DAGS: Key Encapsulation using Dyadic GS Codes


13 April 2018
Based on the hardness of decoding random linear codes (syndrome decoding problem).

Follows McEliece/Niederreiter framework.

Very efficient computation.

Natural implementation features thanks to binary vectors arithmetic.

Drawback: large keys (around 1 MByte).
Why Structured Codes

Try to tackle the large key issue.

Idea: public matrix with compact description.

Quasi-Cyclic Codes (as seen before).

**Quasi-Dyadic Codes** (Misoczki, Barreto ‘09).

Several code families have QD description:

If dyadic *signature* and code *support* verify certain conditions...

...then Dyadic $\cap$ Cauchy $\cap$ Goppa.
Alternant codes with non-trivial intersection with Goppa codes.

Admit parity-check which is superposition of $s$ blocks of size $t \times n$.

Each block $H_\ell$ has $ij$-th element $\frac{z_j}{(v_j - u_\ell)}$, (distinct) nonzero elements of $\mathbb{F}_{q^m}$.

If $t = 1$ this is a Goppa code.

Can generate QD-GS codes using (modified) algorithm for QD Goppa (P.'12).

Efficient decoder, similar performance, more flexibility.
Select hash functions $\mathcal{G}, \mathcal{H}, \mathcal{K}$. 
Select hash functions $G, H, K$.

**Key Generation**

- Generate a QD-GS code.
- SK: parity-check matrix $H$ in alternant form over $\mathbb{F}_{q^m}$.
- PK: generator matrix $G$ in systematic form over $\mathbb{F}_q$.

**Encapsulation**

Choose random word $m \in \mathbb{F}_{k^{'}}$.

Compute $(\rho \parallel \sigma) = G(m)$ and $d = H(m)$.

Generate error vector $e \in \mathbb{F}_n$ of weight $w$ from seed $\sigma$.

Output $(c, d)$ where $c = (\rho \parallel m) G + e$ and set $k = K(m)$.

**Decapsulation**

Recover codeword $((\rho \parallel m) \parallel e)$ from $\text{Decode}(c)$.

Recompute $G(m)$, $H(m)$ and $e'$, then compare.

Return $\perp$ if decoding fails or any check fails, else return $k = K(m')$. 

---

**Doardo Persichetti**

**Florida Atlantic University**

13 April 2018 5/11
DAGS: a QD-GS based KEM

Select hash functions $G, H, K$.

**Key Generation**
- Generate a QD-GS code.
- SK: parity-check matrix $H$ in alternant form over $\mathbb{F}_{q^m}$.
- PK: generator matrix $G$ in systematic form over $\mathbb{F}_q$.

**Encapsulation**
- Choose random word $m \in \mathbb{F}_{q^m}^{k'}$.
- Compute $(\rho \parallel \sigma) = G(m)$ and $d = H(m)$.
- Generate error vector $e \in \mathbb{F}_q^n$ of weight $w$ from seed $\sigma$.
- Output $(c, d)$ where $c = (\rho \parallel m)G + e$ and set $k = K(m)$. 
DAGS: a QD-GS based KEM

Select hash functions $G, H, K$.

### Key Generation
- Generate a QD-GS code.
- SK: parity-check matrix $H$ in alternant form over $\mathbb{F}_{q^m}$.
- PK: generator matrix $G$ in systematic form over $\mathbb{F}_q$.

### Encapsulation
- Choose random word $m \in \mathbb{F}_{q^k}'$.
- Compute $(\rho \parallel \sigma) = G(m)$ and $d = H(m)$.
- Generate error vector $e \in \mathbb{F}_{q^n}$ of weight $w$ from seed $\sigma$.
- Output $(c, d)$ where $c = (\rho \parallel m)G + e$ and set $k = K(m)$.

