LESS: Digital Signatures from Linear Code Equivalence

NIST PQC Seminars

Marco Baldi, Alessandro Barenghi, Jean-François Biasse, Andre Esser,

Gerardo Pelosi, **Edoardo Persichetti**, Markku-J. O. Saarinen, Paolo Santini

14 March 2023

In This Talk
Roadmap

- Background
- Code-based Signatures
- Group Actions
- LESS
- Considerations

Roadmap

- Background
- Code-based Signatures
- Group Actions
- LESS
- Considerations

$[n, k]$ Linear Code over $\mathbb{F}_q$

A subspace of dimension k of $\mathbb{F}_q^n$. Value n is called length.

$[n, k]$ Linear Code over $\mathbb{F}_q$

A subspace of dimension k of $\mathbb{F}_q^n$. Value n is called length.

Hamming Metric

$wt(x) = |\{i : x_i \neq 0, 1 \leq i \leq n\}|$, $d(x, y) = wt(x - y)$.
Minimum distance (of $\mathfrak{C}$): $\min\{d(x, y) : x, y \in \mathfrak{C}\}$.

$[n, k]$ Linear Code over $\mathbb{F}_q$

A subspace of dimension k of $\mathbb{F}_q^n$. Value n is called length.

Hamming Metric

$wt(x) = |\{i : x_i \neq 0, 1 \leq i \leq n\}|$, $d(x, y) = wt(x - y)$.
Minimum distance (of $\mathfrak{C}$): $\min\{d(x, y) : x, y \in \mathfrak{C}\}$.

Generator Matrix

$G \in \mathbb{F}_q^{k \times n}$ defines the code as : $x \in \mathfrak{C} \iff x = uG$ for $u \in \mathbb{F}_q^k$.
Not unique: $SG, S \in GL(k, q)$; Systematic form: $(I_k|M)$.

$[n,k]$ Linear Code over $\mathbb{F}_q$

A subspace of dimension k of $\mathbb{F}_q^n$. Value n is called length.

Hamming Metric

$wt(x) = |\{i : x_i \neq 0, 1 \leq i \leq n\}|$, $d(x,y) = wt(x-y)$.
Minimum distance (of $\mathfrak{C}$): $\min\{d(x,y) : x, y \in \mathfrak{C}\}$.

Generator Matrix

$G \in \mathbb{F}_q^{k \times n}$ defines the code as : $x \in \mathfrak{C} \iff x = uG$ for $u \in \mathbb{F}_q^k$.
Not unique: $SG, S \in GL(k,q)$; Systematic form: $(I_k|M)$.

Parity-check Matrix

$H \in \mathbb{F}_q^{(n-k)\times n}$ defines the code as: $x \in \mathfrak{C} \iff Hx^T = 0$ (syndrome).
Not unique: $SH, S \in GL(n-k,q)$; Systematic form: $(M^T|I_{n-k})$.

$[n,k]$ Linear Code over $\mathbb{F}_q$

A subspace of dimension k of $\mathbb{F}_q^n$. Value n is called length.

Hamming Metric

$wt(x) = |\{i : x_i \neq 0, 1 \leq i \leq n\}|$, $d(x,y) = wt(x-y)$.
Minimum distance (of $\mathfrak{C}$): $\min\{d(x,y) : x, y \in \mathfrak{C}\}$.

Generator Matrix

$G \in \mathbb{F}_q^{k \times n}$ defines the code as : $x \in \mathfrak{C} \iff x = uG$ for $u \in \mathbb{F}_q^k$.
Not unique: SG, $S \in GL(k,q)$; Systematic form: $(I_k|M)$.

Parity-check Matrix

$H \in \mathbb{F}_q^{(n-k)\times n}$ defines the code as: $x \in \mathfrak{C} \iff Hx^T = 0$ (syndrome).
Not unique: SH, $S \in GL(n-k,q)$; Systematic form: $(M^T|I_{n-k})$.

w-error correcting: $\exists$ algorithm that corrects up to w errors.

Example: Goppa Codes

Select $g(X) \in \mathbb{F}_{q^m}[X]$ and non-zero $\alpha_1, \ldots, \alpha_n \in \mathbb{F}_{q^m}$ with $g(\alpha_i) \neq 0$.

Parity-check given by $H = \{H_{ij}\} = \{\alpha_j^{i-1}/g(\alpha_j)\}$. Codewords over $\mathbb{F}_q$.

Let noisy codeword be $y = x + e$, $x \in \mathfrak{C}$, $wt(e) \leq w$.

For Goppa codes, $w = r/2$ (or $w = r$ if binary), where $r = deg(g)$.

Example: Goppa Codes

Select $g(X) \in \mathbb{F}_{q^m}[X]$ and non-zero $\alpha_1, \ldots, \alpha_n \in \mathbb{F}_{q^m}$ with $g(\alpha_i) \neq 0$.

Parity-check given by $H = \{H_{ij}\} = \{\alpha_j^{i-1}/g(\alpha_j)\}$. Codewords over $\mathbb{F}_q$.

Let noisy codeword be $y = x + e$, $x \in \mathfrak{C}$, $wt(e) \leq w$.

For Goppa codes, $w = r/2$ (or $w = r$ if binary), where $r = deg(g)$.

To decode:

1. Compute syndrome $s = Hy^T = (s_0, \ldots, s_{r-1})$.
2. Obtain *error locator* poly $\sigma(X)$ and *error evaluator* poly $\omega(X)$ by solving *key equation*
$$\frac{\omega(X)}{\sigma(X)} \equiv s(X) \mod X^r.$$
3. Find roots; error positions are reciprocals (values from $\omega(X)$).

Decoding Problems

In general, it is hard to decode random codes.

In general, it is hard to decode random codes.

General Decoding Problem (GDP)

Given: $G \in \mathbb{F}_q^{k\times n}$, $y \in \mathbb{F}_q^n$ and $w \in \mathbb{N}$.
Goal: find a word $e \in \mathbb{F}_q^n$ with $wt(e) \leq w$ such that $y - e = x \in \mathfrak{C}_G$.

In general, it is hard to decode random codes.

General Decoding Problem (GDP)

Given: $G \in \mathbb{F}_q^{k\times n}$, $y \in \mathbb{F}_q^n$ and $w \in \mathbb{N}$.
Goal: find a word $e \in \mathbb{F}_q^n$ with $wt(e) \leq w$ such that $y - e = x \in \mathfrak{C}_G$.

Easy to see this is equivalent to the following.

Decoding Problems

In general, it is hard to decode random codes.

General Decoding Problem (GDP)

Given: $G \in \mathbb{F}_q^{k \times n}$, $y \in \mathbb{F}_q^n$ and $w \in \mathbb{N}$.
Goal: find a word $e \in \mathbb{F}_q^n$ with $wt(e) \leq w$ such that $y - e = x \in \mathfrak{C}_G$.

Easy to see this is equivalent to the following.

Syndrome Decoding Problem (SDP)

Given: $H \in \mathbb{F}_q^{(n-k) \times n}$, $y \in \mathbb{F}_q^{(n-k)}$ and $w \in \mathbb{N}$.
Goal: find a word $e \in \mathbb{F}_q^n$ with $wt(e) \leq w$ such that $He^T = y$.

Decoding Problems

In general, it is hard to decode random codes.

General Decoding Problem (GDP)

Given: $G \in \mathbb{F}_q^{k \times n}$, $y \in \mathbb{F}_q^n$ and $w \in \mathbb{N}$.
Goal: find a word $e \in \mathbb{F}_q^n$ with $wt(e) \leq w$ such that $y - e = x \in \mathfrak{C}_G$.

Easy to see this is equivalent to the following.

Syndrome Decoding Problem (SDP)

Given: $H \in \mathbb{F}_q^{(n-k) \times n}$, $y \in \mathbb{F}_q^{(n-k)}$ and $w \in \mathbb{N}$.
Goal: find a word $e \in \mathbb{F}_q^n$ with $wt(e) \leq w$ such that $He^T = y$.

NP-Complete (Berlekamp, McEliece and Van Tilborg, 1978; Barg, 1994).

In general, it is hard to decode random codes.

General Decoding Problem (GDP)

Given: $G \in \mathbb{F}_q^{k \times n}$, $y \in \mathbb{F}_q^n$ and $w \in \mathbb{N}$.
Goal: find a word $e \in \mathbb{F}_q^n$ with $wt(e) \leq w$ such that $y - e = x \in \mathfrak{C}_G$.

Easy to see this is equivalent to the following.

Syndrome Decoding Problem (SDP)

Given: $H \in \mathbb{F}_q^{(n-k) \times n}$, $y \in \mathbb{F}_q^{(n-k)}$ and $w \in \mathbb{N}$.
Goal: find a word $e \in \mathbb{F}_q^n$ with $wt(e) \leq w$ such that $He^T = y$.

NP-Complete (Berlekamp, McEliece and Van Tilborg, 1978; Barg, 1994).
Unique solution when w is below a certain threshold.

In general, it is hard to decode random codes.

General Decoding Problem (GDP)

Given: $G \in \mathbb{F}_q^{k \times n}$, $y \in \mathbb{F}_q^n$ and $w \in \mathbb{N}$.
Goal: find a word $e \in \mathbb{F}_q^n$ with $wt(e) \leq w$ such that $y - e = x \in \mathfrak{C}_G$.

Easy to see this is equivalent to the following.

Syndrome Decoding Problem (SDP)

Given: $H \in \mathbb{F}_q^{(n-k) \times n}$, $y \in \mathbb{F}_q^{(n-k)}$ and $w \in \mathbb{N}$.
Goal: find a word $e \in \mathbb{F}_q^n$ with $wt(e) \leq w$ such that $He^T = y$.

NP-Complete (Berlekamp, McEliece and Van Tilborg, 1978; Barg, 1994).
Unique solution when w is below a certain threshold.

Gilbert-Varshamov (GV) Bound

For a given finite field $\mathbb{F}_q$ and integers n, k, the Gilbert-Varshamov (GV) distance is the largest integer d_0 such that

$$|\mathcal{B}(0, d_0 - 1)| \leq q^{n-k}.$$

In general, it is hard to decode random codes.

General Decoding Problem (GDP)

Given: $G \in \mathbb{F}_q^{k\times n}$, $y \in \mathbb{F}_q^n$ and $w \in \mathbb{N}$.
Goal: find a word $e \in \mathbb{F}_q^n$ with $wt(e) \leq w$ such that $y - e = x \in \mathfrak{C}_G$.

