MicroEliece: McEliece for Embedded Devices

MicroEliece

T.Eisenbarth, T. Güneysu, S. Heyse, C. Paar

Horst Görtz Institute for IT-Security
Ruhr-University Bochum

CHES2009 Lausanne Switzerland

1. Intro
2. McEliece
3. Adaptions
4. Implementation
5. Results
6. Conclusion

1 Intro
- History
- Motivation

2 McEliece

3 Adaptions

4 Implementation

5 Results

6 Conclusion

History

- Proposed 1978 by Robert McEliece
- Makes use of linear error correcting code (originally Goppa Codes)
- Underlying problem (decoding of generic linear codes) is NP-hard [1]
- Up to now unbroken, but not well studied like RSA, ECC

History

- Proposed 1978 by Robert McEliece
- Makes use of linear error correcting code (originally Goppa Codes)
- Underlying problem (decoding of generic linear codes) is NP-hard [1]
- Up to now unbroken, but not well studied like RSA, ECC

History

- Proposed 1978 by Robert McEliece
- Makes use of linear error correcting code (originally Goppa Codes)
- Underlying problem (decoding of generic linear codes) is NP-hard [1]
- Up to now unbroken, but not well studied like RSA, ECC

History

- Proposed 1978 by Robert McEliece
- Makes use of linear error correcting code (originally Goppa Codes)
- Underlying problem (decoding of generic linear codes) is NP-hard [1]
- Up to now unbroken, but not well studied like RSA, ECC

Why not earlier?

- Memory requirements prevent implementation on μCs and FPGAs (450 KB for 80 bit security)
- But today off-the-shelf hardware contains sufficient memory

Why not earlier?

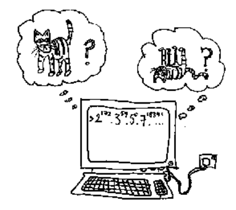
- Memory requirements prevent implementation on μCs and FPGAs (450 KB for 80 bit security)
- But today off-the-shelf hardware contains sufficient memory

Why not earlier?

- Memory requirements prevent implementation on μCs and FPGAs (450 KB for 80 bit security)
- But today off-the-shelf hardware contains sufficient memory

Why now?

- Except large keys, McEliece is very efficient
- Existence of quantum computers are a threat to systems based on the discrete log (DLP) and factorization (FP) problem
- Generally larger diversification for future public key systems is desirable

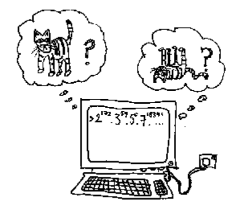


Why not earlier?

- Memory requirements prevent implementation on μCs and FPGAs (450 KB for 80 bit security)
- But today off-the-shelf hardware contains sufficient memory

Why now?

- Except large keys, McEliece is very efficient
- Existence of quantum computers are a threat to systems based on the discrete log (DLP) and factorization (FP) problem
- Generally larger diversification for future public key systems is desirable

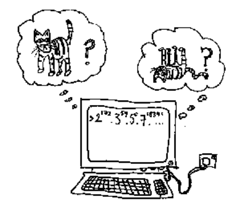


Why not earlier?

- Memory requirements prevent implementation on μCs and FPGAs (450 KB for 80 bit security)
- But today off-the-shelf hardware contains sufficient memory

Why now?

- Except large keys, McEliece is very efficient
- Existence of quantum computers are a threat to systems based on the discrete log (DLP) and factorization (FP) problem
- Generally larger diversification for future public key systems is desirable

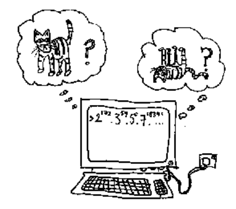


Why not earlier?

- Memory requirements prevent implementation on μCs and FPGAs (450 KB for 80 bit security)
- But today off-the-shelf hardware contains sufficient memory

Why now?

- Except large keys, McEliece is very efficient
- Existence of quantum computers are a threat to systems based on the discrete log (DLP) and factorization (FP) problem
- Generally larger diversification for future public key systems is desirable



1 Intro

2 McEliece
- Key Generation
- Encryption
- Decryption

3 Adaptions

4 Implementation

5 Results

6 Conclusion

Key Generation

- Randomly select a binary $(n \times k)$ generator matrix G of a code C capable of correcting t errors
- Select a random $(k \times k)$ binary non-singular scrambler matrix S
- Select a random $(n \times n)$ permutation matrix P
- Compute the $(k \times n)$ matrix $G_{pub} = S \times G \times P$
- **Public key is** (G_{pub}, t); **Private key is** (S, C, P).

Key Generation

- Randomly select a binary $(n \times k)$ generator matrix G of a code C capable of correcting t errors
- Select a random $(k \times k)$ binary non-singular scrambler matrix S
- Select a random $(n \times n)$ permutation matrix P
- Compute the $(k \times n)$ matrix $G_{pub} = S \times G \times P$
- **Public key is** (G_{pub}, t); **Private key is** (S, C, P).

Key Generation

- Randomly select a binary $(n \times k)$ generator matrix G of a code C capable of correcting t errors
- Select a random $(k \times k)$ binary non-singular scrambler matrix S
- Select a random $(n \times n)$ permutation matrix P
- Compute the $(k \times n)$ matrix $G_{pub} = S \times G \times P$
- **Public key is** (G_{pub}, t); **Private key is** (S, C, P).

Key Generation

- Randomly select a binary $(n \times k)$ generator matrix G of a code C capable of correcting t errors
- Select a random $(k \times k)$ binary non-singular scrambler matrix S
- Select a random $(n \times n)$ permutation matrix P
- Compute the $(k \times n)$ matrix $G_{pub} = S \times G \times P$
- **Public key is (G_{pub}, t); Private key is (S, C, P).**

Key Generation

- Randomly select a binary $(n \times k)$ generator matrix G of a code C capable of correcting t errors
- Select a random $(k \times k)$ binary non-singular scrambler matrix S
- Select a random $(n \times n)$ permutation matrix P
- Compute the $(k \times n)$ matrix $G_{pub} = S \times G \times P$
- **Public key is (G_{pub}, t); Private key is (S, C, P).**

Key Generation

- Randomly select a binary $(n \times k)$ generator matrix G of a code C capable of correcting t errors
- Select a random $(k \times k)$ binary non-singular scrambler matrix S
- Select a random $(n \times n)$ permutation matrix P
- Compute the $(k \times n)$ matrix $G_{pub} = S \times G \times P$
- **Public key is** (G_{pub}, t); **Private key is** (S, C, P).

Key Generation

- Randomly select a binary $(n \times k)$ generator matrix G of a code C capable of correcting t errors
- Select a random $(k \times k)$ binary non-singular scrambler matrix S
- Select a random $(n \times n)$ permutation matrix P
- Compute the $(k \times n)$ matrix $G_{pub} = S \times G \times P$
- **Public key is** (G_{pub}, t)**; Private key is** (S, C, P)**.**

In practice n determines the ciphertext size, k the plaintext size and t corresponds to the number of errors added.