### Decapsulation
- Recover codeword $((\rho' \parallel m')$ and error $e')$ from $\text{Decode}(c)$.
- Recompute $G(m')$, $H(m')$ and $e''$, then compare.
- Return ⊥ if decoding fails or any check fails, else return $k = K(m')$. 
Uses McEliece framework and IND-CCA KEM transform (Hofheinz, Hövelmanns, Kiltz '17).

Length $k'$ of input $m$ is kept short, but long enough for 256 bits of entropy. This helps keeping $d$ small and making hashing more practical.

Private key (matrix $H$) can be efficiently represented by "support" ($v, y$). Alternant matrix is reconstructed on the fly together with syndrome computation. This results in a small private key without computational overhead.
Uses McEliece framework and IND-CCA KEM transform (Hofheinz, Hövelmanns, Kiltz ’17).

Leverages “randomized” IND-CPA McEliece variant for tighter security proof.
Uses McEliece framework and IND-CCA KEM transform (Hofheinz, Hövelmanns, Kiltz '17).

Leverages “randomized” IND-CPA McElieic variant for tighter security proof.

Length $k'$ of input $m$ is kept short, but long enough for 256 bits of entropy.
Uses McEliece framework and IND-CCA KEM transform (Hofheinz, Hövelmanns, Kiltz ’17).

Leverages “randomized” IND-CPA McEliece variant for tighter security proof.

Length $k'$ of input $m$ is kept short, but long enough for 256 bits of entropy.

This helps keeping $d$ small and making hashing more practical.
Uses McEliece framework and IND-CCA KEM transform (Hofheinz, Hövelmanns, Kiltz '17).

Leverages “randomized” IND-CPA McEliece variant for tighter security proof.

Length $k'$ of input $m$ is kept short, but long enough for 256 bits of entropy.

This helps keeping $d$ small and making hashing more practical.

Private key (matrix $H$) can be efficiently represented by “support” $(v, y)$.
Uses McEliece framework and IND-CCA KEM transform (Hofheinz, Hövelmanns, Kiltz '17).

Leverages “randomized” IND-CPA McEliece variant for tighter security proof.

Length $k'$ of input $m$ is kept short, but long enough for 256 bits of entropy.

This helps keeping $d$ small and making hashing more practical.

Private key (matrix $H$) can be efficiently represented by “support” $(v, y)$.

Alternant matrix is reconstructed on the fly together with syndrome computation.
Uses McEliece framework and IND-CCA KEM transform (Hofheinz, Hövelmanns, Kiltz ’17).

Leverages “randomized” IND-CPA McEliece variant for tighter security proof.

Length $k'$ of input $m$ is kept short, but long enough for 256 bits of entropy.

This helps keeping $d$ small and making hashing more practical.

Private key (matrix $H$) can be efficiently represented by “support” $(v, y)$.

Alternant matrix is reconstructed on the fly together with syndrome computation.

This results in a small private key without computational overhead.
There exist structural attacks targeting structured alternate codes: FOPT and variants (Faugère, Otmani, Perret, Tillich '10).

QC/QD structure crucial to reduce number of unknowns of system. No definitive complexity analysis available. Experimental evidence + (loose) theoretical bound = hardness scales with dimension of solution space (number of free variables). This is given by $m-1$ for QD Goppa, but it is $m^t-1$ for QD-GS codes. All QD Goppa parameters broken except for largest instances ($m=16$). No broken QD-GS parameters to date.
There exist structural attacks targeting structured alternant codes: FOPT and variants (Faugère, Otmani, Perret, Tillich ’10).

QC/QD structure crucial to reduce number of unknowns of system.
There exist structural attacks targeting structured alternant codes: FOPT and variants (Faugère, Otmani, Perret, Tillich ’10).

QC/QD structure crucial to reduce number of unknowns of system.

No definitive complexity analysis available.
There exist structural attacks targeting structured alternant codes: FOPT and variants (Faugère, Otmani, Perret, Tillich ’10).

QC/QD structure crucial to reduce number of unknowns of system.

No definitive complexity analysis available.