Easy to see this is equivalent to the following.

Syndrome Decoding Problem (SDP)

Given: $H \in \mathbb{F}_q^{(n-k)\times n}$, $y \in \mathbb{F}_q^{(n-k)}$ and $w \in \mathbb{N}$.
Goal: find a word $e \in \mathbb{F}_q^n$ with $wt(e) \leq w$ such that $He^T = y$.

NP-Complete (Berlekamp, McEliece and Van Tilborg, 1978; Barg, 1994).
Unique solution when w is below a certain threshold.

Gilbert-Varshamov (GV) Bound

For a given finite field $\mathbb{F}_q$ and integers n, k, the Gilbert-Varshamov (GV) distance is the largest integer d_0 such that

$$|\mathcal{B}(0, d_0 - 1)| \leq q^{n-k}.$$

Very well-studied, solid security understanding (ISD).

What is Code-Based Cryptography?

The family of primitives based on hard problems from coding theory.

The family of primitives based on hard problems from coding theory.

If trapdoor is required (e.g. encryption), need one more ingredient.

What is Code-Based Cryptography?

The family of primitives based on hard problems from coding theory.

If trapdoor is required (e.g. encryption), need one more ingredient.

Assumption (Code Indistinguishability)

Let M be a matrix defining a code. Then M is indistinguishable from a randomly generated matrix of the same size.

The family of primitives based on hard problems from coding theory.

If trapdoor is required (e.g. encryption), need one more ingredient.

Assumption (Code Indistinguishability)

Let M be a matrix defining a code. Then M is indistinguishable from a randomly generated matrix of the same size.

Choose a code family with efficient decoding algorithm associated to description Δ and hide the structure.

What is Code-Based Cryptography?

The family of primitives based on hard problems from coding theory.

If trapdoor is required (e.g. encryption), need one more ingredient.

Assumption (Code Indistinguishability)

Let M be a matrix defining a code. Then M is indistinguishable from a randomly generated matrix of the same size.

Choose a code family with efficient decoding algorithm associated to description Δ and hide the structure.

Example (McEliece/Niederreiter): use change of basis S and permutation P to obtain equivalent code.

The family of primitives based on hard problems from coding theory.

If trapdoor is required (e.g. encryption), need one more ingredient.

Assumption (Code Indistinguishability)

Let M be a matrix defining a code. Then M is indistinguishable from a randomly generated matrix of the same size.

Choose a code family with efficient decoding algorithm associated to description Δ and hide the structure.

Example (McEliece/Niederreiter): use change of basis S and permutation P to obtain equivalent code.

Hardness of assumption depends on chosen code family.

Roadmap

- Background
- Code-based Signatures
- Group Actions
- LESS
- Considerations

Use hash-and-sign framework as in e.g. Full Domain Hash (RSA).

Idea 1: Trapdoor-based Schemes

Use hash-and-sign framework as in e.g. Full Domain Hash (RSA).

Given message msg, trapdoor OW function f and hash function $Hash$.

Idea 1: Trapdoor-based Schemes

Use hash-and-sign framework as in e.g. Full Domain Hash (RSA).

Given message msg, trapdoor OW function f and hash function $Hash$.

Create signature $\sigma = f^{-1}(td, Hash(msg))$. Verify if $f(\sigma) = Hash(msg)$.

Idea 1: Trapdoor-based Schemes

Use hash-and-sign framework as in e.g. Full Domain Hash (RSA).

Given message msg, trapdoor OW function f and hash function $Hash$.

Create signature $\sigma = f^{-1}(td, Hash(msg))$. Verify if $f(\sigma) = Hash(msg)$.

For CBC, traditional SDP-based trapdoor is decoding: CFS scheme.

(Courtois, Finiasz, Sendrier, 2001)

Idea 1: Trapdoor-based Schemes

Use hash-and-sign framework as in e.g. Full Domain Hash (RSA).

Given message msg, trapdoor OW function f and hash function $Hash$.

Create signature $\sigma = f^{-1}(td, Hash(msg))$. Verify if $f(\sigma) = Hash(msg)$.

For CBC, traditional SDP-based trapdoor is decoding: CFS scheme.

(Courtois, Finiasz, Sendrier, 2001)

...except, domain is not “full”.

Idea 1: Trapdoor-based Schemes

Use hash-and-sign framework as in e.g. Full Domain Hash (RSA).

Given message msg, trapdoor OW function f and hash function $Hash$.

Create signature $\sigma = f^{-1}(td, Hash(msg))$. Verify if $f(\sigma) = Hash(msg)$.

For CBC, traditional SDP-based trapdoor is decoding: CFS scheme.

(Courtois, Finiasz, Sendrier, 2001)

...except, domain is not "full".

Complex sampling leads to slow signing, large keys and potential weaknesses.

(Bleichenbacher, 2009; Faugère Gauthier-Umana, Otmani, Perret, Tillich, 2013; Landais, Sendrier, 2012; Bernstein, Chou, Schwabe, 2013)

Idea 1: Trapdoor-based Schemes

Use hash-and-sign framework as in e.g. Full Domain Hash (RSA).

Given message msg, trapdoor OW function f and hash function $Hash$.

Create signature $\sigma = f^{-1}(td, Hash(msg))$. Verify if $f(\sigma) = Hash(msg)$.

For CBC, traditional SDP-based trapdoor is decoding: CFS scheme.

(Courtois, Finiasz, Sendrier, 2001)

...except, domain is not "full".

Complex sampling leads to slow signing, large keys and potential weaknesses.

(Bleichenbacher, 2009; Faugère Gauthier-Umana, Otmani, Perret, Tillich, 2013; Landais, Sendrier, 2012; Bernstein, Chou, Schwabe, 2013)

Recent renditions show great improvements, but still exhibit similar features.

(Debris-Alazard, Sendrier, Tillich, 2018)

Idea 2: Zero-Knowledge Protocols

ZKIDs can be turned into signature schemes using Fiat-Shamir transformation.

Idea 2: Zero-Knowledge Protocols

ZKIDs can be turned into signature schemes using Fiat-Shamir transformation.

This method is very promising and usually leads to efficient schemes.

(Schnorr, 1989;...)

Idea 2: Zero-Knowledge Protocols

ZKIDs can be turned into signature schemes using Fiat-Shamir transformation.

This method is very promising and usually leads to efficient schemes.

(Schnorr, 1989;...)

Strong security guarantees. No trapdoor is required!

Idea 2: Zero-Knowledge Protocols

ZKIDs can be turned into signature schemes using Fiat-Shamir transformation.

This method is very promising and usually leads to efficient schemes.

(Schnorr, 1989;...)

Strong security guarantees. No trapdoor is required!

For CBC, can avoid decoding: rely directly on SDP.

Idea 2: Zero-Knowledge Protocols

ZKIDs can be turned into signature schemes using Fiat-Shamir transformation.

This method is very promising and usually leads to efficient schemes.
(Schnorr, 1989;...)

Strong security guarantees. No trapdoor is required!

For CBC, can avoid decoding: rely directly on SDP.

Use random codes and exploit hardness of finding low-weight words.
(Stern, 1993;...)

Idea 2: Zero-Knowledge Protocols

ZKIDs can be turned into signature schemes using Fiat-Shamir transformation.

This method is very promising and usually leads to efficient schemes.

(Schnorr, 1989;...)

Strong security guarantees. No trapdoor is required!

For CBC, can avoid decoding: rely directly on SDP.

Use random codes and exploit hardness of finding low-weight words.

(Stern, 1993;...)

High soundness error requires several repetitions to achieve security.

Idea 2: Zero-Knowledge Protocols

ZKIDs can be turned into signature schemes using Fiat-Shamir transformation.

This method is very promising and usually leads to efficient schemes.

(Schnorr, 1989;...)

Strong security guarantees. No trapdoor is required!

For CBC, can avoid decoding: rely directly on SDP.

Use random codes and exploit hardness of finding low-weight words.

(Stern, 1993;...)

High soundness error requires several repetitions to achieve security.

Due to protocol structure and nature of objects, this results in rather large signatures (e.g. > 20 kB for 128 sec. bits).

Roadmap

- Background
- Code-based Signatures
- **Group Actions**
- LESS
- Considerations

Group Action

Let $\mathcal{X}$ be a set and $(\mathcal{G}, \cdot)$ be a group. A group action is a mapping

$$\begin{aligned} \star : \quad \mathcal{G} \times \mathcal{X} &\to \mathcal{X} \\ (g, x) &\mapsto g \star x \end{aligned}$$

such that, for all $x \in \mathcal{X}$ and $g_1, g_2 \in \mathcal{G}$, $g_2 \star (g_1 \star x) = (g_2 \cdot g_1) \star x$.

Group Action

Let $\mathcal{X}$ be a set and $(\mathcal{G}, \cdot)$ be a group. A group action is a mapping

$$\begin{array}{rccc} \star : & \mathcal{G} \times \mathcal{X} & \to & \mathcal{X} \\ & (g, x) & \mapsto & g \star x \end{array}$$

such that, for all $x \in \mathcal{X}$ and $g_1, g_2 \in \mathcal{G}$, $g_2 \star (g_1 \star x) = (g_2 \cdot g_1) \star x$.

The word cryptographic means that it has some properties of interest in cryptography, e.g.:

Group Action

Let $\mathcal{X}$ be a set and $(\mathcal{G}, \cdot)$ be a group. A group action is a mapping

$$\begin{aligned} \star : \quad \mathcal{G} \times \mathcal{X} &\rightarrow \quad \mathcal{X} \\ (g, x) &\mapsto \quad g \star x \end{aligned}$$

such that, for all $x \in \mathcal{X}$ and $g_1, g_2 \in \mathcal{G}$, $g_2 \star (g_1 \star x) = (g_2 \cdot g_1) \star x$.

The word cryptographic means that it has some properties of interest in cryptography, e.g.:

- Efficient evaluation, sampling and membership testing algorithms.

Group Action

Let $\mathcal{X}$ be a set and $(\mathcal{G}, \cdot)$ be a group. A group action is a mapping

$$\begin{aligned} \star : \quad \mathcal{G} \times \mathcal{X} &\rightarrow \quad \mathcal{X} \\ (g, x) &\mapsto \quad g \star x \end{aligned}$$

such that, for all $x \in \mathcal{X}$ and $g_1, g_2 \in \mathcal{G}$, $g_2 \star (g_1 \star x) = (g_2 \cdot g_1) \star x$.