Toy Example 1

For simplification (and size), a single error correcting $(7,4)$ Hamming code $\mathcal{H}$ is used.

$$G = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{pmatrix} S = \begin{pmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 \end{pmatrix} P = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

$$G_{pub} = S * G * P = \begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 \end{pmatrix}$$

Table: Security of McEliece Depending on Parameters

Security Level	**Parameters** (n, k, t)	**Size K_{pub}** in KBits	**Size K_{sec}** $(G(z), P, S)$ in KBits
(60 bit)	$(1024, 644, 38)$	644	$(0.38, 10, 405)$
(80 bit)	$(2048, 1751, 27)$	$3,502$	$(0.30, 22, 2994)$
(256 bit)	$(6624, 5129, 115)$	$33,178$	$(1.47, 104, 25690)$

Suggestion for fixed key sizes and the achieved security levels are made in [2].

Encryption

- Encode the message as a binary string m of length k
- Compute the vector $c' = m \times G_{pub}$ of length n
- Generate a random n-bit vector e containing at most t ones
- Compute the ciphertext as $c = c' + e$

Toy Example 2

$$
\begin{aligned}
m &= (1101) \\
c' &= m \times G_{pub} \\
&= (1101) \times \begin{Bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 \end{Bmatrix} \\
&= (1110010)
\end{aligned}
$$

$$
\begin{aligned}
c &= c' + e = (1110010) + (0000100) \\
&= (1110110)
\end{aligned}
$$

Encryption

- Encode the message as a binary string m of length k
- Compute the vector $c' = m \times G_{pub}$ of length n
- Generate a random n-bit vector e containing at most t ones
- Compute the ciphertext as $c = c' + e$

Toy Example 2

$$\begin{aligned} m &= (1101) \\ c' &= m \times G_{pub} \\ &= (1101) \times \begin{Bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 \end{Bmatrix} \\ &= (1110010) \end{aligned}$$

$$\begin{aligned} c &= c' + e = (1110010) + (0000100) \\ &= (1110110) \end{aligned}$$

Encryption

- Encode the message as a binary string m of length k
- Compute the vector $c' = m \times G_{pub}$ of length n
- Generate a random n-bit vector e containing at most t ones
- Compute the ciphertext as $c = c' + e$

Toy Example 2

$$
\begin{aligned}
m &= (1101) \\
c' &= m \times G_{pub} \\
&= (1101) \times \left\{\begin{matrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 \end{matrix}\right\} \\
&= (1110010)
\end{aligned}
$$

$$
\begin{aligned}
c &= c' + e = (1110010) + (0000100) \\
&= (1110110)
\end{aligned}
$$

Encryption

- Encode the message as a binary string m of length k
- Compute the vector $c' = m \times G_{pub}$ of length n
- Generate a random n-bit vector e containing at most t ones
- Compute the ciphertext as $c = c' + e$

Toy Example 2

$$
\begin{aligned}
m &= (1101) \\
c' &= m \times G_{pub} \\
&= (1101) \times \left\{ \begin{matrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 \end{matrix} \right\} \\
&= (1110010)
\end{aligned}
$$

$$
\begin{aligned}
c &= c' + e = (1110010) + (0000100) \\
&= (1110110)
\end{aligned}
$$

Encryption

- Encode the message as a binary string m of length k
- Compute the vector $c' = m \times G_{pub}$ of length n
- Generate a random n-bit vector e containing at most t ones
- Compute the ciphertext as $c = c' + e$

Toy Example 2

$$
\begin{aligned}
m &= (1101) \\
c' &= m \times G_{pub} \\
&= (1101) \times \begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 \end{pmatrix} \\
&= (1110010) \\
c &= c' + e = (1110010) + (0000100) \\
&= (1110110)
\end{aligned}
$$

Decryption

- Revert the permutation P => $\hat{c} = c \cdot P^{-1}$
- Use the decoding algorithm for the code C to decode $\hat{c}$ to $\hat{m}$
- Compute $m = \hat{m} \cdot S^{-1}$

Toy Example 3

$$\begin{aligned} c &= (0110110) \\ \hat{c} &= c \times P^{-1} = (1000111) \end{aligned}$$

Now use the secret information to efficiently decode $\hat{c}$ and correct the error. Here the error is at position seven.

$$\hat{c}_{corrected} = (1000110)$$

Because G is in systematic form, the first 4 bits are the message bits. By unscrambling with S^{-1} we can recover the original message.

$$\begin{aligned} \hat{c}_{corrected} &= (1000110) \\ \hat{m} &= (1000) \\ m &= \hat{m} \times S^{-1} = (1000) \times \begin{Bmatrix} 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \end{Bmatrix} \\ &= (1101) \end{aligned}$$

Decryption

- Revert the permutation P => $\hat{c} = c \cdot P^{-1}$
- Use the decoding algorithm for the code C to decode $\hat{c}$ to $\hat{m}$
- Compute $m = \hat{m} \cdot S^{-1}$

Toy Example 3

$$
\begin{aligned}
c &= (0110110) \\
\hat{c} &= c \times P^{-1} = (1000111)
\end{aligned}
$$

Now use the secret information to efficiently decode $\hat{c}$ and correct the error. Here the error is at position seven.

$$\hat{c}_{corrected} = (1000110)$$

Because G is in systematic form, the first 4 bits are the message bits. By unscrambling with S^{-1} we can recover the original message.

$$
\begin{aligned}
\hat{c}_{corrected} &= (1000110) \\
\hat{m} &= (1000) \\
m &= \hat{m} \times S^{-1} = (1000) \times \begin{Bmatrix} 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \end{Bmatrix} \\
&= (1101)
\end{aligned}
$$

Decryption

- Revert the permutation P => $\hat{c} = c \cdot P^{-1}$
- Use the decoding algorithm for the code C to decode $\hat{c}$ to $\hat{m}$
- Compute $m = \hat{m} \cdot S^{-1}$

Toy Example 3

$$\begin{aligned} c &= (0110110) \\ \hat{c} &= c \times P^{-1} = (1000111) \end{aligned}$$

Now use the secret information to efficiently decode $\hat{c}$ and correct the error. Here the error is at position seven.

$$\hat{c}_{corrected} = (1000110)$$

Because G is in systematic form, the first 4 bits are the message bits. By unscrambling with S^{-1} we can recover the original message.

$$\begin{aligned} \hat{c}_{corrected} &= (1000110) \\ \hat{m} &= (1000) \\ m &= \hat{m} \times S^{-1} = (1000) \times \left\{ \begin{array}{cccc} 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \end{array} \right\} \\ &= (1101) \end{aligned}$$

Toy Example 3

Decryption

- Revert the permutation P => $\hat{c} = c \cdot P^{-1}$
- Use the decoding algorithm for the code C to decode $\hat{c}$ to $\hat{m}$
- Compute $m = \hat{m} \cdot S^{-1}$

$$\begin{aligned} c &= (0110110) \\ \hat{c} &= c \times P^{-1} = (1000111) \end{aligned}$$

Now use the secret information to efficiently decode $\hat{c}$ and correct the error. Here the error is at position seven.

$$\hat{c}_{corrected} = (1000110)$$

Because G is in systematic form, the first 4 bits are the message bits. By unscrambling with S^{-1} we can recover the original message.

$$\begin{aligned} \hat{c}_{corrected} &= (1000110) \\ \hat{m} &= (1000) \\ m &= \hat{m} \times S^{-1} = (1000) \times \left\{ \begin{array}{cccc} 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \end{array} \right\} \\ &= (1101) \end{aligned}$$