Experimental evidence + (loose) theoretical bound
There exist structural attacks targeting structured alternant codes: FOPT and variants (Faugère, Otmani, Perret, Tillich ’10).

QC/QD structure crucial to reduce number of unknowns of system.

No definitive complexity analysis available.

Experimental evidence + (loose) theoretical bound = hardness scales with *dimension of solution space* (number of free variables).
There exist structural attacks targeting structured alternant codes: FOPT and variants (Faugère, Otmani, Perret, Tillich ’10).

QC/QD structure crucial to reduce number of unknowns of system.

No definitive complexity analysis available.

Experimental evidence + (loose) theoretical bound
= hardness scales with dimension of solution space (number of free variables).

This is given by $m - 1$ for QD Goppa, but it is $mt - 1$ for QD-GS codes.
There exist structural attacks targeting structured alternant codes: FOPT and variants (Faugère, Otmani, Perret, Tillich ’10).

QC/QD structure crucial to reduce number of unknowns of system.

No definitive complexity analysis available.

Experimental evidence + (loose) theoretical bound = hardness scales with dimension of solution space (number of free variables).

This is given by $m - 1$ for QD Goppa, but it is $mt - 1$ for QD-GS codes.

All QD Goppa parameters broken except for largest instances ($m = 16$).
There exist structural attacks targeting structured alternant codes: FOPT and variants (Faugère, Otmani, Perret, Tillich ’10).

QC/QD structure crucial to reduce number of unknowns of system.

No definitive complexity analysis available.

Experimental evidence + (loose) theoretical bound = hardness scales with dimension of solution space (number of free variables).

This is given by $m - 1$ for QD Goppa, but it is $mt - 1$ for QD-GS codes.

All QD Goppa parameters broken except for largest instances ($m = 16$).

No broken QD-GS parameters to date.
We choose:
- Small $m$
- Large $s$ (power of 2)
- $t > 1$ odd
- Non-binary base field

\[ \mathbb{F}_{2^N} \text{ large enough to define code, without being huge (}N \leq 12 \). \]

Stay clear of algebraic attacks ($mt > 21$).

High error-correction capacity ($st/2$) $\rightarrow$ smaller codes.

Parameters (sizes in bytes):

<table>
<thead>
<tr>
<th>$q^m$</th>
<th>$n$</th>
<th>$k$</th>
<th>$k'$</th>
<th>$s$</th>
<th>$t$</th>
<th>$w$</th>
<th>$PK$</th>
<th>$SK$</th>
<th>$Ciphertext$</th>
<th>$Sec. Level$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{5}$</td>
<td>$2^{832}$</td>
<td>$416$</td>
<td>$2^{26}$</td>
<td>$2^{4}$</td>
<td>$13$</td>
<td>$10^{16}$</td>
<td>$6,760$</td>
<td>$2,496$</td>
<td>$552$</td>
<td>$2^{1}$</td>
</tr>
<tr>
<td>$2^{6}$</td>
<td>$2^{1216}$</td>
<td>$512$</td>
<td>$2^{43}$</td>
<td>$2^{5}$</td>
<td>$11$</td>
<td>$176$</td>
<td>$8,448$</td>
<td>$3,648$</td>
<td>$944$</td>
<td>$2^{3}$</td>
</tr>
<tr>
<td>$2^{6}$</td>
<td>$2^{2112}$</td>
<td>$704$</td>
<td>$2^{43}$</td>
<td>$2^{6}$</td>
<td>$11$</td>
<td>$352$</td>
<td>$11,616$</td>
<td>$6,336$</td>
<td>$1,616$</td>
<td>$2^{5}$</td>
</tr>
</tbody>
</table>
We choose:

- Small $m$
- Large $s$ (power of 2)
- $t > 1$ odd
- Non-binary base field

$\mathbb{F}_{q^m} = \mathbb{F}_{2^N}$ large enough to define code, without being huge ($N \leq 12$).
We choose:
- Small $m$
- Large $s$ (power of 2)
- $t > 1$ odd
- Non-binary base field

$\mathbb{F}_{q^m} = \mathbb{F}_{2^N}$ large enough to define code, without being huge ($N \leq 12$).