The word cryptographic means that it has some properties of interest in cryptography, e.g.:

- Efficient evaluation, sampling and membership testing algorithms.
- A hard vectorization problem.

Group Action

Let $\mathcal{X}$ be a set and $(\mathcal{G}, \cdot)$ be a group. A group action is a mapping

$$\begin{aligned} \star : \quad \mathcal{G} \times \mathcal{X} &\rightarrow \mathcal{X} \\ (g, x) &\mapsto g \star x \end{aligned}$$

such that, for all $x \in \mathcal{X}$ and $g_1, g_2 \in \mathcal{G}$, $g_2 \star (g_1 \star x) = (g_2 \cdot g_1) \star x$.

The word cryptographic means that it has some properties of interest in cryptography, e.g.:

- Efficient evaluation, sampling and membership testing algorithms.
- A hard vectorization problem.

Group Action Vectorization Problem

Given the pair $x_1, x_2 \in \mathcal{X}$, find, if any, $g \in \mathcal{G}$ such that $g \star x_1 = x_2$.

Let $\mathcal{X}$ be a group of prime order p and $\mathcal{G} = \mathbb{Z}_p^*$.

Let $\mathcal{X}$ be a group of prime order p and $\mathcal{G} = \mathbb{Z}_p^*$.

Then the vectorization problem is exactly DLP in $\mathcal{X}$.

Famous Examples

Let $\mathcal{X}$ be a group of prime order p and $\mathcal{G} = \mathbb{Z}_p^*$.

Then the vectorization problem is exactly DLP in $\mathcal{X}$.

A huge amount of cryptography has been built using this simple, but very special group action!

Famous Examples

Let $\mathcal{X}$ be a group of prime order p and $\mathcal{G} = \mathbb{Z}_p^*$.

Then the vectorization problem is exactly DLP in $\mathcal{X}$.

A huge amount of cryptography has been built using this simple, but very special group action!

Choosing the set $\mathcal{X}$ with this extra structure comes with several advantages and disadvantages.

Famous Examples

Let $\mathcal{X}$ be a group of prime order p and $\mathcal{G} = \mathbb{Z}_p^*$.

Then the vectorization problem is exactly DLP in $\mathcal{X}$.

A huge amount of cryptography has been built using this simple, but very special group action!

Choosing the set $\mathcal{X}$ with this extra structure comes with several advantages and disadvantages.

- Useful properties (e.g. commutativity) and design options.

Famous Examples

Let $\mathcal{X}$ be a group of prime order p and $\mathcal{G} = \mathbb{Z}_p^*$.

Then the vectorization problem is exactly DLP in $\mathcal{X}$.

A huge amount of cryptography has been built using this simple, but very special group action!

Choosing the set $\mathcal{X}$ with this extra structure comes with several advantages and disadvantages.

- Useful properties (e.g. commutativity) and design options.
- Not post-quantum!

Famous Examples

Let $\mathcal{X}$ be a group of prime order p and $\mathcal{G} = \mathbb{Z}_p^*$.

Then the vectorization problem is exactly DLP in $\mathcal{X}$.

A huge amount of cryptography has been built using this simple, but very special group action!

Choosing the set $\mathcal{X}$ with this extra structure comes with several advantages and disadvantages.

- Useful properties (e.g. commutativity) and design options.
- Not post-quantum!

Recently, isogeny-based group actions have captivated the cryptographic scene, showing a unique performance profile.

Let $\mathcal{X}$ be a group of prime order p and $\mathcal{G} = \mathbb{Z}_p^*$.

Then the vectorization problem is exactly DLP in $\mathcal{X}$.

A huge amount of cryptography has been built using this simple, but very special group action!

Choosing the set $\mathcal{X}$ with this extra structure comes with several advantages and disadvantages.

- Useful properties (e.g. commutativity) and design options.
- Not post-quantum!

Recently, isogeny-based group actions have captivated the cryptographic scene, showing a unique performance profile.

What about group actions from coding theory?

Three types:

- Permutations: $\pi\big((a_1, a_2, \ldots, a_n)\big) = \big(a_{\pi(1)}, a_{\pi(2)}, \ldots, a_{\pi(n)}\big)$.

Three types:

- Permutations: $\pi\big((a_1, a_2, \dots, a_n)\big) = \big(a_{\pi(1)}, a_{\pi(2)}, \dots, a_{\pi(n)}\big)$.
- Monomials: permutations + scaling factors: $\mu = (v; \pi)$, with $v \in (\mathbb{F}_q^*)^n$

$$\mu\big((a_1, a_2, \dots, a_n)\big) = \big(v_1 \cdot a_{\pi(1)}, v_2 \cdot a_{\pi(2)}, \dots, v_n \cdot a_{\pi(n)}\big)$$

Monomial matrix: permutation $\times$ diagonal.

Three types:

- Permutations: $\pi\big(\,(a_1, a_2, \ldots, a_n)\,\big) = \big(a_{\pi(1)}, a_{\pi(2)}, \ldots, a_{\pi(n)}\big)$.
- Monomials: permutations + scaling factors: $\mu = (v; \pi)$, with $v \in (\mathbb{F}_q^*)^n$

  $$\mu\big(\,(a_1, a_2, \ldots, a_n)\,\big) = \big(v_1 \cdot a_{\pi(1)}, v_2 \cdot a_{\pi(2)}, \ldots, v_n \cdot a_{\pi(n)}\big)$$

  Monomial matrix: permutation $\times$ diagonal.
- Monomials + field automorphism.

Three types:

- Permutations: $\pi\big((a_1, a_2, \dots, a_n)\big) = \big(a_{\pi(1)}, a_{\pi(2)}, \dots, a_{\pi(n)}\big)$.
- Monomials: permutations + scaling factors: $\mu = (v; \pi)$, with $v \in (\mathbb{F}_q^*)^n$

$$\mu\big((a_1, a_2, \dots, a_n)\big) = \big(v_1 \cdot a_{\pi(1)}, v_2 \cdot a_{\pi(2)}, \dots, v_n \cdot a_{\pi(n)}\big)$$

  Monomial matrix: permutation $\times$ diagonal.

- Monomials + field automorphism.

Two codes are equivalent if they are connected by an isometry.

Three types:

- Permutations: $\pi\big((a_1, a_2, \ldots, a_n)\big) = \big(a_{\pi(1)}, a_{\pi(2)}, \ldots, a_{\pi(n)}\big)$.
- Monomials: permutations + scaling factors: $\mu = (v; \pi)$, with $v \in (\mathbb{F}_q^*)^n$

$$\mu\big((a_1, a_2, \ldots, a_n)\big) = \big(v_1 \cdot a_{\pi(1)}, v_2 \cdot a_{\pi(2)}, \ldots, v_n \cdot a_{\pi(n)}\big)$$

  Monomial matrix: permutation $\times$ diagonal.

- Monomials + field automorphism.

Two codes are equivalent if they are connected by an isometry.

We talk about permutation, linear and semilinear equivalence, respectively.

Code equivalence can be described using generator (or parity-check) matrices. Clearly:

Code equivalence can be described using generator (or parity-check) matrices. Clearly:

$$\mathfrak{C}_0 \overset{\text{PE}}{\sim} \mathfrak{C}_1 \iff \exists (S, P) \in \mathrm{GL}_k(q) \times S_n \;\text{ s.t. }\; G_1 = SG_0P,$$
$$\mathfrak{C}_0 \overset{\text{LE}}{\sim} \mathfrak{C}_1 \iff \exists (S, Q) \in \mathrm{GL}_k(q) \times M_n(q) \;\text{ s.t. }\; G_1 = SG_0Q,$$

where P is a permutation matrix, and Q a monomial matrix.

Code equivalence can be described using generator (or parity-check) matrices. Clearly:

$$\mathfrak{C}_0 \overset{\text{PE}}{\sim} \mathfrak{C}_1 \iff \exists (S, P) \in \text{GL}_k(q) \times S_n \;\text{ s.t. }\; G_1 = SG_0P,$$
$$\mathfrak{C}_0 \overset{\text{LE}}{\sim} \mathfrak{C}_1 \iff \exists (S, Q) \in \text{GL}_k(q) \times M_n(q) \;\text{ s.t. }\; G_1 = SG_0Q,$$

where P is a permutation matrix, and Q a monomial matrix.

Can be seen as a group action of $\mathcal{G} = \text{GL}_k(q) \times M_n(q)$ on full-rank matrices in $\mathbb{F}_q^{k\times n}$.

Code equivalence can be described using generator (or parity-check) matrices. Clearly:

$$\mathfrak{C}_0 \overset{\mathsf{PE}}{\sim} \mathfrak{C}_1 \iff \exists (S, P) \in \mathsf{GL}_k(q) \times S_n \text{ s.t. } G_1 = SG_0P,$$
$$\mathfrak{C}_0 \overset{\mathsf{LE}}{\sim} \mathfrak{C}_1 \iff \exists (S, Q) \in \mathsf{GL}_k(q) \times M_n(q) \text{ s.t. } G_1 = SG_0Q,$$

where P is a permutation matrix, and Q a monomial matrix.

Can be seen as a group action of $\mathcal{G} = \mathsf{GL}_k(q) \times M_n(q)$ on full-rank matrices in $\mathbb{F}_q^{k \times n}$.

Code-based Group Action

$$\begin{array}{rccc} \star : & \mathcal{G} \times \mathcal{X} & \rightarrow & \mathcal{X} \\ & ((S, Q), G_0) & \mapsto & SG_0Q \end{array}$$

Can imagine $\mathcal{G}$ acting on codes if we choose canonical representation, i.e. systematic form.

Code equivalence can be described using generator (or parity-check) matrices. Clearly:

$$\mathfrak{C}_0 \overset{\text{PE}}{\sim} \mathfrak{C}_1 \iff \exists (S, P) \in \mathrm{GL}_k(q) \times S_n \;\text{ s.t. }\; G_1 = SG_0P,$$
$$\mathfrak{C}_0 \overset{\text{LE}}{\sim} \mathfrak{C}_1 \iff \exists (S, Q) \in \mathrm{GL}_k(q) \times M_n(q) \;\text{ s.t. }\; G_1 = SG_0Q,$$

where P is a permutation matrix, and Q a monomial matrix.

Can be seen as a group action of $\mathcal{G} = \mathrm{GL}_k(q) \times M_n(q)$ on full-rank matrices in $\mathbb{F}_q^{k\times n}$.