Decoding Goppa Codes

- Syndrome computation
- Solve key equation
- Searching roots of a polynomial

Algorithm 1 Decoding Goppa Codes

Input: Received codeword r with up to t errors
Output: Recovered message $\hat{m}$

Compute syndrome $Syn(z)$ for codeword r
$T(z) \leftarrow Syn(z)^{-1}$
if $T(z) = z$ **then**
    $\sigma(z) \leftarrow z$
else
    $R(z) \leftarrow \sqrt{T(z) + z}$
    Compute $a(z)$ and $b(z)$ with $a(z) \equiv b(z) \cdot R(z) \mod G(z)$
    $\sigma(z) \leftarrow a(z)^2 + z \cdot b(z)^2$
end if
Determine roots of $\sigma(z)$, correct errors in r which results in $\hat{m}$
return $\hat{m}$

Decoding Goppa Codes

- Syndrome computation
- Solve key equation
  - two times polynomial EEA
  - polynomial square root
  - two polynomial squares
- Searching roots of a polynomial

Algorithm 2 Decoding Goppa Codes

Input: Received codeword r with up to t errors

Output: Recovered message $\hat{m}$

```
Compute syndrome Syn(z) for codeword r
T(z) ← Syn(z)^{-1}
if T(z) = z then
    σ(z) ← z
else
    R(z) ← √(T(z) + z)
    Compute a(z) and b(z) with a(z) ≡ b(z) · R(z) mod G(z)
    σ(z) ← a(z)^2 + z · b(z)^2
end if
Determine roots of σ(z), correct errors in r which results in m̂
return m̂
```

Decoding Goppa Codes

- Syndrome computation
- Solve key equation
  - two times polynomial EEA
  - polynomial square root
  - two polynomial squares
- Searching roots of a polynomial

Algorithm 3 Decoding Goppa Codes

Input: Received codeword r with up to t errors

Output: Recovered message $\hat{m}$

Compute syndrome $Syn(z)$ for codeword r
$T(z) \leftarrow Syn(z)^{-1}$
if $T(z) = z$ **then**
  $\sigma(z) \leftarrow z$
else
  $R(z) \leftarrow \sqrt{T(z) + z}$
  Compute $a(z)$ and $b(z)$ with $a(z) \equiv b(z) \cdot R(z) \mod G(z)$
  $\sigma(z) \leftarrow a(z)^2 + z \cdot b(z)^2$
end if
Determine roots of $\sigma(z)$, correct errors in r which results in $\hat{m}$
return $\hat{m}$

Decoding Goppa Codes

- Syndrome computation
- Solve key equation
  - two times polynomial EEA
  - polynomial square root
  - two polynomial squares
- Searching roots of a polynomial

Algorithm 4 Decoding Goppa Codes

Input: Received codeword r with up to t errors
Output: Recovered message $\hat{m}$

```
Compute syndrome Syn(z) for codeword r
T(z) ← Syn(z)^{-1}
if T(z) = z then
    σ(z) ← z
else
    R(z) ← sqrt(T(z) + z)
    Compute a(z) and b(z) with a(z) ≡ b(z) · R(z) mod G(z)
    σ(z) ← a(z)^2 + z · b(z)^2
end if
Determine roots of σ(z), correct errors in r which results in m̂
return m̂
```

Decoding Goppa Codes

- Syndrome computation
- Solve key equation
  - two times polynomial EEA
  - polynomial square root
  - two polynomial squares
- Searching roots of a polynomial

Algorithm 5 Decoding Goppa Codes

Input: Received codeword r with up to t errors

Output: Recovered message $\hat{m}$

1: Compute syndrome $Syn(z)$ for codeword r
2: $T(z) \leftarrow Syn(z)^{-1}$
3: **if** $T(z) = z$ **then**
4: $\quad \sigma(z) \leftarrow z$
5: **else**
6: $\quad R(z) \leftarrow \sqrt{T(z) + z}$
7: $\quad$ Compute $a(z)$ and $b(z)$ with $a(z) \equiv b(z) \cdot R(z) \mod G(z)$
8: $\quad \sigma(z) \leftarrow a(z)^2 + z \cdot b(z)^2$
9: **end if**
10: Determine roots of $\sigma(z)$, correct errors in r which results in $\hat{m}$
11: **return** $\hat{m}$

Decoding Goppa Codes

- Syndrome computation
- Solve key equation
  - two times polynomial EEA
  - polynomial square root
  - two polynomial squares
- Searching roots of a polynomial

Algorithm 6 Decoding Goppa Codes

Input: Received codeword r with up to t errors

Output: Recovered message $\hat{m}$

1: Compute syndrome $Syn(z)$ for codeword r
2: $T(z) \leftarrow Syn(z)^{-1}$
3: **if** $T(z) = z$ **then**
4: $\quad \sigma(z) \leftarrow z$
5: **else**
6: $\quad R(z) \leftarrow \sqrt{T(z) + z}$
7: $\quad$ Compute $a(z)$ and $b(z)$ with $a(z) \equiv b(z) \cdot R(z) \mod G(z)$
8: $\quad \sigma(z) \leftarrow a(z)^2 + z \cdot b(z)^2$
9: **end if**
10: Determine roots of $\sigma(z)$, correct errors in r which results in $\hat{m}$
11: **return** $\hat{m}$

1 Intro

2 McEliece

3 Adaptions
- Generation of the Parity Check Matrix
- Generation of the Scrambling Matrix

4 Implementation

5 Results

6 Conclusion

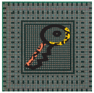
Our model

Typically one tries to reduce the public key size. We try to reduce secret key size. Why?

Our model

Typically one tries to reduce the public key size. We try to reduce secret key size. Why?

The large secret key must not be stored in an off-chip memory. It has to be kept in the internal flash of the μC and FPGA, respectively. Additional memory needed to speed up decryption.



Actually not the secret generator matrix is needed for decryption, but a corresponding parity check matrix H. H is a (2048×297) matrix $=$ 75 KByte. **How can we save space?**

Generation of the Parity Check Matrix H

- Very regular structure
- Only goppa polynomial and support required to compute H.
- Reverting the permutation P can be merged in.
- Instead 75 KByte only 3 KByte

Actually not the secret generator matrix is needed for decryption, but a corresponding parity check matrix H. H is a (2048×297) matrix $=$ 75 KByte. **How can we save space?**

Generation of the Parity Check Matrix H

- Very regular structure
- Only goppa polynomial and support required to compute H.
- Reverting the permutation P can be merged in.
- Instead 75 KByte only 3 KByte

Actually not the secret generator matrix is needed for decryption, but a corresponding parity check matrix H. H is a (2048×297) matrix $=$ 75 KByte. **How can we save space?**

Generation of the Parity Check Matrix H

- Very regular structure
- Only goppa polynomial and support required to compute H.
- Reverting the permutation P can be merged in.
- Instead 75 KByte only 3 KByte

Actually not the secret generator matrix is needed for decryption, but a corresponding parity check matrix H. H is a (2048×297) matrix $=$ 75 KByte. **How can we save space?**

Generation of the Parity Check Matrix H

- Very regular structure
- Only goppa polynomial and support required to compute H.
- Reverting the permutation P can be merged in.
- Instead 75 KByte only 3 KByte

Actually not the secret generator matrix is needed for decryption, but a corresponding parity check matrix H. H is a (2048×297) matrix $=$ 75 KByte. **How can we save space?**