Stay clear of algebraic attacks ($mt > 21$).
We choose:
- Small $m$
- Large $s$ (power of 2)
- $t > 1$ odd
- Non-binary base field

$\mathbb{F}_{q^m} = \mathbb{F}_{2^N}$ large enough to define code, without being huge ($N \leq 12$).

Stay clear of algebraic attacks ($mt > 21$).

High error-correction capacity ($st/2$) $\rightarrow$ smaller codes.
**Performance**

We choose:
- Small $m$
- Large $s$ (power of 2)
- $t > 1$ odd
- Non-binary base field

$\mathbb{F}_{q^m} = \mathbb{F}_{2^N}$ large enough to define code, without being huge ($N \leq 12$).

Stay clear of algebraic attacks ($mt > 21$).

High error-correction capacity ($st/2$) $\rightarrow$ smaller codes.

Parameters (sizes in bytes):

<table>
<thead>
<tr>
<th>$q$</th>
<th>$m$</th>
<th>$n$</th>
<th>$k$</th>
<th>$k'$</th>
<th>$s$</th>
<th>$t$</th>
<th>$w$</th>
<th>PK</th>
<th>SK</th>
<th>Ciphertext</th>
<th>Sec. Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^5$</td>
<td>2</td>
<td>832</td>
<td>416</td>
<td>26</td>
<td>$2^4$</td>
<td>13</td>
<td>104</td>
<td>6,760</td>
<td>2,496</td>
<td>552</td>
<td>1</td>
</tr>
<tr>
<td>$2^6$</td>
<td>2</td>
<td>1216</td>
<td>512</td>
<td>43</td>
<td>$2^5$</td>
<td>11</td>
<td>176</td>
<td>8,448</td>
<td>3,648</td>
<td>944</td>
<td>3</td>
</tr>
<tr>
<td>$2^6$</td>
<td>2</td>
<td>2112</td>
<td>704</td>
<td>43</td>
<td>$2^6$</td>
<td>11</td>
<td>352</td>
<td>11,616</td>
<td>6,336</td>
<td>1,616</td>
<td>5</td>
</tr>
</tbody>
</table>
**ONGOING AND FUTURE WORK**

Simple reference implementation, designed for clarity.
Ongoing and Future Work

Simple reference implementation, designed for clarity.

Implementation is *isochronous*, to resist timing attacks and the like.
Simple reference implementation, designed for clarity.

Implementation is isochronous, to resist timing attacks and the like.

Much more efficient implementations are being developed:
  - Vectorized/Assembly/C++
  - Hardware (FPGA)
  - ...

Simple reference implementation, designed for clarity.
Implementation is isochronous, to resist timing attacks and the like.
Much more efficient implementations are being developed:
- Vectorized/Assembly/C++
- Hardware (FPGA)
- ...
Several optimizations from practice/theory are being investigated.
Ongoing and Future Work

Simple reference implementation, designed for clarity.

Implementation is isochronous, to resist timing attacks and the like.

Much more efficient implementations are being developed:

- Vectorized/Assembly/C++
- Hardware (FPGA)
- ...

Several optimizations from practice/theory are being investigated.

Work in progress to make implementation side-channel resistant.
Ongoing and Future Work

Simple reference implementation, designed for clarity.

Implementation is isochronous, to resist timing attacks and the like.

Much more efficient implementations are being developed:

- Vectorized/Assembly/C++
- Hardware (FPGA)
- ...

Several optimizations from practice/theory are being investigated.

Work in progress to make implementation side-channel resistant.