Code-based Group Action

$$\begin{array}{rccc} \star : & \mathcal{G} \times \mathcal{X} & \to & \mathcal{X} \\ & ((S, Q), G_0) & \mapsto & SG_0Q \end{array}$$

Can imagine $\mathcal{G}$ acting on codes if we choose canonical representation, i.e. systematic form.

In practice, we consider simply $RREF(G_0Q)$.

The problem of deciding if two codes are equivalent is well-known in coding theory.

The problem of deciding if two codes are equivalent is well-known in coding theory.

For our purpose, we are interested in the computational version: this is the vectorization problem for our action.

The problem of deciding if two codes are equivalent is well-known in coding theory.

For our purpose, we are interested in the computational version: this is the vectorization problem for our action.

Permutation Equivalence Problem (PEP)

Given $\mathfrak{C}_0, \mathfrak{C}_1 \subseteq \mathbb{F}_q^n$, find a permutation π such that $\pi(\mathfrak{C}_0) = \mathfrak{C}_1$. Equivalently, given generators $G_0, G_1 \in \mathbb{F}_q^{k \times n}$, find $P \in S_n$ such that

$$G_1 = RREF(G_0 P).$$

The problem of deciding if two codes are equivalent is well-known in coding theory.

For our purpose, we are interested in the computational version: this is the vectorization problem for our action.

Permutation Equivalence Problem (PEP)

Given $\mathfrak{C}_0, \mathfrak{C}_1 \subseteq \mathbb{F}_q^n$, find a permutation π such that $\pi(\mathfrak{C}_0) = \mathfrak{C}_1$. Equivalently, given generators $G_0, G_1 \in \mathbb{F}_q^{k \times n}$, find $P \in S_n$ such that

$$G_1 = RREF(G_0 P).$$

Linear Equivalence Problem (LEP)

Given $\mathfrak{C}_0, \mathfrak{C}_1 \subseteq \mathbb{F}_q^n$, find a monomial μ such that $\mu(\mathfrak{C}_0) = \mathfrak{C}_1$.
Equivalently, given generators $G_0, G_1 \in \mathbb{F}_q^{k \times n}$, find $Q \in M_n(q)$ such that

$$G_1 = RREF(G_0 Q).$$

Code Equivalence Problems

The problem of deciding if two codes are equivalent is well-known in coding theory.

For our purpose, we are interested in the computational version: this is the vectorization problem for our action.

Permutation Equivalence Problem (PEP)

Given $\mathfrak{C}_0, \mathfrak{C}_1 \subseteq \mathbb{F}_q^n$, find a permutation π such that $\pi(\mathfrak{C}_0) = \mathfrak{C}_1$. Equivalently, given generators $G_0, G_1 \in \mathbb{F}_q^{k\times n}$, find $P \in S_n$ such that

$$G_1 = RREF(G_0 P).$$

Linear Equivalence Problem (LEP)

Given $\mathfrak{C}_0, \mathfrak{C}_1 \subseteq \mathbb{F}_q^n$, find a monomial μ such that $\mu(\mathfrak{C}_0) = \mathfrak{C}_1$.
Equivalently, given generators $G_0, G_1 \in \mathbb{F}_q^{k\times n}$, find $Q \in M_n(q)$ such that

$$G_1 = RREF(G_0 Q).$$

For practical applications, we are not interested in the semilinear version of the problem.

Roadmap

- Background
- Code-based Signatures
- Group Actions
- **LESS**
- Considerations

Could Code Equivalence be used as a stand-alone problem?

Applications in Cryptography

Could Code Equivalence be used as a stand-alone problem?

The situation for isometries recalls that of other group actions, such as for DLP (although without commutativity).

Applications in Cryptography

Could Code Equivalence be used as a stand-alone problem?

The situation for isometries recalls that of other group actions, such as for DLP (although without commutativity).

This means several existing constructions could be adapted to be based on Code Equivalence.

Applications in Cryptography

Could Code Equivalence be used as a stand-alone problem?

The situation for isometries recalls that of other group actions, such as for DLP (although without commutativity).

This means several existing constructions could be adapted to be based on Code Equivalence.

Possible to construct a ZK protocol based exclusively on the hardness of the code equivalence problem.

(Biasse, Micheli, P., Santini, 2020)

Could Code Equivalence be used as a stand-alone problem?

The situation for isometries recalls that of other group actions, such as for DLP (although without commutativity).

This means several existing constructions could be adapted to be based on Code Equivalence.

Possible to construct a ZK protocol based exclusively on the hardness of the code equivalence problem.
(Biasse, Micheli, P., Santini, 2020)

This can be then transformed into a full-fledged signature scheme via Fiat-Shamir.

Could Code Equivalence be used as a stand-alone problem?

The situation for isometries recalls that of other group actions, such as for DLP (although without commutativity).

This means several existing constructions could be adapted to be based on Code Equivalence.

Possible to construct a ZK protocol based exclusively on the hardness of the code equivalence problem.

(Biasse, Micheli, P., Santini, 2020)

This can be then transformed into a full-fledged signature scheme via Fiat-Shamir.

Protocol can be tweaked to increase efficiency (e.g. multiple public keys, fixed-weight challenges).

(Barenghi, Biasse, P., Santini, 2021)

Applications in Cryptography

Could Code Equivalence be used as a stand-alone problem?

The situation for isometries recalls that of other group actions, such as for DLP (although without commutativity).

This means several existing constructions could be adapted to be based on Code Equivalence.

Possible to construct a ZK protocol based exclusively on the hardness of the code equivalence problem.

(Biasse, Micheli, P., Santini, 2020)

This can be then transformed into a full-fledged signature scheme via Fiat-Shamir.

Protocol can be tweaked to increase efficiency (e.g. multiple public keys, fixed-weight challenges).

(Barenghi, Biasse, P., Santini, 2021)

Other applications (e.g. ring signatures) will not be discussed in this talk.

(Barenghi, Biasse, Ngo, P., Santini, 2022)

LESS ZK Identification Scheme

Public data: system params, hash function $Hash$, code $\mathfrak{C}$ with generator G_0.

LESS ZK Identification Scheme

Public data: system params, hash function $Hash$, code $\mathfrak{C}$ with generator G_0.

Key Generation

- SK: monomial matrix Q.
- PK: matrix $G_1 = RREF(G_0 Q)$.

LESS ZK Identification Scheme

Public data: system params, hash function $Hash$, code $\mathfrak{C}$ with generator G_0.

Key Generation

- SK: monomial matrix Q.
- PK: matrix $G_1 = RREF(G_0 Q)$.

Commit

- Choose random monomial matrix $\tilde{Q} \in M_n(q)$.
- Compute $\tilde{G} = RREF(G_0 \tilde{Q})$
- Commit to $cmt = Hash(\tilde{G})$.

LESS ZK Identification Scheme

Public data: system params, hash function $Hash$, code $\mathfrak{C}$ with generator G_0.

Key Generation

- SK: monomial matrix Q.
- PK: matrix $G_1 = RREF(G_0 Q)$.

Commit

- Choose random monomial matrix $\tilde{Q} \in M_n(q)$.
- Compute $\tilde{G} = RREF(G_0 \tilde{Q})$
- Commit to $cmt = Hash(\tilde{G})$.

Challenge

- Choose random bit $ch \in \{0, 1\}$.

Public data: system params, hash function $Hash$, code $\mathfrak{C}$ with generator G_0.

Key Generation

- SK: monomial matrix Q.
- PK: matrix $G_1 = RREF(G_0 Q)$.

Commit

- Choose random monomial matrix $\tilde{Q} \in M_n(q)$.
- Compute $\tilde{G} = RREF(G_0 \tilde{Q})$
- Commit to $cmt = Hash(\tilde{G})$.

Challenge

- Choose random bit $ch \in \{0, 1\}$.

Response

- If $ch = 0$ respond with $rsp = \tilde{Q}$.
- If $ch = 1$ respond with $rsp = Q^{-1}\tilde{Q}$.

LESS ZK Identification Scheme

Public data: system params, hash function $Hash$, code $\mathfrak{C}$ with generator G_0.

Key Generation

- SK: monomial matrix Q.
- PK: matrix $G_1 = RREF(G_0Q)$.

Commit

- Choose random monomial matrix $\tilde{Q} \in M_n(q)$.
- Compute $\tilde{G} = RREF(G_0\tilde{Q})$
- Commit to $cmt = Hash(\tilde{G})$.

Challenge

- Choose random bit $ch \in \{0, 1\}$.

Response

- If $ch = 0$ respond with $rsp = \tilde{Q}$.
- If $ch = 1$ respond with $rsp = Q^{-1}\tilde{Q}$.

Verify

- If $ch = 0$ verify that $Hash(RREF(G_0 \cdot rsp)) = cmt$.
- If $ch = 1$ verify that $Hash(RREF(G_1 \cdot rsp)) = cmt$.

It is easy to prove that the ZK protocol is complete, 2-special sound and honest-verifier zero-knowledge.

It is easy to prove that the ZK protocol is complete, 2-special sound and honest-verifier zero-knowledge.

Before applying Fiat-Shamir, need to reduce soundness error $\to t = \lambda$ parallel repetitions.

It is easy to prove that the ZK protocol is complete, 2-special sound and honest-verifier zero-knowledge.

Before applying Fiat-Shamir, need to reduce soundness error $\rightarrow t = \lambda$ parallel repetitions.

The protocol can be greatly improved with the following modifications:

It is easy to prove that the ZK protocol is complete, 2-special sound and honest-verifier zero-knowledge.

Before applying Fiat-Shamir, need to reduce soundness error $\rightarrow t = \lambda$ parallel repetitions.

The protocol can be greatly improved with the following modifications:

- Use non-binary challenges.

It is easy to prove that the ZK protocol is complete, 2-special sound and honest-verifier zero-knowledge.

Before applying Fiat-Shamir, need to reduce soundness error $\rightarrow t = \lambda$ parallel repetitions.

The protocol can be greatly improved with the following modifications:

- Use non-binary challenges.

+ Lower soundness error: $1/2 \rightarrow 1/2^\ell$.

– Rapid increase in public key size.

It is easy to prove that the ZK protocol is complete, 2-special sound and honest-verifier zero-knowledge.

Before applying Fiat-Shamir, need to reduce soundness error $\rightarrow t = \lambda$ parallel repetitions.

The protocol can be greatly improved with the following modifications:

- Use non-binary challenges.

+ Lower soundness error: $1/2 \rightarrow 1/2^\ell$.
− Rapid increase in public key size.

- Use a fixed-weight challenge string.