Generation of the Parity Check Matrix H

- Very regular structure
- Only goppa polynomial and support required to compute H.
- Reverting the permutation P can be merged in.
- Instead 75 KByte only 3 KByte

Scrambling matrix S is a (1751×1751) matrix $= 347$ KByte. **How can we save space?**

Generation of the Scrambling Matrix

- Sole requirement for S is invertibility.
- About 33% of random matrices are invertible.
- Generate S^{-1} with a PRNG on-the-fly from a small seed.
- Assure invertibility during key generation.
- Instead 347 KByte only 80 bits (38.000 times smaller)

Scrambling matrix S is a (1751×1751) matrix $= 347$ KByte. **How can we save space?**

Generation of the Scrambling Matrix

- Sole requirement for S is invertibility.
- About 33% of random matrices are invertible.
- Generate S^{-1} with a PRNG on-the-fly from a small seed.
- Assure invertibility during key generation.
- Instead 347 KByte only 80 bits (38.000 times smaller)

Scrambling matrix S is a (1751×1751) matrix $= 347$ KByte. **How can we save space?**

Generation of the Scrambling Matrix

- Sole requirement for S is invertibility.
- About 33% of random matrices are invertible.
- Generate S^{-1} with a PRNG on-the-fly from a small seed.
- Assure invertibility during key generation.
- Instead 347 KByte only 80 bits (38.000 times smaller)

Scrambling matrix S is a (1751×1751) matrix $= 347$ KByte. **How can we save space?**

Generation of the Scrambling Matrix

- Sole requirement for S is invertibility.
- About 33% of random matrices are invertible.
- Generate S^{-1} with a PRNG on-the-fly from a small seed.
- Assure invertibility during key generation.
- Instead 347 KByte only 80 bits (38.000 times smaller)

Scrambling matrix S is a (1751×1751) matrix $= 347$ KByte. **How can we save space?**

Generation of the Scrambling Matrix

- Sole requirement for S is invertibility.
- About 33% of random matrices are invertible.
- Generate S^{-1} with a PRNG on-the-fly from a small seed.
- Assure invertibility during key generation.
- Instead 347 KByte only 80 bits (38.000 times smaller)

Scrambling matrix S is a (1751×1751) matrix $=$ 347 KByte. **How can we save space?**

Generation of the Scrambling Matrix

- Sole requirement for S is invertibility.
- About 33% of random matrices are invertible.
- Generate S^{-1} with a PRNG on-the-fly from a small seed.
- Assure invertibility during key generation.
- Instead 347 KByte only 80 bits (38.000 times smaller)

1 Intro

2 McEliece

3 Adaptions

4 Implementation
- Memory Requirements
- AVR
- FPGA

5 Results

6 Conclusion

For 80-bit security ($m = 11, n = 2048, k = 1751, t = 27$)

Table	**Size**
G_{pub} Matrix	428 KByte
Goppa Polynomial	308 bit
Support	22,528 bit
ω Polynomial	297 bit
logtable	22,528 bit
anti-log table	22,528 bit
S^{-1} Matrix	347 KByte, reduced to 80 bit
P^{-1} Matrix	only 2,75 Kbyte as array
iG Matrix	428 KByte, not needed when G in standard form

Table: Sizes of Stored Values

AVR Encryption

- Read in G_{pub} via UART or from external memory and store it to SRAM. Only once at system start-up!
- Multiply message m with G_{pub}.
- Distribute 27 errors.

AVR Encryption

- Read in G_{pub} via UART or from external memory and store it to SRAM. Only once at system start-up!
- Multiply message m with G_{pub}.
- Distribute 27 errors.

AVR Encryption

- Read in G_{pub} via UART or from external memory and store it to SRAM. Only once at system start-up!
- Multiply message m with G_{pub}.
- Distribute 27 errors.

AVR Decryption

- Compute Syndrome of ciphertext.
  - Run time computation. Slower (size of a second), but only (8 Kbyte) memory required
  - Use precomputed and pre-permuted values. Is fast, but large storage needed (108 Kbyte). Our choice
- Syndrome decoding. TLU based field arithmetic(2×4 KBytes).
- Searching roots. Very expensive (55.296 multiplications and adds).
- Revert substitution.
  - Use precomputed matrix. Reasonable fast, but too much memory needed (374 KByte).
  - Run time computation. Slower, but only 80 bits memory required. Stands or falls with speed of the PRNG.

AVR Decryption

- Compute Syndrome of ciphertext.
  - Run time computation. Slower (size of a second), but only (8 Kbyte) memory required
  - Use precomputed and pre-permuted values. Is fast, but large storage needed (108 Kbyte). Our choice
- Syndrome decoding. TLU based field arithmetic(2×4 KBytes).
- Searching roots. Very expensive (55.296 multiplications and adds).
- Revert substitution.
  - Use precomputed matrix. Reasonable fast, but too much memory needed (374 KByte).
  - Run time computation. Slower, but only 80 bits memory required. Stands or falls with speed of the PRNG.

AVR Decryption

- Compute Syndrome of ciphertext.
    - Run time computation. Slower (size of a second), but only (8 Kbyte) memory required
    - Use precomputed and pre-permuted values. Is fast, but large storage needed (108 Kbyte). Our choice
- Syndrome decoding. TLU based field arithmetic(2×4 KBytes).
- Searching roots. Very expensive (55.296 multiplications and adds).
- Revert substitution.
    - Use precomputed matrix. Reasonable fast, but too much memory needed (374 KByte).
    - Run time computation. Slower, but only 80 bits memory required. Stands or falls with speed of the PRNG.

AVR Decryption

- Compute Syndrome of ciphertext.
  - Run time computation. Slower (size of a second), but only (8 Kbyte) memory required
  - Use precomputed and pre-permuted values. Is fast, but large storage needed (108 Kbyte). Our choice
- Syndrome decoding. TLU based field arithmetic(2×4 KBytes).
- Searching roots. Very expensive (55.296 multiplications and adds).
- Revert substitution.
  - Use precomputed matrix. Reasonable fast, but too much memory needed (374 KByte)
  - Run time computation. Slower, but only 80 bits memory required Stands or falls with speed of the PRNG.

AVR Decryption

- Compute Syndrome of ciphertext.
  - Run time computation. Slower (size of a second), but only (8 Kbyte) memory required
  - Use precomputed and pre-permuted values. Is fast, but large storage needed (108 Kbyte). Our choice
- Syndrome decoding. TLU based field arithmetic(2×4 KBytes).
- Searching roots. Very expensive (55.296 multiplications and adds).
- Revert substitution.
  - Use precomputed matrix. Reasonable fast, but too much memory needed (374 KByte)
  - Run time computation. Slower, but only 80 bits memory required. Stands or falls with speed of the PRNG.