Accurate complexity analysis of algebraic attacks is ongoing/future project.
CONCLUSIONS

DAGS has many good features:

- Small sizes for all data (pk, sk, ciphertext): few Kb or less
- Many intertwined parameters → high flexibility and scalability
- Option for "binary DAGS" is being developed
- Alternant decoding presents no decryption failures → allows use of static keys
- Efficient in practice
  - Preliminary results in hardware show a speedup of up to 46x e.g. timing of 78,318 ns for DAGS

Entirely patent-free

Some delicate points:
- Caution required with structural attacks
  - Easy to avoid with appropriate choice of parameters
- Folding attacks don't perform well on large (non-binary) base field
- Non-binary arithmetic → more complex implementation
- Price to pay is actually fairly small
CONCLUSIONS

DAGS has many good features:

- Small sizes for all data (pk, sk, ciphertext): few Kb or less
Conclusions

DAGS has many good features:

- Small sizes for all data (pk, sk, ciphertext): few Kb or less
- Many intertwined parameters → high flexibility and scalability

Option for “binary DAGS” is being developed

Alternant decoding presents no decryption failures → allows use of static keys

Efficient in practice

Preliminary results in hardware show a speedup of up to 46x e.g. timing of 78,318 ns for DAGS

Entirely patent-free

Some delicate points:

Caution required with structural attacks

Easy to avoid with appropriate choice of parameters

Folding attacks don’t perform well on large (non-binary) base field

Non-binary arithmetic → more complex implementation

Price to pay is actually fairly small
DAGS has many good features:

- Small sizes for all data (pk, sk, ciphertext): few Kb or less
- Many intertwined parameters → high flexibility and scalability
  - Option for “binary DAGS” is being developed
CONCLUSIONS

DAGS has many good features:

- Small sizes for all data (pk, sk, ciphertext): few Kb or less
- Many intertwined parameters → high flexibility and scalability
  - Option for “binary DAGS” is being developed
- Alternant decoding presents no decryption failures → allows use of static keys
DAGS has many good features:

- Small sizes for all data (pk, sk, ciphertext): few Kb or less
- Many intertwined parameters → high flexibility and scalability
  - Option for “binary DAGS” is being developed
- Alternant decoding presents no decryption failures → allows use of static keys
- Efficient in practice

Preliminary results in hardware show a speedup of up to 46x e.g. timing of 78,318 ns for DAGS

Entirely patent-free

Some delicate points:
- Caution required with structural attacks
- Easy to avoid with appropriate choice of parameters
- Folding attacks don’t perform well on large (non-binary) base field

Non-binary arithmetic → more complex implementation

Price to pay is actually fairly small
CONCLUSIONS

DAGS has many good features:

- Small sizes for all data (pk, sk, ciphertext): few Kb or less
- Many intertwined parameters → high flexibility and scalability
  - Option for “binary DAGS” is being developed
- Alternant decoding presents no decryption failures → allows use of static keys
- Efficient in practice
  - Preliminary results in hardware show a speedup of up to 46x

Easy to avoid with appropriate choice of parameters

Some delicate points:

- Caution required with structural attacks
- Folding attacks don’t perform well on large (non-binary) base field

Non-binary arithmetic → more complex implementation

Price to pay is actually fairly small
CONCLUSIONS

DAGS has many good features:

- Small sizes for all data (pk, sk, ciphertext): few Kb or less
- Many intertwined parameters → high flexibility and scalability
  - Option for “binary DAGS” is being developed
- Alternant decoding presents no decryption failures → allows use of static keys
- Efficient in practice
  - Preliminary results in hardware show a speedup of up to 46x
  - e.g. timing of 78,318 ns for DAGS_3 Encapsulation
CONCLUSIONS

DAGS has many good features:

- Small sizes for all data (pk, sk, ciphertext): few Kb or less
- Many intertwined parameters  \( \rightarrow \) high flexibility and scalability
  - Option for “binary DAGS” is being developed
- Alternant decoding presents no decryption failures  \( \rightarrow \) allows use of static keys
- Efficient in practice
  - Preliminary results in hardware show a speedup of up to 46x
    - e.g. timing of 78,318 ns for DAGS\_3 Encapsulation
- Entirely patent-free
CONCLUSIONS