It is easy to prove that the ZK protocol is complete, 2-special sound and honest-verifier zero-knowledge.

Before applying Fiat-Shamir, need to reduce soundness error $\rightarrow t = \lambda$ parallel repetitions.

The protocol can be greatly improved with the following modifications:

- Use non-binary challenges.

+ Lower soundness error: $1/2 \rightarrow 1/2^\ell$.
– Rapid increase in public key size.

- Use a fixed-weight challenge string.

+ Exploits imbalance in cost of response: seed vs monomial.
– Larger number of iterations.

It is easy to prove that the ZK protocol is complete, 2-special sound and honest-verifier zero-knowledge.

Before applying Fiat-Shamir, need to reduce soundness error $\rightarrow t = \lambda$ parallel repetitions.

The protocol can be greatly improved with the following modifications:

- Use non-binary challenges.

+ Lower soundness error: $1/2 \rightarrow 1/2^{\ell}$.
– Rapid increase in public key size.

- Use a fixed-weight challenge string.

+ Exploits imbalance in cost of response: seed vs monomial.
– Larger number of iterations.

Both modifications do not affect security, only require small tweaks in proofs.

Input: system params, code $\mathfrak{C}$ with generator G_0.

Key Generation

Input: system params, code $\mathfrak{C}$ with generator G_0.

Key Generation

1. Set $SK_0 = I_n$ and $PK_0 = G_0$.
2. Choose random seed $seed_{sk} \in \{0,1\}^\lambda$.
3. Generate $Q_1, \ldots, Q_{s-1}$ from $seed_{sk}$.
4. for $i := 1$ to $s-1$
5. Set $SK(i) = Q_i$ and $PK(i) = RREF(G_0 Q_i)$.
6. Output $SK = (SK_0, \ldots, SK_{s-1})$ and $PK = (PK_0, \ldots, PK_{s-1})$.

Key Generation

Input: system params, code $\mathfrak{C}$ with generator G_0.

Key Generation

1. Set $SK_0 = I_n$ and $PK_0 = G_0$.
2. Choose random seed $seed_{sk} \in \{0,1\}^\lambda$.
3. Generate $Q_1, \dots, Q_{s-1}$ from $seed_{sk}$.
4. for $i := 1$ to $s-1$
5. Set $SK(i) = Q_i$ and $PK(i) = RREF(G_0 Q_i)$.
6. Output $SK = (SK_0, \dots, SK_{s-1})$ and $PK = (PK_0, \dots, PK_{s-1})$.

Private key can be easily compressed to a single seed.

Input: system params, hash function $Hash$, private key SK, message msg.

Input: system params, hash function $Hash$, private key SK, message msg.

Sign

1. Choose random master seed $mseed \in \{0, 1\}^\lambda$.
2. Generate $seed_0, \ldots, seed_{t-1}$ from $mseed$.
3. for $i := 1$ to $t - 1$
4. Generate $\tilde{Q}_i$ from $seed_i$.
5. Compute $\tilde{G}_i = RREF(G_0\tilde{Q}_i)$.
6. Set $d = Hash(\tilde{G}_0 || \ldots || \tilde{G}_{t-1} || msg)$.
7. Expand d to string $(x_0, \ldots, x_{t-1})$ with ω non-zero elements from $[0; s-1]$.
8. for $i := 0$ to $t - 1$
9. Set rsp_i to either $seed_i$ (if $x_i = 0$) or $Q_{x_i}^{-1}\tilde{Q}_i$ (otherwise).
10. Output $\sigma = (rsp_0, \ldots, rsp_{t-1}, d)$.

Input: system params, hash function $Hash$, private key SK, message msg.

Sign

1. Choose random master seed $mseed \in \{0,1\}^\lambda$.
2. Generate $seed_0, \dots, seed_{t-1}$ from $mseed$.
3. for $i := 1$ to $t-1$
4. Generate $\tilde{Q}_i$ from $seed_i$.
5. Compute $\tilde{G}_i = RREF(G_0 \tilde{Q}_i)$.
6. Set $d = Hash(\tilde{G}_0 || \dots || \tilde{G}_{t-1} || msg)$.
7. Expand d to string $(x_0, \dots, x_{t-1})$ with ω non-zero elements from $[0; s-1]$.
8. for $i := 0$ to $t-1$
9. Set rsp_i to either $seed_i$ (if $x_i = 0$) or $Q_{x_i}^{-1}\tilde{Q}_i$ (otherwise).
10. Output $\sigma = (rsp_0, \dots, rsp_{t-1}, d)$.

The expand function (7.) is obtained via application of a PRNG, sampling uniformly at random from the target set.

Verify

Input: system params, hash function $Hash$, public key PK, message msg, signature $sigma$.

Input: system params, hash function $Hash$, public key PK, message msg, signature $sigma$.

Verify

1. Expand d to string $(x_0, \ldots, x_{t-1})$ with ω non-zero elements from $[0; s-1]$.
2. for $i := 1$ to $t-1$
3. Recover $\overline{Q}_i$ from rsp_i.
4. Compute $\overline{G}_i = RREF(G_{x_i}\overline{Q}_i)$.
5. Set $d' = Hash(\overline{G}_0 || \ldots || \overline{G}_{t-1} || msg)$.
6. Output $true$ if $d = d'$, or $false$ otherwise.

Input: system params, hash function $Hash$, public key PK, message msg, signature $sigma$.

Verify

1. Expand d to string $(x_0, \ldots, x_{t-1})$ with ω non-zero elements from $[0; s-1]$.
2. for $i := 1$ to $t-1$
3. Recover $\overline{Q}_i$ from rsp_i.
4. Compute $\overline{G}_i = RREF(G_{x_i}\overline{Q}_i)$.
5. Set $d' = Hash(\overline{G}_0 || \ldots || \overline{G}_{t-1} || msg)$.
6. Output $true$ if $d = d'$, or $false$ otherwise.

The recover function (3.) compactly describes: rsp is either already a monomial, or a matrix can be obtained expanding a seed.

Roadmap

- Background
- Code-based Signatures
- Group Actions
- LESS
- Considerations

PEP is not NP-complete, unless the polynomial hierarchy collapses.

(Petrank, Roth, 1997)

PEP is not NP-complete, unless the polynomial hierarchy collapses.

(Petrank, Roth, 1997)

PEP is also deeply connected with Graph Isomorphism (GI) (reductions in both ways!), solvable in quasi-polynomial time.

PEP is not NP-complete, unless the polynomial hierarchy collapses.

(Petrank, Roth, 1997)

PEP is also deeply connected with Graph Isomorphism (GI) (reductions in both ways!), solvable in quasi-polynomial time.

At the same time, PEP is “not necessarily easy”.

(Petrank, Roth, 1997)

Security Considerations

PEP is not NP-complete, unless the polynomial hierarchy collapses.

(Petrank, Roth, 1997)

PEP is also deeply connected with Graph Isomorphism (GI) (reductions in both ways!), solvable in quasi-polynomial time.

At the same time, PEP is “not necessarily easy”.

(Petrank, Roth, 1997)

PEP is a special case of LEP; indeed, with time $O(q)$, we have

$$PEP \xleftarrow{\text{Reduces to}} LEP$$

Security Considerations
5 Considerations

PEP is not NP-complete, unless the polynomial hierarchy collapses.
(Petrank, Roth, 1997)

PEP is also deeply connected with Graph Isomorphism (GI) (reductions in both ways!), solvable in quasi-polynomial time.

At the same time, PEP is “not necessarily easy”.
(Petrank, Roth, 1997)

PEP is a special case of LEP; indeed, with time $O(q)$, we have

$$PEP \xleftarrow{\text{Reduces to}} LEP$$

As a consequence, most solvers for PEP can be easily adapted to solve LEP as well.

Attack Strategy 1: Weak Instances

Exploit a variety of properties, give rise to (potentially) most efficient solvers.

Attack Strategy 1: Weak Instances

Exploit a variety of properties, give rise to (potentially) most efficient solvers.

- Support Splitting Algorithm (SSA) looks for invariants to distinguish equivalent codes.
  (Sendrier, 2000)

Exploit a variety of properties, give rise to (potentially) most efficient solvers.

- Support Splitting Algorithm (SSA) looks for invariants to distinguish equivalent codes.
  (Sendrier, 2000)

  Weight Enumerator Function (WEF) is one, but too expensive; compute on hull.

$$\mathcal{H}(\mathfrak{C}) = \mathfrak{C} \cap \mathfrak{C}^{\perp}$$

Attack Strategy 1: Weak Instances

5 Considerations

Exploit a variety of properties, give rise to (potentially) most efficient solvers.

- Support Splitting Algorithm (SSA) looks for invariants to distinguish equivalent codes.

  (Sendrier, 2000)

  Weight Enumerator Function (WEF) is one, but too expensive; compute on hull.

$$\mathcal{H}(\mathfrak{C}) = \mathfrak{C} \cap \mathfrak{C}^{\perp}$$

If $\mathfrak{C}_1 = \pi(\mathfrak{C}_0)$, then $\mathcal{H}(\mathfrak{C}_1) = \pi\big(\mathcal{H}(\mathfrak{C}_0)\big)$; running in $\mathcal{O}(q^h)$.

Attack Strategy 1: Weak Instances

Exploit a variety of properties, give rise to (potentially) most efficient solvers.

- Support Splitting Algorithm (SSA) looks for invariants to distinguish equivalent codes.

  (Sendrier, 2000)

  Weight Enumerator Function (WEF) is one, but too expensive; compute on hull.

  $$\mathcal{H}(\mathfrak{C}) = \mathfrak{C} \cap \mathfrak{C}^{\perp}$$

  If $\mathfrak{C}_1 = \pi(\mathfrak{C}_0)$, then $\mathcal{H}(\mathfrak{C}_1) = \pi\big(\mathcal{H}(\mathfrak{C}_0)\big)$; running in $\mathcal{O}(q^h)$.

  Random codes tend to have small hulls, which makes attack practical.

Attack Strategy 1: Weak Instances

Exploit a variety of properties, give rise to (potentially) most efficient solvers.

- Support Splitting Algorithm (SSA) looks for invariants to distinguish equivalent codes.

  (Sendrier, 2000)

  Weight Enumerator Function (WEF) is one, but too expensive; compute on hull.

  $$\mathcal{H}(\mathfrak{C}) = \mathfrak{C} \cap \mathfrak{C}^{\perp}$$

  If $\mathfrak{C}_1 = \pi(\mathfrak{C}_0)$, then $\mathcal{H}(\mathfrak{C}_1) = \pi\big(\mathcal{H}(\mathfrak{C}_0)\big)$; running in $\mathcal{O}(q^h)$.

