v} \cdot (G \cdot \vec{s} + \vec{e}) \]
(over a random choice of secret and sparse noise) is non-negligibly biased.

**Claim:** Assume \( t \) (number of noisy coordinates) is set to a security parameter. If there is a constant \( c \) such that every subset of \( c \cdot n \) rows of \( G \) is linearly independent, no linear test can distinguish \( G \cdot \vec{s} + \vec{e} \) from random.

**We have a sum of two distributions:**
- Induced by the **codeword**
- Induced by the **noise vector**
A Sufficient Condition to Withstand all Linear Tests

The adversary wins in the distribution induced by

\[ \vec{v} \cdot \left( G \cdot \vec{s} + \vec{e} \right) \]

(over a random choice of secret and sparse noise) is non-negligibly biased.

**Claim:** Assume \( t \) (number of noisy coordinates) is set to a security parameter. If there is a constant \( c \) such that every subset of \( c \cdot n \) rows of \( G \) is linearly independent, no linear test can distinguish \( G \cdot \vec{s} + \vec{e} \) from random.

**Proof:** We consider two complementary cases for any possible attack vector \( \vec{v} \):

- Induced by the codeword
- Induced by the noise vector
A Sufficient Condition to Withstand all Linear Tests

The adversary wins in the distribution induced by

\[ \mathbf{v} \cdot (\mathbf{G} \cdot \mathbf{I} + \hat{\mathbf{e}}) \]

(over a random choice of secret and sparse noise) is non-negligibly biased.

**Claim:** Assume \( t \) (number of noisy coordinates) is set to a security parameter. If there is a constant \( c \) such that every subset of \( c \cdot n \) rows of \( \mathbf{G} \) is linearly independent, no linear test can distinguish \( \mathbf{G} \cdot \hat{\mathbf{s}} + \hat{\mathbf{e}} \) from random.

**Proof:** We consider two complementary cases for any possible attack vector \( \mathbf{v} \):

1. \( \text{HW}(\mathbf{v}) \leq c \cdot n \)

   If every subset of \( w \) rows of \( \mathbf{G} \) is linearly independent, then the distribution of \( (\mathbf{v} \cdot \mathbf{G}) \cdot \hat{\mathbf{s}} \) is truly random (as \( \hat{\mathbf{s}} \) is random and \( \mathbf{v} \cdot \mathbf{G} \) cannot be 0).
A Sufficient Condition to Withstand all Linear Tests

The adversary wins in the distribution induced by
\[
\vec{v} \cdot \left( G \cdot \left[ \begin{array}{c} 1 \\ 1 \\ \vdots \\ 1 \end{array} \right] + \right) \]
(over a random choice of secret and sparse noise) is non-negligibly biased.

**Claim:** Assume \( t \) (number of noisy coordinates) is set to a security parameter. If there is a constant \( c \) such that every subset of \( c \cdot n \) rows of \( G \) is linearly independent, no linear test can distinguish \( G \cdot \vec{s} + \vec{e} \) from random.

**Proof:** We consider two complementary cases for any possible attack vector \( \vec{v} \):

I. \( \text{HW}(\vec{v}) \leq c \cdot n \)
   If every subset of \( w \) rows of \( G \) is linearly independent, then the distribution of \( (\vec{v} \cdot G) \cdot \vec{s} \) is truly random (as \( \vec{s} \) is random and \( \vec{v} \cdot G \) cannot be \( 0 \)).

II. \( \text{HW}(\vec{v}) \geq c \cdot n \)
   The noise vector has \( t \) randomly chosen nonzero coordinates out of \( n \) entries. Each of them hits a nonzero entry of \( \vec{v} \) with proba \( \geq c \cdot n/n = c \), hence:
   \[
   \Pr[\vec{v} \cdot \vec{e} = 1] \geq \frac{1}{2} + (1 - c)^t \approx \frac{1}{2} + e^{-ct}
   \]
A Sufficient Condition to Withstand all Linear Tests

The adversary wins in the distribution induced by

\[
\vec{v} \cdot \left( \begin{pmatrix} G \\ \cdot \end{pmatrix} + \begin{pmatrix} \cdot \end{pmatrix} \right)
\]

(over a random choice of secret and sparse noise) is non-negligibly biased.