DAGS has many good features:

- Small sizes for all data (pk, sk, ciphertext): few Kb or less
- Many intertwined parameters $\rightarrow$ high flexibility and scalability
  - Option for “binary DAGS” is being developed
- Alternant decoding presents no decryption failures $\rightarrow$ allows use of static keys
- Efficient in practice
  - Preliminary results in hardware show a speedup of up to 46x
  - e.g. timing of 78,318 ns for DAGS_3 Encapsulation
- Entirely patent-free

Some delicate points:
CONCLUSIONS

DAGS has many good features:

- Small sizes for all data (pk, sk, ciphertext): few Kb or less
- Many intertwined parameters → high flexibility and scalability
  - Option for “binary DAGS” is being developed
- Alternant decoding presents no decryption failures → allows use of static keys
- Efficient in practice
  - Preliminary results in hardware show a speedup of up to 46x
  - e.g. timing of 78,318 ns for DAGS_3 Encapsulation
- Entirely patent-free

Some delicate points:

- Caution required with structural attacks
CONCLUSIONS

DAGS has many good features:

- Small sizes for all data (pk, sk, ciphertext): few Kb or less
- Many intertwined parameters → high flexibility and scalability
  - Option for “binary DAGS” is being developed
- Alternant decoding presents no decryption failures → allows use of static keys
- Efficient in practice
  - Preliminary results in hardware show a speedup of up to 46x
    - e.g. timing of 78,318 ns for DAGS_3 Encapsulation
- Entirely patent-free

Some delicate points:

- Caution required with structural attacks
  - Easy to avoid with appropriate choice of parameters
CONCLUSIONS

DAGS has many good features:

- Small sizes for all data (pk, sk, ciphertext): few Kb or less
- Many intertwined parameters → high flexibility and scalability
  - Option for “binary DAGS” is being developed
- Alternate decoding presents no decryption failures → allows use of static keys
- Efficient in practice
  - Preliminary results in hardware show a speedup of up to 46x
  - e.g. timing of 78,318 ns for DAGS_3 Encapsulation
- Entirely patent-free

Some delicate points:

- Caution required with structural attacks
  - Easy to avoid with appropriate choice of parameters
  - Folding attacks don’t perform well on large (non-binary) base field
CONCLUSIONS

DAGS has many good features:

- Small sizes for all data (pk, sk, ciphertext): few Kb or less
- Many intertwined parameters → high flexibility and scalability
  - Option for “binary DAGS” is being developed
- Alternant decoding presents no decryption failures → allows use of static keys
- Efficient in practice
  - Preliminary results in hardware show a speedup of up to 46x
  - e.g. timing of 78,318 ns for DAGS.3 Encapsulation
- Entirely patent-free

Some delicate points:

- Caution required with structural attacks
  - Easy to avoid with appropriate choice of parameters
  - Folding attacks don’t perform well on large (non-binary) base field
- Non-binary arithmetic → more complex implementation
CONCLUSIONS

DAGS has many good features:

- Small sizes for all data (pk, sk, ciphertext): few Kb or less
- Many intertwined parameters $\rightarrow$ high flexibility and scalability
  - Option for “binary DAGS” is being developed
- Alternant decoding presents no decryption failures $\rightarrow$ allows use of static keys
- Efficient in practice
  - Preliminary results in hardware show a speedup of up to 46x
    - e.g. timing of 78,318 ns for DAGS.3 Encapsulation
- Entirely patent-free

Some delicate points:

- Caution required with structural attacks
  - Easy to avoid with appropriate choice of parameters
  - Folding attacks don’t perform well on large (non-binary) base field
- Non-binary arithmetic $\rightarrow$ more complex implementation
  - Price to pay is actually fairly small
Thank you

www.dags-project.org