  Random codes tend to have small hulls, which makes attack practical.

  * Use (weakly) self-dual codes to avoid attack.

Exploit a variety of properties, give rise to (potentially) most efficient solvers.

- Support Splitting Algorithm (SSA) looks for invariants to distinguish equivalent codes.
  (Sendrier, 2000)

  Weight Enumerator Function (WEF) is one, but too expensive; compute on hull.

  $$\mathcal{H}(\mathfrak{C}) = \mathfrak{C} \cap \mathfrak{C}^{\perp}$$

  If $\mathfrak{C}_1 = \pi(\mathfrak{C}_0)$, then $\mathcal{H}(\mathfrak{C}_1) = \pi\big(\mathcal{H}(\mathfrak{C}_0)\big)$; running in $\mathcal{O}(q^h)$.

  Random codes tend to have small hulls, which makes attack practical.

  * Use (weakly) self-dual codes to avoid attack.
  * To solve LEP, need to target closure of the code; these are always self-dual for $q \geq 5$.

Attack Strategy 1: Weak Instances

Exploit a variety of properties, give rise to (potentially) most efficient solvers.

- Support Splitting Algorithm (SSA) looks for invariants to distinguish equivalent codes. (Sendrier, 2000)

  Weight Enumerator Function (WEF) is one, but too expensive; compute on hull.

  $$\mathcal{H}(\mathfrak{C}) = \mathfrak{C} \cap \mathfrak{C}^{\perp}$$

  If $\mathfrak{C}_1 = \pi(\mathfrak{C}_0)$, then $\mathcal{H}(\mathfrak{C}_1) = \pi\big(\mathcal{H}(\mathfrak{C}_0)\big)$; running in $\mathcal{O}(q^h)$.

  Random codes tend to have small hulls, which makes attack practical.

  * Use (weakly) self-dual codes to avoid attack.
  * To solve LEP, need to target closure of the code; these are always self-dual for $q \geq 5$.

- Algebraic approaches of different nature, for example:

Attack Strategy 1: Weak Instances

Exploit a variety of properties, give rise to (potentially) most efficient solvers.

- Support Splitting Algorithm (SSA) looks for invariants to distinguish equivalent codes. (Sendrier, 2000)

  Weight Enumerator Function (WEF) is one, but too expensive; compute on hull.

  $$\mathcal{H}(\mathfrak{C}) = \mathfrak{C} \cap \mathfrak{C}^{\perp}$$

  If $\mathfrak{C}_1 = \pi(\mathfrak{C}_0)$, then $\mathcal{H}(\mathfrak{C}_1) = \pi\big(\mathcal{H}(\mathfrak{C}_0)\big)$; running in $\mathcal{O}(q^h)$.

  Random codes tend to have small hulls, which makes attack practical.
  - * Use (weakly) self-dual codes to avoid attack.
  - * To solve LEP, need to target closure of the code; these are always self-dual for $q \geq 5$.

- Algebraic approaches of different nature, for example:
  - * Set up a system of equations, solve via Gröbner basis. (Saeed-Taha, 2017)

Exploit a variety of properties, give rise to (potentially) most efficient solvers.

- Support Splitting Algorithm (SSA) looks for invariants to distinguish equivalent codes. (Sendrier, 2000)

  Weight Enumerator Function (WEF) is one, but too expensive; compute on hull.

  $$\mathcal{H}(\mathfrak{C}) = \mathfrak{C} \cap \mathfrak{C}^{\perp}$$

  If $\mathfrak{C}_1 = \pi(\mathfrak{C}_0)$, then $\mathcal{H}(\mathfrak{C}_1) = \pi\big(\mathcal{H}(\mathfrak{C}_0)\big)$; running in $\mathcal{O}(q^h)$.

  Random codes tend to have small hulls, which makes attack practical.
  - Use (weakly) self-dual codes to avoid attack.
  - To solve LEP, need to target closure of the code; these are always self-dual for $q \geq 5$.
- Algebraic approaches of different nature, for example:
  - Set up a system of equations, solve via Gröbner basis. (Saeed-Taha, 2017)
  - Exploit reduction to graph isomorphism. (Bardet et al., 2020)

Attack Strategy 1: Weak Instances

Exploit a variety of properties, give rise to (potentially) most efficient solvers.

- Support Splitting Algorithm (SSA) looks for invariants to distinguish equivalent codes. (Sendrier, 2000)

  Weight Enumerator Function (WEF) is one, but too expensive; compute on hull.

  $$\mathcal{H}(\mathfrak{C}) = \mathfrak{C} \cap \mathfrak{C}^{\perp}$$

  If $\mathfrak{C}_1 = \pi(\mathfrak{C}_0)$, then $\mathcal{H}(\mathfrak{C}_1) = \pi\big(\mathcal{H}(\mathfrak{C}_0)\big)$; running in $\mathcal{O}(q^h)$.

  Random codes tend to have small hulls, which makes attack practical.
  * Use (weakly) self-dual codes to avoid attack.
  * To solve LEP, need to target closure of the code; these are always self-dual for $q \geq 5$.

- Algebraic approaches of different nature, for example:
  * Set up a system of equations, solve via Gröbner basis. (Saeed-Taha, 2017)
  * Exploit reduction to graph isomorphism. (Bardet et al., 2020)

  These are only efficient (or applicable in the first place) if hull is trivial.

Action of π can be guessed from the set of all codewords with small weight w. (Leon, 1982)

Action of π can be guessed from the set of all codewords with small weight w. (Leon, 1982)

Moderate w guarantees no spurious solution and sufficiently low number of codewords.

Attack Strategy 2: Codeword Search

Action of π can be guessed from the set of all codewords with small weight w. (Leon, 1982)

Moderate w guarantees no spurious solution and sufficiently low number of codewords.

In practice, minimum distance plus 1 or 2 is enough to guarantee enough structure.

Attack Strategy 2: Codeword Search

Action of π can be guessed from the set of all codewords with small weight w. (Leon, 1982)

Moderate w guarantees no spurious solution and sufficiently low number of codewords.

In practice, minimum distance plus 1 or 2 is enough to guarantee enough structure.

The attack then consists of:

Attack Strategy 2: Codeword Search

Action of π can be guessed from the set of all codewords with small weight w. (Leon, 1982)

Moderate w guarantees no spurious solution and sufficiently low number of codewords.

In practice, minimum distance plus 1 or 2 is enough to guarantee enough structure.

The attack then consists of:

- Finding codewords (use ISD).

Attack Strategy 2: Codeword Search

Action of π can be guessed from the set of all codewords with small weight w. (Leon, 1982)

Moderate w guarantees no spurious solution and sufficiently low number of codewords.

In practice, minimum distance plus 1 or 2 is enough to guarantee enough structure.

The attack then consists of:

- Finding codewords (use ISD).
- Matching to extract permutation.

Attack Strategy 2: Codeword Search

Action of π can be guessed from the set of all codewords with small weight w. (Leon, 1982)

Moderate w guarantees no spurious solution and sufficiently low number of codewords.

In practice, minimum distance plus 1 or 2 is enough to guarantee enough structure.

The attack then consists of:

- Finding codewords (use ISD).
- Matching to extract permutation.

Cost is $\approx 2\log(N_w)C_{\mathsf{isd}}(n, k, q, w)$ + linear algebra.

Attack Strategy 2: Codeword Search

Action of π can be guessed from the set of all codewords with small weight w. (Leon, 1982)

Moderate w guarantees no spurious solution and sufficiently low number of codewords.

In practice, minimum distance plus 1 or 2 is enough to guarantee enough structure.

The attack then consists of:

- Finding codewords (use ISD).
- Matching to extract permutation.

Cost is $\approx 2\log(N_w)C_{\text{isd}}(n,k,q,w)$ + linear algebra.

Permutations preserve multiset of entries $\implies$ no need to find all words of weight w. (Beullens, 2020)

Attack Strategy 2: Codeword Search

Action of π can be guessed from the set of all codewords with small weight w. (Leon, 1982)

Moderate w guarantees no spurious solution and sufficiently low number of codewords.

In practice, minimum distance plus 1 or 2 is enough to guarantee enough structure.

The attack then consists of:

- Finding codewords (use ISD).
- Matching to extract permutation.

Cost is $\approx 2\log(N_w)C_{\text{isd}}(n, k, q, w)$ + linear algebra.

Permutations preserve multiset of entries $\implies$ no need to find all words of weight w. (Beullens, 2020)

Probabilistic algorithm, advantageous only if q is large.

Attack Strategy 2: Codeword Search

Action of π can be guessed from the set of all codewords with small weight w. (Leon, 1982)

Moderate w guarantees no spurious solution and sufficiently low number of codewords.

In practice, minimum distance plus 1 or 2 is enough to guarantee enough structure.

The attack then consists of:

- Finding codewords (use ISD).
- Matching to extract permutation.

Cost is $\approx 2\log(N_w)C_{\mathsf{isd}}(n,k,q,w)$ + linear algebra.

Permutations preserve multiset of entries $\implies$ no need to find all words of weight w. (Beullens, 2020)

Probabilistic algorithm, advantageous only if q is large.

Can obtain small improvement by carefully matching 2-dimensional subcodes instead. (Barenghi, Biasse, P., Santini, 2023)

Information-Set Decoding

An iterative procedure aimed at finding low-weight words. (Prange, 1962)

An iterative procedure aimed at finding low-weight words. (Prange, 1962)

In a nutshell: guess information set to reveal (error) positions.

An iterative procedure aimed at finding low-weight words. (Prange, 1962)

In a nutshell: guess information set to reveal (error) positions.

Several improvements over the years:

An iterative procedure aimed at finding low-weight words. (Prange, 1962)

In a nutshell: guess information set to reveal (error) positions.

Several improvements over the years:

- Carefully allocating positions (e.g. allow errors in IS).

An iterative procedure aimed at finding low-weight words. (Prange, 1962)

In a nutshell: guess information set to reveal (error) positions.

Several improvements over the years:

- Carefully allocating positions (e.g. allow errors in IS).
- Looking for collisions.

Information-Set Decoding

An iterative procedure aimed at finding low-weight words. (Prange, 1962)

In a nutshell: guess information set to reveal (error) positions.

Several improvements over the years:

- Carefully allocating positions (e.g. allow errors in IS).
- Looking for collisions.
- Using representations (e.g. $1 + 1 = 0$).