**We have a sum of two distributions:**

<table>
<thead>
<tr>
<th>Induced by the codeword</th>
<th>Induced by the noise vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>Protects against light linear tests</td>
<td>Protects against heavy linear tests</td>
</tr>
</tbody>
</table>

**Claim:** Assume \( t \) (number of noisy coordinates) is set to a security parameter. If there is a constant \( c \) such that every subset of \( c \cdot n \) rows of \( G \) is linearly independent, no linear test can distinguish \( G \cdot \vec{s} + \vec{e} \) from random.

**Proof:** We consider two complementary cases for any possible attack vector \( \vec{v} \):

**I.** \( \text{HW}(\vec{v}) \leq c \cdot n \)

If every subset of \( w \) rows of \( G \) is linearly independent, then the distribution of \( (\vec{v} \cdot G) \cdot \vec{s} \) is truly random (as \( \vec{s} \) is random and \( \vec{v} \cdot G \) cannot be \( 0 \)).

**II.** \( \text{HW}(\vec{v}) \geq c \cdot n \)

The noise vector has \( t \) randomly chosen nonzero coordinates out of \( n \) entries. Each of them hits a nonzero entry of \( \vec{v} \) with proba \( \geq c \cdot n / n = c \), hence:

\[
\Pr[\vec{v} \cdot \vec{e} = 1] \geq \frac{1}{2} + (1 - c)^t \approx \frac{1}{2} + e^{-ct}
\]
Rephrasing the Sufficient Condition

Every subset of $O(n)$ rows of $G$ is linearly independent
$\iff$ the left-kernel of $G$ does not contain nonzero vector of weight less than $O(n)$
$\iff$ the *dual code* of $G$, i.e., the code generated by the transpose of its parity check matrix $H$, has linear minimum distance

‘Provable’ candidates: recursive codes such as GDP, Spielman, Druk-Ishai (lack concrete efficiency).
Heuristic / experimental candidates: [CRYPTO:CRS21] (based on Tillick-Zémor LDPC codes)
More to come in the future
Rephrasing the Sufficient Condition

Every subset of $O(n)$ rows of $G$ is linearly independent
$\iff$ the left-kernel of $G$ does not contain nonzero vector of weight less than $O(n)$
$\iff$ the dual code of $G$, i.e., the code generated by the transpose of its parity check matrix $H$, has linear minimum distance

The expansion of the PCG boils down to the computation of

$$H \cdot \vec{e}$$

Where $\vec{e}$ is a very sparse vector, and (the shares of) the entries of $x \cdot \vec{e}$ can be computed individually in log-time.

‘Provable’ candidates: recursive codes such as GDP, Spielman, Druk-Ishai (lack concrete efficiency).
Heuristic / experimental candidates: [CRYPTO:CRS21] (based on Tillick-Zémor LDPC codes)
More to come in the future
Rephrasing the Sufficient Condition

Every subset of \( O(n) \) rows of \( G \) is linearly independent
\[ \iff \] the left-kernel of \( G \) does not contain nonzero vector of weight less than \( O(n) \)
\[ \iff \] the dual code of \( G \), i.e., the code generated by the transpose of its parity check matrix \( H \), has linear minimum distance

The expansion of the PCG boils down to the computation of

\[ H \cdot x \cdot \vec{e} \]

where \( \vec{e} \) is a very sparse vector, and (the shares of) the entries of \( x \cdot \vec{e} \) can be computed individually in log-time.

We want to find a matrix \( M = H^\top \) such that:

- The code generated by \( M \) is a good code
- Computing \( M^\top \cdot \vec{v} \) takes time \( O(n) \) for any \( \vec{v} \)

‘Provable’ candidates: recursive codes such as GDP, Spielman, Druk-Ishai (lack concrete efficiency).