AVR Decryption

- Compute Syndrome of ciphertext.
  - Run time computation. Slower (size of a second), but only (8 Kbyte) memory required
  - Use precomputed and pre-permuted values. Is fast, but large storage needed (108 Kbyte). Our choice
- Syndrome decoding. TLU based field arithmetic(2×4 KBytes).
- Searching roots. Very expensive (55.296 multiplications and adds).
- Revert substitution.
  - Use precomputed matrix. Reasonable fast, but too much memory needed (374 KByte).
  - Run time computation. Slower, but only 80 bits memory required. Stands or falls with speed of the PRNG. Our choice

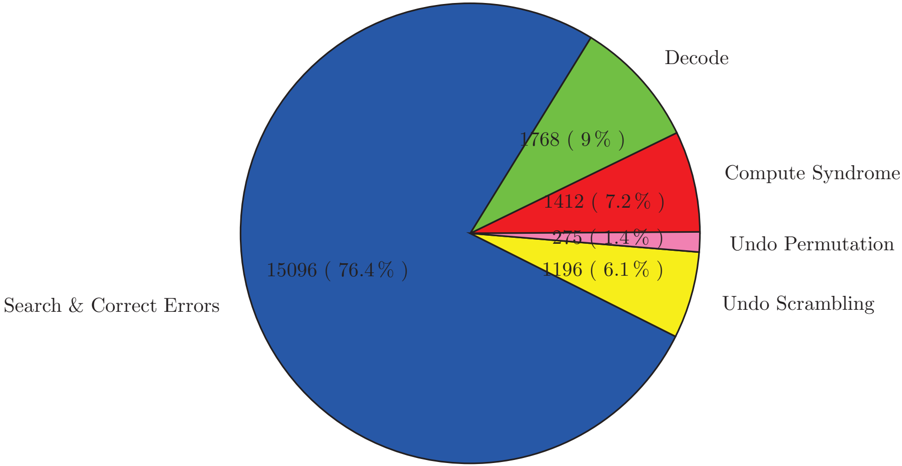
AVR Decryption

- Compute Syndrome of ciphertext.
  - Run time computation. Slower (size of a second), but only (8 Kbyte) memory required
  - Use precomputed and pre-permuted values. Is fast, but large storage needed (108 Kbyte). Our choice
- Syndrome decoding. TLU based field arithmetic(2×4 KBytes).
- Searching roots. Very expensive (55.296 multiplications and adds).
- Revert substitution.
  - Use precomputed matrix. Reasonable fast, but too much memory needed (374 KByte).
  - Run time computation. Slower, but only 80 bits memory required. Stands or falls with speed of the PRNG. Our choice

AVR Decryption

- Compute Syndrome of ciphertext.
  - Run time computation. Slower (size of a second), but only (8 Kbyte) memory required
  - Use precomputed and pre-permuted values. Is fast, but large storage needed (108 Kbyte). Our choice
- Syndrome decoding. TLU based field arithmetic(2×4 KBytes).
- Searching roots. Very expensive (55.296 multiplications and adds).
- Revert substitution.
  - Use precomputed matrix. Reasonable fast, but too much memory needed (374 KByte).
  - Run time computation. Slower, but only 80 bits memory required. Stands or falls with speed of the PRNG. Our choice

AVR-Decryption: Break Down of the Execution Time



Numbers shown are clock cycles x1000.