An iterative procedure aimed at finding low-weight words. (Prange, 1962)

In a nutshell: guess information set to reveal (error) positions.

Several improvements over the years:

- Carefully allocating positions (e.g. allow errors in IS).
- Looking for collisions.
- Using representations (e.g. $1 + 1 = 0$).
- Considering nearest neighbors.

An iterative procedure aimed at finding low-weight words. (Prange, 1962)

In a nutshell: guess information set to reveal (error) positions.

Several improvements over the years:

- Carefully allocating positions (e.g. allow errors in IS).
- Looking for collisions.
- Using representations (e.g. $1 + 1 = 0$).
- Considering nearest neighbors.
- ...

Information-Set Decoding

An iterative procedure aimed at finding low-weight words. (Prange, 1962)

In a nutshell: guess information set to reveal (error) positions.

Several improvements over the years:

- Carefully allocating positions (e.g. allow errors in IS).
- Looking for collisions.
- Using representations (e.g. $1 + 1 = 0$).
- Considering nearest neighbors.
- ...

Running time is $2^{\kappa w\left(1+o(1)\right)}$, where κ depends on rate R and w/n. (Canto Torres, Sendrier, 2016)

An iterative procedure aimed at finding low-weight words. (Prange, 1962)

In a nutshell: guess information set to reveal (error) positions.

Several improvements over the years:

- Carefully allocating positions (e.g. allow errors in IS).
- Looking for collisions.
- Using representations (e.g. $1 + 1 = 0$).
- Considering nearest neighbors.
- ...

Running time is $2^{\kappa w\left(1+o(1)\right)}$, where κ depends on rate R and w/n. (Canto Torres, Sendrier, 2016)

When $w = o(n)$, asymptotically κ is the the same for all algorithms:

$$\kappa = -log_2(1 - R)$$

An iterative procedure aimed at finding low-weight words. (Prange, 1962)

In a nutshell: guess information set to reveal (error) positions.

Several improvements over the years:

- Carefully allocating positions (e.g. allow errors in IS).
- Looking for collisions.
- Using representations (e.g. $1 + 1 = 0$).
- Considering nearest neighbors.
- $\ldots$

Running time is $2^{\kappa w \left(1+o(1)\right)}$, where κ depends on rate R and w/n. (Canto Torres, Sendrier, 2016)

When $w = o(n)$, asymptotically κ is the the same for all algorithms:

$$\kappa = -log_2(1 - R)$$

Improvements to Prange are only polynomial in n. They also come at a high memory cost.

An iterative procedure aimed at finding low-weight words. (Prange, 1962)

In a nutshell: guess information set to reveal (error) positions.

Several improvements over the years:

- Carefully allocating positions (e.g. allow errors in IS).
- Looking for collisions.
- Using representations (e.g. $1+1=0$).
- Considering nearest neighbors.
- $\dots$

Running time is $2^{\kappa w\left(1+o(1)\right)}$, where κ depends on rate R and w/n. (Canto Torres, Sendrier, 2016)

When $w=o(n)$, asymptotically κ is the the same for all algorithms:

$$\kappa = -log_2(1-R)$$

Improvements to Prange are only polynomial in n. They also come at a high memory cost.

Easy to adapt "early" variants to $\mathbb{F}_q$, $q \geq 3$, e.g. Stern's.(Peters, 2010)

Information-Set Decoding

5 Considerations

An iterative procedure aimed at finding low-weight words. (Prange, 1962)

In a nutshell: guess information set to reveal (error) positions.

Several improvements over the years:

- Carefully allocating positions (e.g. allow errors in IS).
- Looking for collisions.
- Using representations (e.g. $1 + 1 = 0$).
- Considering nearest neighbors.
- $\ldots$

Running time is $2^{\kappa w\left(1+o(1)\right)}$, where κ depends on rate R and w/n. (Canto Torres, Sendrier, 2016)

When $w = o(n)$, asymptotically κ is the the same for all algorithms:

$$\kappa = -log_2(1 - R)$$

Improvements to Prange are only polynomial in n. They also come at a high memory cost.

Easy to adapt "early" variants to $\mathbb{F}_q, q \geq 3$, e.g. Stern's.(Peters, 2010)

Gain from advanced techniques deteriorates quickly for increasing values of q.(Meurer, 2013)

We parametrize using latter type of attacks, following conservative criterion. Namely, we pick n, k, q so that, for any d and any w, we have:

$$\sqrt{N_d(w)} \cdot C_{\mathsf{ISD}}^{(d)}(n, k, q, w) > 2^{\lambda}.$$

We parametrize using latter type of attacks, following conservative criterion. Namely, we pick n, k, q so that, for any d and any w, we have:

$$\sqrt{N_d(w)} \cdot C_{\text{ISD}}^{(d)}(n, k, q, w) > 2^{\lambda}.$$

The design of LESS allows for high degree of flexibility and customizable features according to goal.

We parametrize using latter type of attacks, following conservative criterion. Namely, we pick n, k, q so that, for any d and any w, we have:

$$\sqrt{N_d(w)} \cdot C_{\text{ISD}}^{(d)}(n, k, q, w) > 2^{\lambda}.$$

The design of LESS allows for high degree of flexibility and customizable features according to goal.

We select two parameter sets per category level:

We parametrize using latter type of attacks, following conservative criterion. Namely, we pick n, k, q so that, for any d and any w, we have:

$$\sqrt{N_d(w)} \cdot C_{\mathsf{ISD}}^{(d)}(n, k, q, w) > 2^\lambda.$$

The design of LESS allows for high degree of flexibility and customizable features according to goal.

We select two parameter sets per category level:

- Balanced: yields similar sizes for PK and signature, e.g. minimizing their sum.

Design Considerations

We parametrize using latter type of attacks, following conservative criterion. Namely, we pick n, k, q so that, for any d and any w, we have:

$$\sqrt{N_d(w)} \cdot C_{\text{ISD}}^{(d)}(n, k, q, w) > 2^\lambda.$$

The design of LESS allows for high degree of flexibility and customizable features according to goal.

We select two parameter sets per category level:

- Balanced: yields similar sizes for PK and signature, e.g. minimizing their sum.
- Short: sacrifices PK size to push for smallest signature.

We parametrize using latter type of attacks, following conservative criterion. Namely, we pick n, k, q so that, for any d and any w, we have:

$$\sqrt{N_d(w)} \cdot C_{\text{ISD}}^{(d)}(n, k, q, w) > 2^{\lambda}.$$

The design of LESS allows for high degree of flexibility and customizable features according to goal.

We select two parameter sets per category level:

- Balanced: yields similar sizes for PK and signature, e.g. minimizing their sum.
- Short: sacrifices PK size to push for smallest signature.

We use SHAKE as our PRNG and SHA-3 for the collision-resistant hash function $Hash$.

We parametrize using latter type of attacks, following conservative criterion. Namely, we pick n, k, q so that, for any d and any w, we have:

$$\sqrt{N_d(w)} \cdot C_{\mathsf{ISD}}^{(d)}(n, k, q, w) > 2^{\lambda}.$$

The design of LESS allows for high degree of flexibility and customizable features according to goal.

We select two parameter sets per category level:

- Balanced: yields similar sizes for PK and signature, e.g. minimizing their sum.
- Short: sacrifices PK size to push for smallest signature.

We use SHAKE as our PRNG and SHA-3 for the collision-resistant hash function $Hash$.

We compactly generate and transmit seeds using a seed tree structure.

Sizes and Timings

Protocol parameters (t, ω, s) infer performance profile:

Protocol parameters (t, ω, s) infer performance profile:

					Seed Tree: Yes					Seed Tree: No				
NIST Cat.	Parameter Set	Code Params. n	k	q	Prot. Params. t	ω	s	PK (kB)	Sig (kB)	Prot. Params. t	ω	s	PK (kB)	Sig (kB)
1	Balanced	252	126	127	1053	18	2	13.7	6.1	247	30	2	13.7	10.8
	Short	252	126	127	1263	9	64	862.4	3.3	46	15	64	862.4	4.2
3	Balanced	468	234	31	1776	26	2	33.7	14.8	377	44	2	33.7	26.5
	Short	400	200	127	1297	14	64	2167.2	8	72	22	64	2167.2	10.3
5	Balanced	636	318	31	2518	34	2	62.1	27.5	525	57	2	62.1	49.7
	Short	506	253	509	2300	18	64	4447.9	14.6	116	28	64	4447.9	19.3

Protocol parameters (t, ω, s) infer performance profile:

					Seed Tree									
					Yes					No				
NIST Cat.	Parameter Set	Code Params. n	k	q	Prot. Params. t	ω	s	PK (kB)	Sig (kB)	Prot. Params. t	ω	s	PK (kB)	Sig (kB)
1	Balanced	252	126	127	1053	18	2	13.7	6.1	247	30	2	13.7	10.8
	Short	252	126	127	1263	9	64	862.4	3.3	46	15	64	862.4	4.2
3	Balanced	468	234	31	1776	26	2	33.7	14.8	377	44	2	33.7	26.5
	Short	400	200	127	1297	14	64	2167.2	8	72	22	64	2167.2	10.3
5	Balanced	636	318	31	2518	34	2	62.1	27.5	525	57	2	62.1	49.7
	Short	506	253	509	2300	18	64	4447.9	14.6	116	28	64	4447.9	19.3

Runtime is dominated by RREF computation, for both Keygen and Sign/Verify.

Protocol parameters (t, ω, s) infer performance profile:

					Seed Tree										
					Yes					No					
NIST Cat.	Parameter Set	Code Params. n	k	q	Prot. Params. t	ω	s	PK (kB)	Sig (kB)	Prot. Params. t	ω	s	PK (kB)	Sig (kB)
1	Balanced	252	126	127	1053	18	2	13.7	6.1	247	30	2	13.7	10.8
	Short	252	126	127	1263	9	64	862.4	3.3	46	15	64	862.4	4.2
3	Balanced	468	234	31	1776	26	2	33.7	14.8	377	44	2	33.7	26.5
	Short	400	200	127	1297	14	64	2167.2	8	72	22	64	2167.2	10.3
5	Balanced	636	318	31	2518	34	2	62.1	27.5	525	57	2	62.1	49.7
	Short	506	253	509	2300	18	64	4447.9	14.6	116	28	64	4447.9	19.3

Runtime is dominated by RREF computation, for both Keygen and Sign/Verify.