Heuristic / experimental candidates: [CRYPTO:CRS21] (based on Tillick-Zémor LDPC codes)

More to come in the future
Rephrasing the Sufficient Condition

Every subset of $O(n)$ rows of $G$ is linearly independent
$\iff$ the left-kernel of $G$ does not contain nonzero vector of weight less than $O(n)$
$\iff$ the dual code of $G$, i.e., the code generated by the transpose of its parity check matrix $H$, has linear minimum distance

The expansion of the PCG boils down to the computation of

$$H \cdot x \cdot \vec{e}$$

Big random matrix

Where $\vec{e}$ is a very sparse vector, and (the shares of) the entries of $x \cdot \vec{e}$ can be computed individually in log-time.

We want to find a matrix $M = H^\top$ such that:

- The code generated by $M$ is a good code
- Computing $M^\top \cdot \vec{v}$ takes time $O(n)$ for any $\vec{v}$

This is the transposition principle

‘Provable’ candidates: recursive codes such as GDP, Spielman, Druk-Ishai (lack concrete efficiency).
Heuristic / experimental candidates: [CRYPTO:CRS21] (based on Tillick-Zémor LDPC codes)
More to come in the future
Rephrasing the Sufficient Condition

Every subset of \( O(n) \) rows of \( G \) is linearly independent
\[ \iff \text{the left-kernel of } G \text{ does not contain nonzero vector of weight less than } O(n) \]
\[ \iff \text{the dual code of } G, \text{ i.e., the code generated by the transpose of its parity check matrix } H, \text{ has linear minimum distance} \]

The expansion of the PCG boils down to the computation of

\[
H \cdot \hat{x} \cdot \hat{e}
\]

Big random matrix

Where \( \hat{e} \) is a very sparse vector, and (the shares of) the entries of \( x \cdot \hat{e} \) can be computed individually in log-time.

We want to find a matrix \( M = H^\top \) such that:

- The code generated by \( M \) is a good code
- Computing \( M^\top \vec{v} \) takes time \( O(n) \) for any \( \vec{v} \)
  \[ M \cdot \vec{v} \] (this is the transposition principle)

\[ \implies \text{We need to find a good and linear-time encodable code. And we want it concretely efficient!} \]

‘Provable’ candidates: recursive codes such as GDP, Spielman, Druk-Ishai (lack concrete efficiency).

Heuristic / experimental candidates: [CRYPTO:CRS21] (based on Tillick-Zémor LDPC codes)

More to come in the future
Every subset of $O(n)$ rows of $G$ is linearly independent
\[ \iff \text{the left-kernel of } G \text{ does not contain nonzero vector of weight less than } O(n) \]
\[ \iff \text{the dual code of } G, \text{i.e., the code generated by the transpose of its parity check matrix } H, \text{ has linear minimum distance} \]

In a sense, this is a (very partial) converse to the result, described by Benny last Monday, that this condition is also a necessary condition.

(Benny’s slide)

‘Provable’ candidates: recursive codes such as GDP, Spielman, Druk-Ishai (lack concrete efficiency).

Heuristic/experimental candidates: [CRYPTO: CRS21] (based on Tillick-Zémor LDPC codes)

More to come in the future
Rephrasing the Sufficient Condition

**Heuristic / experimental candidates:** [CRYPTO:CRS21] (based on Tillick-Zémor LDPC codes)
Rephrasing the Sufficient Condition

**Heuristic / experimental candidates:** [CRYPTO:CRS21] (based on Tillick-Zémor LDPC codes)

Computing $H$
Rephrasing the Sufficient Condition

**Heuristic / experimental candidates:** [CRYPTO:CRS21] (based on Tillich-Zémor LDPC codes)
### Rephrasing the Sufficient Condition

**Heuristic / experimental candidates:** [CRYPTO:CRS21] (based on Tillick-Zémor LDPC codes)

<table>
<thead>
<tr>
<th>Computing $H$</th>
<th>$\iff$ computing $H^\top$</th>
<th>$\iff$ computing $H^{\text{syst}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$H$</td>
<td>$H^\top$</td>
<td>$H^{\text{syst}}$</td>
</tr>
</tbody>
</table>

\[
H \iff H^\top \iff H^{\text{syst}} = \text{constant}
\]
Rephrasing the Sufficient Condition