1 Intro

2 McEliece

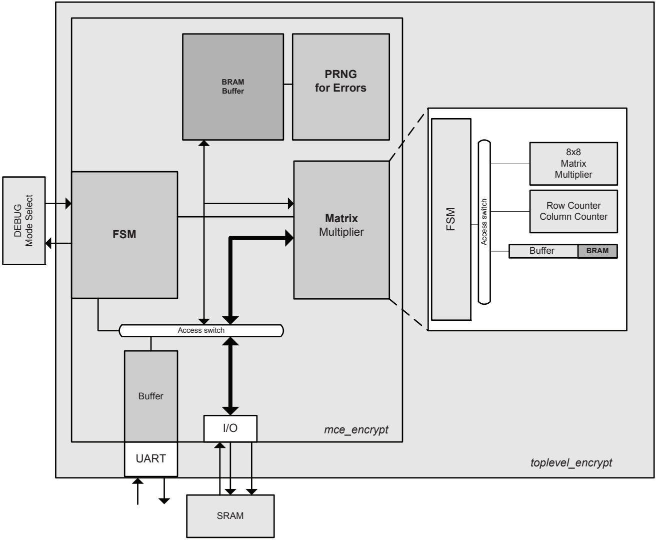
3 Adaptions

4 Implementation
- Memory Requirements
- AVR
- FPGA

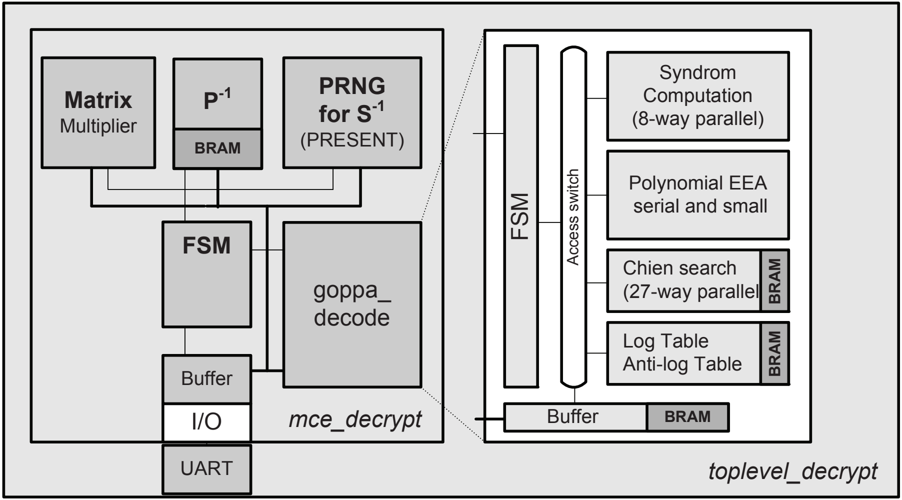
5 Results

6 Conclusion

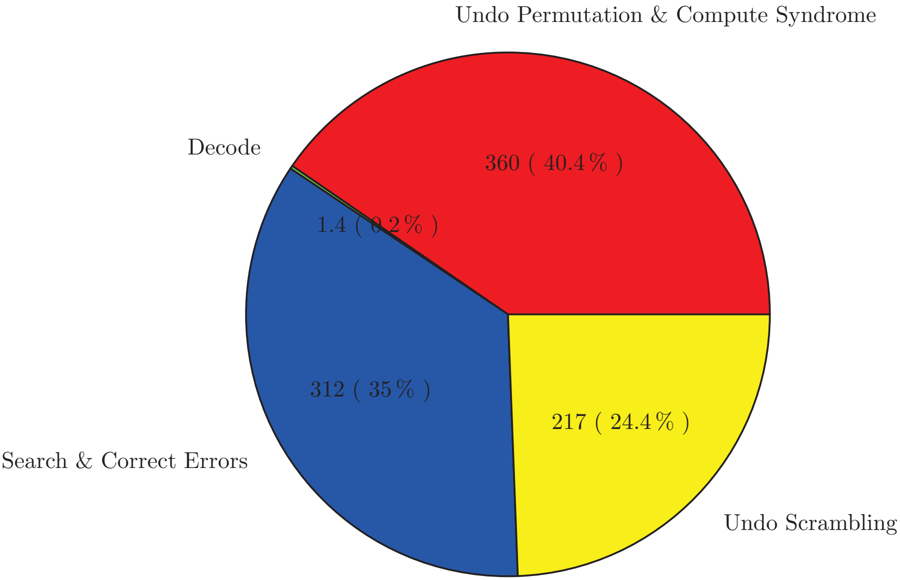
FPGA:Overview of the encryption circuit



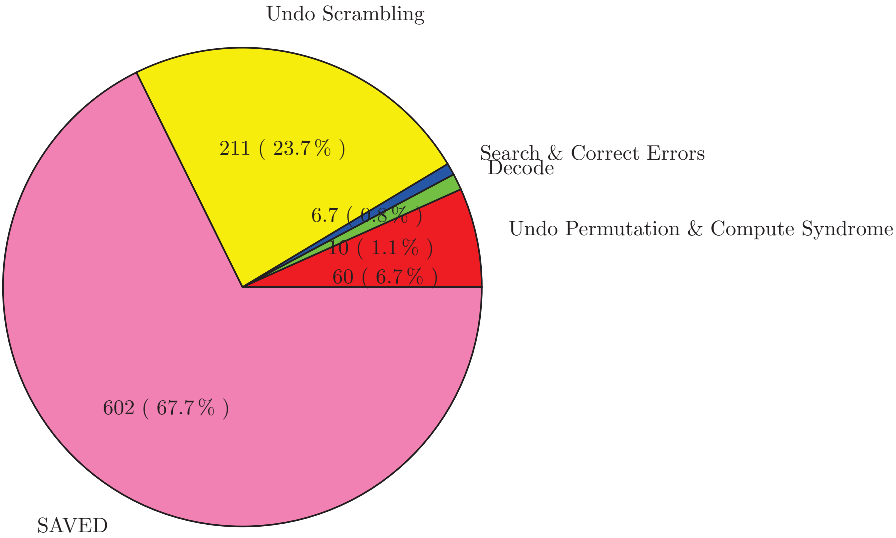
FPGA:Overview of the decryption circuit



FPGA-Decryption Old: Break Down of the Execution Time



FPGA-Decryption New: Break Down of the Execution Time



1 Intro

2 McEliece

3 Adaptions

4 Implementation

5 Results

6 Conclusion

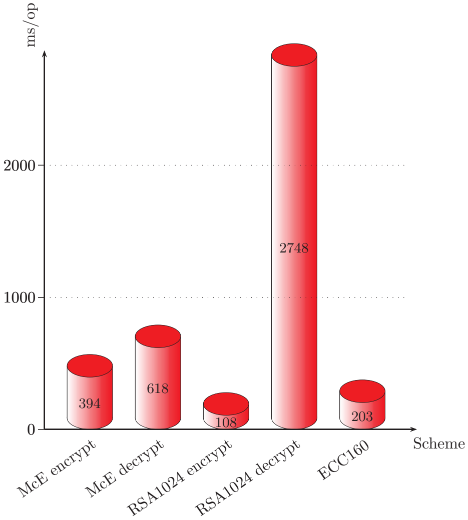
Table: Implementation results of the McEliece scheme with $n = 2048, k = 1751, t = 27$ on the AVR ATxMega192 μC and Spartan-3AN XC3S1400AN-5 FPGA after PAR.

	Resource	Encryption	Decryption	Available
μC	SRAM	512 Byte	12 kByte	16 kByte
	Flash Memory	684 Byte	130.4 kByte	192 kByte
	External Memory	438 kByte	–	–
FPGA	Slices	668 (6%)	9,400 (83%)	11,264
	LUTs	1044 (5%)	9,054 (40%)	22,528
	FFs	804 (4%)	12,870(57%)	22,528
	BRAMs	3 (9%)	32 (100%)	32

Table: Performance of McEliece implementations with $n = 2048$, $k = 1751$, $t = 27$ on the AVR ATxMega192 μC and Spartan-3AN XC3S1400AN-5 FPGA.

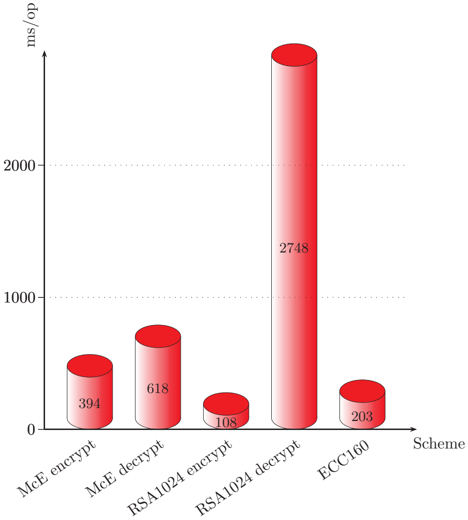
	Aspect	ATxMega192 μC	Spartan-3AN 1400
Encrypt.	Maximum frequency	32 MHz	150 MHz
	Encrypt $c' = m \cdot G_{pub}$	12,635,477 cycles	(7,889,200)161,480 cycles
	Inject errors $c = c' + z$	1,136 cycles	398 cycles
Decryption	Maximum frequency	32 MHz	110 MHz
	Undo permutation $c \cdot P^{-1}$	275,835 cycles	combined with $Syn(z)$
	Determine $Syn(z)$	1,412,514 cycles	69,116 cycles
	Compute $T = Syn(z)^{-1}$	1,164,402 cycles	4,346 cycles
	Compute $\sqrt{T+z}$	286,573 cycles	3,896 cycles
	Solve Key Equation	318,082 cycles	1,958 cycles
	Find & Correct errors	15,096,704 cycles	6,148 cycles
	Undo scrambling $\hat{m} \cdot S^{-1}$	1,196,984 cycles	217,800 cycles

AVR-Implementation for 80 bit security: Timings



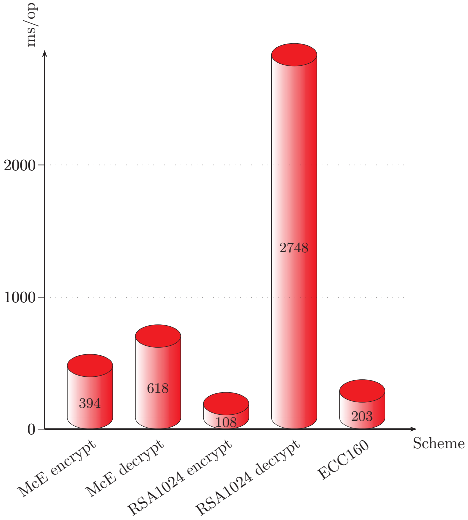
- AVR
  - Encryption 3.5 times slower than RSA and two times slower than ECC.
  - Decryption about 5.5 times slower than ECC, but five times faster than RSA.

AVR-Implementation for 80 bit security: Timings



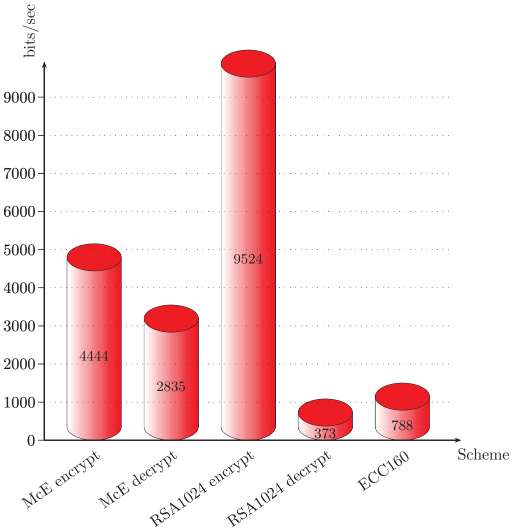
- AVR
  - Encryption 3.5 times slower than RSA and two times slower than ECC.
  - Decryption about 5.5 times slower than ECC, but five times faster than RSA.