This yields timings with contrasting behavior. For our reference code:

Protocol parameters (t, ω, s) infer performance profile:

					Seed Tree									
					Yes					No				
NIST Cat.	Parameter Set	Code Params. n	k	q	Prot. Params. t	ω	s	PK (kB)	Sig (kB)	Prot. Params. t	ω	s	PK (kB)	Sig (kB)
1	Balanced	252	126	127	1053	18	2	13.7	6.1	247	30	2	13.7	10.8
	Short	252	126	127	1263	9	64	862.4	3.3	46	15	64	862.4	4.2
3	Balanced	468	234	31	1776	26	2	33.7	14.8	377	44	2	33.7	26.5
	Short	400	200	127	1297	14	64	2167.2	8	72	22	64	2167.2	10.3
5	Balanced	636	318	31	2518	34	2	62.1	27.5	525	57	2	62.1	49.7
	Short	506	253	509	2300	18	64	4447.9	14.6	116	28	64	4447.9	19.3

Runtime is dominated by RREF computation, for both Keygen and Sign/Verify.

This yields timings with contrasting behavior. For our reference code:

- Balanced, Cat. 1: Keygen $\approx$ 8 Mcycles, Sign/Verify $\approx$ 834 Mcycles

Sizes and Timings

5 Considerations

Protocol parameters (t, ω, s) infer performance profile:

					Seed Tree									
					Yes					No				
NIST Cat.	Parameter Set	Code Params.			Prot. Params.					Prot. Params.				
		n	k	q	t	ω	s	PK (kB)	Sig (kB)	t	ω	s	PK (kB)	Sig (kB)
1	Balanced	252	126	127	1053	18	2	13.7	6.1	247	30	2	13.7	10.8
	Short	252	126	127	1263	9	64	862.4	3.3	46	15	64	862.4	4.2
3	Balanced	468	234	31	1776	26	2	33.7	14.8	377	44	2	33.7	26.5
	Short	400	200	127	1297	14	64	2167.2	8	72	22	64	2167.2	10.3
5	Balanced	636	318	31	2518	34	2	62.1	27.5	525	57	2	62.1	49.7
	Short	506	253	509	2300	18	64	4447.9	14.6	116	28	64	4447.9	19.3

Runtime is dominated by RREF computation, for both Keygen and Sign/Verify.

This yields timings with contrasting behavior. For our reference code:

- Balanced, Cat. 1: Keygen $\approx$ 8 Mcycles, Sign/Verify $\approx$ 834 Mcycles
- Short, Cat. 1: Keygen $\approx$ 205 Mcycles, Sign/Verify $\approx$ 115 Mcycles

Performance Considerations

The flexibility of LESS allows multiple options for deployment.

[1] This is optimized code.

Performance Considerations

The flexibility of LESS allows multiple options for deployment.

For instance, can fit on a microcontroller (PK + Sig ≤ 20 kB) or push for ≈ 3 kB signature.

[1] This is optimized code.

The flexibility of LESS allows multiple options for deployment.

For instance, can fit on a microcontroller (PK + Sig ≤ 20 kB) or push for ≈ 3 kB signature.

Our balanced set is competitive with SPHINCS+. For Cat. 1:

[1] This is optimized code.

Performance Considerations

The flexibility of LESS allows multiple options for deployment.

For instance, can fit on a microcontroller (PK + Sig $\leq$ 20 kB) or push for $\approx$ 3 kB signature.

Our balanced set is competitive with SPHINCS+. For Cat. 1:

- Sizes: signature 7.8 kB ("small") or 17 kB ("fast"), public key very small.

[1]This is optimized code.

Performance Considerations

The flexibility of LESS allows multiple options for deployment.

For instance, can fit on a microcontroller (PK + Sig ≤ 20 kB) or push for ≈ 3 kB signature.

Our balanced set is competitive with SPHINCS+. For Cat. 1:

- Sizes: signature 7.8 kB ("small") or 17 kB ("fast"), public key very small.
- Timings: Keygen 9-1195 Mcycles, Sign 239-8995 Mcycles, Verify 4.7-28 Mcycles.

[1] This is optimized code.

The flexibility of LESS allows multiple options for deployment.

For instance, can fit on a microcontroller (PK + Sig $\leq$ 20 kB) or push for $\approx$ 3 kB signature.

Our balanced set is competitive with SPHINCS+. For Cat. 1:

- Sizes: signature 7.8 kB ("small") or 17 kB ("fast"), public key very small.
- Timings: Keygen 9-1195 Mcycles, Sign 239-8995 Mcycles, Verify 4.7-28 Mcycles.

Our short set compares well with e.g. Wave(let). For Cat. 1:

[1]This is optimized code.

Performance Considerations

5 Considerations

The flexibility of LESS allows multiple options for deployment.

For instance, can fit on a microcontroller (PK + Sig ≤ 20 kB) or push for ≈ 3 kB signature.

Our balanced set is competitive with SPHINCS+. For Cat. 1:

- Sizes: signature 7.8 kB ("small") or 17 kB ("fast"), public key very small.
- Timings: Keygen 9-1195 Mcycles, Sign 239-8995 Mcycles, Verify 4.7-28 Mcycles.

Our short set compares well with e.g. Wave(let). For Cat. 1:

- Sizes: signature ≈ 1 kB, public key ≈ 3.1 MB.

[1] This is optimized code.

The flexibility of LESS allows multiple options for deployment.

For instance, can fit on a microcontroller (PK + Sig $\leq$ 20 kB) or push for $\approx$ 3 kB signature.

Our balanced set is competitive with SPHINCS+. For Cat. 1:

- Sizes: signature 7.8 kB (“small”) or 17 kB (“fast”), public key very small.
- Timings: Keygen 9-1195 Mcycles, Sign 239-8995 Mcycles, Verify 4.7-28 Mcycles.

Our short set compares well with e.g. Wave(let). For Cat. 1:

- Sizes: signature $\approx$ 1 kB, public key $\approx$ 3.1 MB.
- Timings[1]: Keygen 7400 Mcycles, Sign 1644 Mcycles, Verify 5 Mcycles.

[1] This is optimized code.

The flexibility of LESS allows multiple options for deployment.

For instance, can fit on a microcontroller (PK + Sig $\leq$ 20 kB) or push for $\approx$ 3 kB signature.

Our balanced set is competitive with SPHINCS+. For Cat. 1:

- Sizes: signature 7.8 kB (“small”) or 17 kB (“fast”), public key very small.
- Timings: Keygen 9-1195 Mcycles, Sign 239-8995 Mcycles, Verify 4.7-28 Mcycles.

Our short set compares well with e.g. Wave(let). For Cat. 1:

- Sizes: signature $\approx$ 1 kB, public key $\approx$ 3.1 MB.
- Timings[1]: Keygen 7400 Mcycles, Sign 1644 Mcycles, Verify 5 Mcycles.

There is ample room for improvement in our implementation:

[1] This is optimized code.

Performance Considerations

The flexibility of LESS allows multiple options for deployment.

For instance, can fit on a microcontroller (PK + Sig $\leq$ 20 kB) or push for $\approx$ 3 kB signature.

Our balanced set is competitive with SPHINCS+. For Cat. 1:

- Sizes: signature 7.8 kB ("small") or 17 kB ("fast"), public key very small.
- Timings: Keygen 9-1195 Mcycles, Sign 239-8995 Mcycles, Verify 4.7-28 Mcycles.

Our short set compares well with e.g. Wave(let). For Cat. 1:

- Sizes: signature $\approx$ 1 kB, public key $\approx$ 3.1 MB.
- Timings[1]: Keygen 7400 Mcycles, Sign 1644 Mcycles, Verify 5 Mcycles.

There is ample room for improvement in our implementation:

- This week: about 5x speed-up for Cat. 1 parameters by tuning 64-bit arithmetic.

[1] This is optimized code.

The flexibility of LESS allows multiple options for deployment.

For instance, can fit on a microcontroller (PK + Sig $\leq$ 20 kB) or push for $\approx$ 3 kB signature.

Our balanced set is competitive with SPHINCS+. For Cat. 1:

- Sizes: signature 7.8 kB ("small") or 17 kB ("fast"), public key very small.
- Timings: Keygen 9-1195 Mcycles, Sign 239-8995 Mcycles, Verify 4.7-28 Mcycles.

Our short set compares well with e.g. Wave(let). For Cat. 1:

- Sizes: signature $\approx$ 1 kB, public key $\approx$ 3.1 MB.
- Timings[1]: Keygen 7400 Mcycles, Sign 1644 Mcycles, Verify 5 Mcycles.

There is ample room for improvement in our implementation:

- This week: about 5x speed-up for Cat. 1 parameters by tuning 64-bit arithmetic.
- Further gains exploiting e.g. vectorization.

[1]This is optimized code.

It is possible to further reduce signature size using a couple of additional techniques:

It is possible to further reduce signature size using a couple of additional techniques:

- Moving from monomials to permutations.

Additional Optimizations

It is possible to further reduce signature size using a couple of additional techniques:

- Moving from monomials to permutations.

  This requires a few small design modifications (e.g. using self-dual codes) and will be integrated for the final submission (June).

Additional Optimizations

It is possible to further reduce signature size using a couple of additional techniques:

- Moving from monomials to permutations.

  This requires a few small design modifications (e.g. using self-dual codes) and will be integrated for the final submission (June).

- Compact commitment and verification exploiting information sets.

Additional Optimizations

It is possible to further reduce signature size using a couple of additional techniques:

- Moving from monomials to permutations.

  This requires a few small design modifications (e.g. using self-dual codes) and will be integrated for the final submission (June).

- Compact commitment and verification exploiting information sets.

  Can transmit partial action and then reconstruct permutation/monomial.

Additional Optimizations

It is possible to further reduce signature size using a couple of additional techniques:

- Moving from monomials to permutations.

  This requires a few small design modifications (e.g. using self-dual codes) and will be integrated for the final submission (June).

- Compact commitment and verification exploiting information sets.

  Can transmit partial action and then reconstruct permutation/monomial.

  This variant is already considered in our document, but not yet implemented.

Additional Optimizations

It is possible to further reduce signature size using a couple of additional techniques:

- Moving from monomials to permutations.

  This requires a few small design modifications (e.g. using self-dual codes) and will be integrated for the final submission (June).

- Compact commitment and verification exploiting information sets.

  Can transmit partial action and then reconstruct permutation/monomial.

  This variant is already considered in our document, but not yet implemented.

Optimized implementations (e.g. ARM, possibly hardware) are also a target for June.

Thank you for listening!
Any questions?