**Heuristic / experimental candidates:** [CRYPTO: CRS21] (based on Tillick-Zémor LDPC codes)

Computing $H$ ⇔ computing $H^\top$ ⇔ computing $H^{syst}$

⇔ finding such that $M$ · = 0, where $M$ is the associated parity-check matrix
Rephrasing the Sufficient Condition

**Heuristic / experimental candidates:** [CRYPTO:CRS21] (based on Tillich-Zémor LDPC codes)

- **Core idea:** use a sparse $M$ which can be brought in approximate lower triangular form:
  - We have fast encoder for such parity-check matrices
  - We have good insights on the minimum distance of the associated code, e.g. Tillich-Zémor, ISIT’06
Rephrasing the Sufficient Condition

**Heuristic / experimental candidates:** [CRYPTO:CRS21] (based on Tillich-Zémor LDPC codes)

Computing $H$ ⟷ computing $H^\top$ ⟷ computing $H^{\text{syst}}$  =  

$\iff$ finding such that $M$  =  0, where $M$ is the associated parity-check matrix

**Core idea:** use a sparse $M$ which can be brought in approximate lower triangular form:
- We have fast encoder for such parity-check matrices
- We have good insights on the minimum distance of the associated code, e.g. Tillich-Zémor, ISIT’06

$$M = \begin{bmatrix} A & B \\ D & E \end{bmatrix} \begin{bmatrix} C & \end{bmatrix}$$
Rephrasing the Sufficient Condition

**Heuristic / experimental candidates:** [CRYPTO:CRS21] (based on Tillich-Zémor LDPC codes)

Computing $H \cdot \iff$ computing $H^\top \cdot \iff$ computing $H_{\text{syst}} \cdot =$

$\iff$ finding such that $M \cdot = 0$, where $M$ is the associated parity-check matrix

**Core idea:** use a sparse $M$ which can be brought in approximate lower triangular form:
- We have fast encoder for such parity-check matrices
- We have good insights on the minimum distance of the associated code, e.g. Tillich-Zémor, ISIT’06

$M = \begin{array}{ccc}
A & B & C \\
D & E & F \\
g & g &
\end{array}$

$\implies$ encode in time $O(n + g^2)$, linear if $g < \sqrt{n}$. 
Questions?
Backup Slides
Secure Computation from Oblivious Transfer

**Warm-up I: 2-Party Product Sharing**

\[(y_1, y_2) \text{ random conditioned on } y_1 \oplus y_2 = x_1 x_2\]

**Step-by-Step Solution**

- We use an OT functionality where Alice is the receiver, and her selection bit is her input. \(x_2\)
- What should be Bob's input? Let’s work out the equation:

\[
\begin{align*}
    s_{x_2} &= x_2 \cdot s_1 + (1 - x_2) \cdot s_0 \\
    &= x_2 \cdot s_1 \oplus (1 \oplus x_2) \cdot s_0 \\
    &= s_0 \oplus (s_0 \oplus s_1) \cdot x_2
\end{align*}
\]

This should be \(x_1\)

\((s_0, s_1)\) are (2,2)-shares of \(x_1\).

**Warm-up II: Variant**

This time, Alice and Bob start with shares of \((x, y)\), and want to compute shares of the product \(x \cdot y\)

\[(a_1, b_1) \text{ are shares of } x\]
\[(a_2, b_2) \text{ are shares of } y\]
\[(z_1, z_2) \text{ are random shares of } z = x \cdot y\]

**Solution**

\[
\begin{align*}
    x \cdot y &= (a_1 + b_1) \cdot (a_2 + b_2) \\
    &= a_1 \cdot a_2 + a_1 \cdot b_2 + a_2 \cdot b_1 + b_1 \cdot b_2
\end{align*}
\]

Value known to Alice

Value known to Bob

Each of these values is the product of a value known to Alice and a value known to Bob