AVR-Implementation for 80 bit security: Timings



- AVR
  - Encryption 3.5 times slower than RSA and two times slower than ECC.
  - Decryption about 5.5 times slower than ECC, but five times faster than RSA.

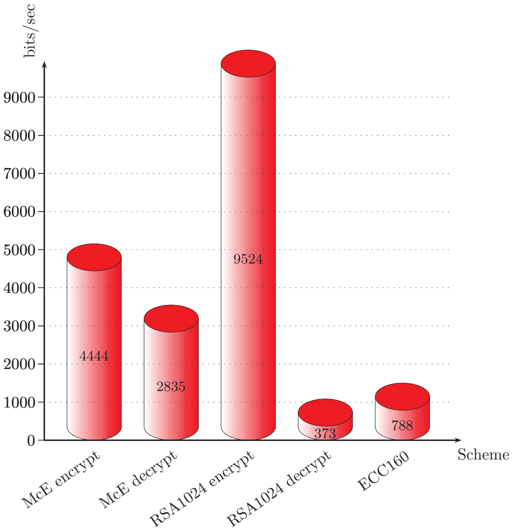
AVR-Implementation for 80 bit security: Throughput



When taking the throughput into account:

- AVR
  - Encryption over 25 times faster then ECC and only two times slower than RSA.
  - Decryption five times faster than ECC and eight times faster than RSA.

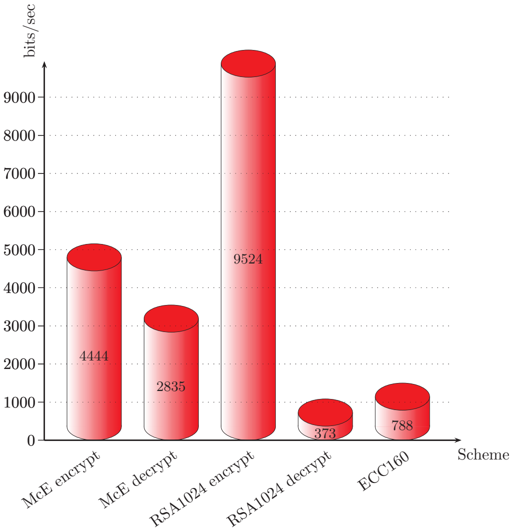
AVR-Implementation for 80 bit security: Throughput



When taking the throughput into account:

- AVR
  - Encryption over 25 times faster then ECC and only two times slower than RSA.
  - Decryption five times faster than ECC and eight times faster than RSA.

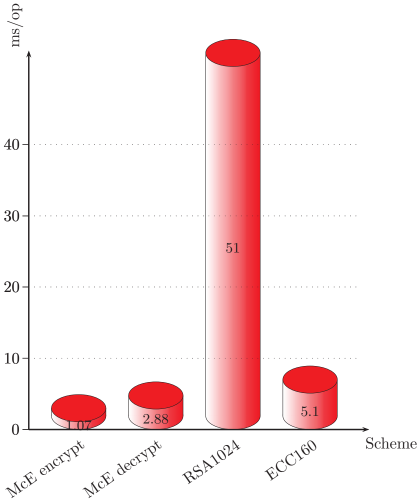
AVR-Implementation for 80 bit security: Throughput



When taking the throughput into account:

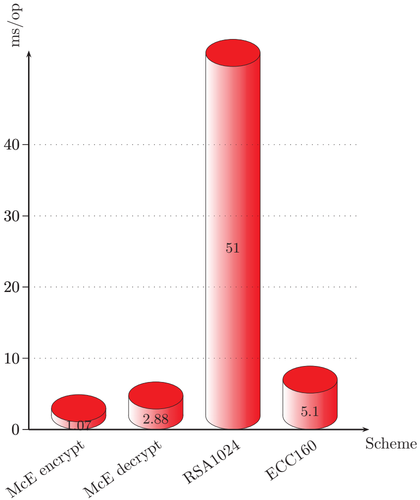
- AVR
  - Encryption over 25 times faster then ECC and only two times slower than RSA.
  - Decryption five times faster than ECC and eight times faster than RSA.

FPGA-Implementation for 80 bit security: Timings



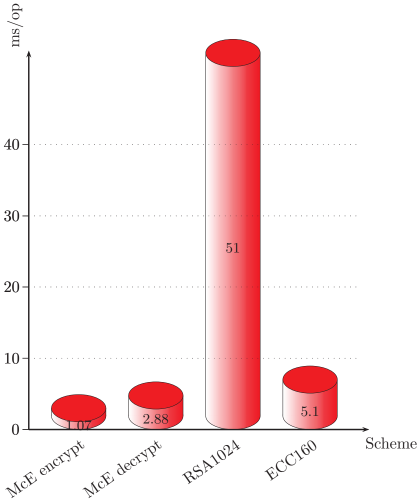
- FPGA
  - Encryption over 47 times faster then RSA and up to five times faster than ECC.
  - Decryption about two times faster than ECC, and 18 times faster than RSA.

FPGA-Implementation for 80 bit security: Timings



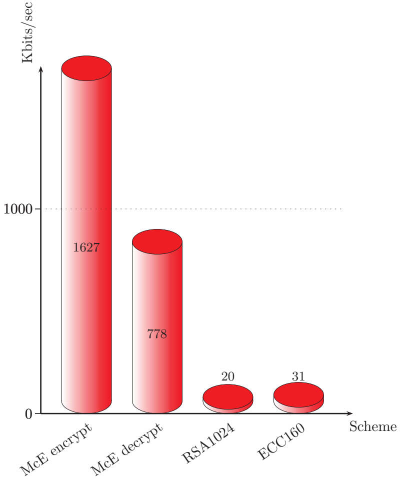
- FPGA
  - Encryption over 47 times faster then RSA and up to five times faster than ECC.
  - Decryption about two times faster than ECC, and 18 times faster than RSA.

FPGA-Implementation for 80 bit security: Timings



- FPGA
  - Encryption over 47 times faster then RSA and up to five times faster than ECC.
  - Decryption about two times faster than ECC, and 18 times faster than RSA.

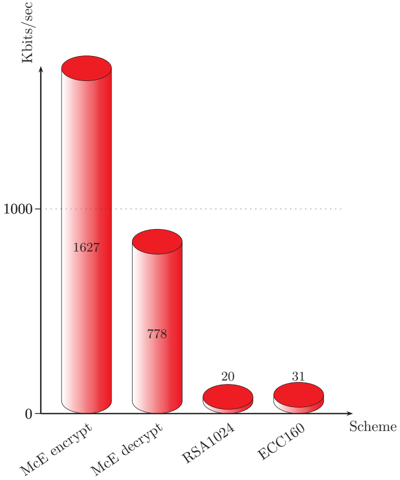
FPGA-Implementation for 80 bit security: Throughput



When taking throughput into account:

- FPGA
  - Encryption over 80 times faster then RSA and 52 times faster than ECC.
  - Decryption 20 times faster than ECC, and 30 times faster than RSA.

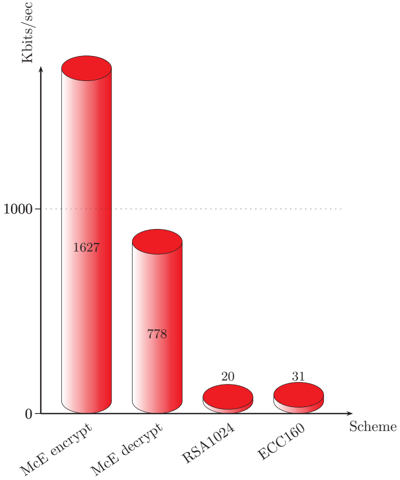
FPGA-Implementation for 80 bit security: Throughput



When taking throughput into account:

- FPGA
  - Encryption over 80 times faster then RSA and 52 times faster than ECC.
  - Decryption 20 times faster than ECC, and 30 times faster than RSA.

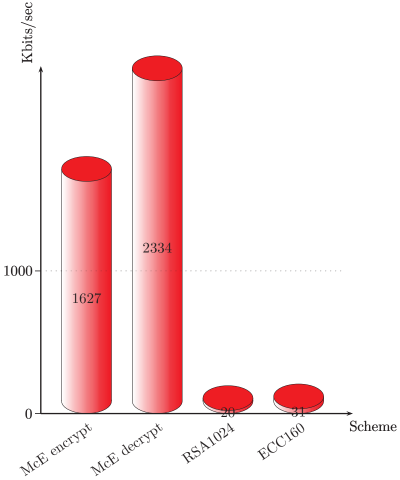
FPGA-Implementation for 80 bit security: Throughput



When taking throughput into account:

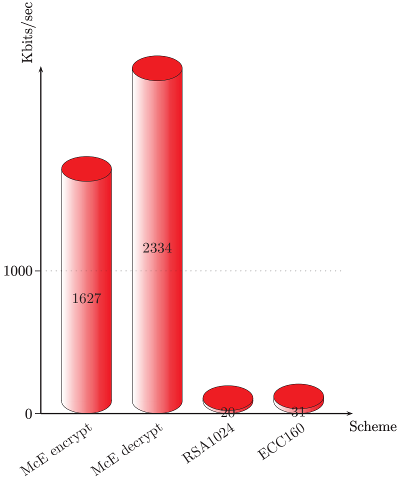
- FPGA
  - Encryption over 80 times faster then RSA and 52 times faster than ECC.
  - Decryption 20 times faster than ECC, and 30 times faster than RSA.

FPGA-Implementation for 80 bit security: Throughput2



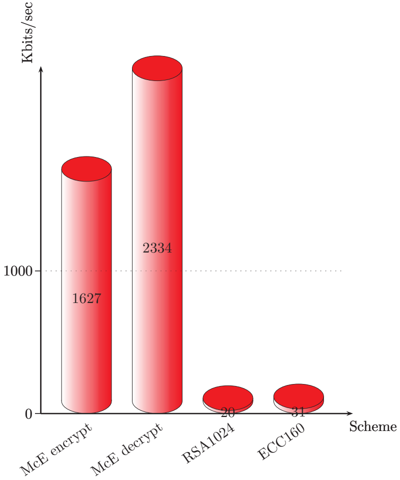
- Three separate algorithm parts
  - Syndrome computation
  - Decoding and error correction
  - Unscrambling
- Pipelined version should double (maybe triple) the throughput

FPGA-Implementation for 80 bit security: Throughput2



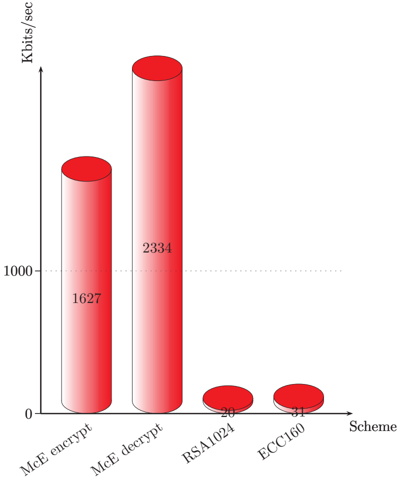
- Three separate algorithm parts
  - Syndrome computation
  - Decoding and error correction
  - Unscrambling
- Pipelined version should double (maybe triple) the throughput

FPGA-Implementation for 80 bit security: Throughput2



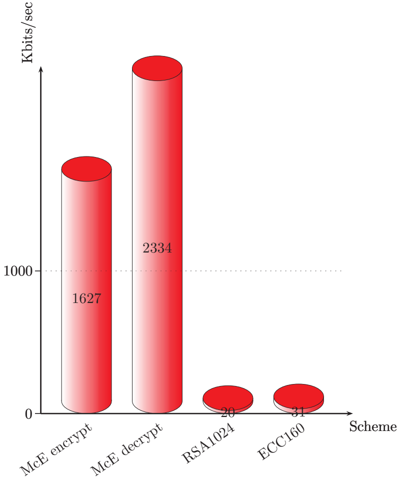
- Three separate algorithm parts
  - Syndrome computation
  - Decoding and error correction
  - Unscrambling
- Pipelined version should double (maybe triple) the throughput

FPGA-Implementation for 80 bit security: Throughput2



- Three separate algorithm parts
  - Syndrome computation
  - Decoding and error correction
  - Unscrambling
- Pipelined version should double (maybe triple) the throughput

FPGA-Implementation for 80 bit security: Throughput2



- Three separate algorithm parts
  - Syndrome computation
  - Decoding and error correction
  - Unscrambling
- Pipelined version should double (maybe triple) the throughput

1 Intro

2 McEliece

3 Adaptions

4 Implementation

5 Results

6 Conclusion

Conclusions

- Proof of concept implementation for 8 bit μC and low cost FPGAs
- μC does not reach timing performance of classic schemes (throughput is in the same order of magnitude)
- but FPGA implementation ROCKS

Conclusions

- Proof of concept implementation for 8 bit μC and low cost FPGAs
- μC does not reach timing performance of classic schemes (throughput is in the same order of magnitude)
- but FPGA implementation ROCKS

Conclusions

- Proof of concept implementation for 8 bit μC and low cost FPGAs
- μC does not reach timing performance of classic schemes (throughput is in the same order of magnitude)
- but FPGA implementation ROCKS

Outlook

- Build semantically secure version (also reduces public key size at the cost of additional computations)
- Better parameters for embedded systems? ($GF(2^8)$ or $GF(2^{16})$)
- Use "Quasi-dyadic Goppa Codes" (R. Misoczki and P. Barreto, SAC2009)

Outlook

- Build semantically secure version (also reduces public key size at the cost of additional computations)
- Better parameters for embedded systems? ($GF(2^8)$ or $GF(2^{16})$)
- Use "Quasi-dyadic Goppa Codes" (R. Misoczki and P. Barreto, SAC2009)

Outlook

- Build semantically secure version (also reduces public key size at the cost of additional computations)
- Better parameters for embedded systems? ($GF(2^8)$ or $GF(2^{16})$)
- Use "Quasi-dyadic Goppa Codes" (R. Misoczki and P. Barreto, SAC2009)

End

Questions?

Further reading

E. R. Berlekamp, R. J. McEliece, and H. C. A. van Tilborg.
On the inherent intractability of certain coding problems.
IEEE Trans. Information Theory, 24(3):384–386, 1978.

D. J. Bernstein, T. Lange, and C. Peters.
Attacking and defending the McEliece cryptosystem.
Cryptology ePrint Archive, Report 2008/318 "http://eprint.iacr.org/", 2008.
http://cr.yp.to/codes/mceliece-20080807.pdf.