A new approximation method for constant weight coding and its hardware implementation

Jingwei Hu, Ray C.C. Cheung *Member, IEEE*, Tim Guneysu *Member, IEEE*

Abstract—We present here a more memory efficient method for encoding binary information into words of prescribed length and weight. Existing solutions either require complicated float point arithmetic or additional memory overhead making it a challenge for resource constrained computing environment. The solution we propose here solves these problems yet obtains better coding efficiency. We also correct a crucial error in previous implementations of code-based cryptography by exploiting and tweaking the proposed encoder. For the time being, the design presented in this work is the most compact one for any code-based encryption schemes. We show for instance that our lightweight implementation of Niederrieter encrypting unit can encrypt approximately 1 million plaintexts per second on a Xilinx Virtex-6 FPGA, requiring 183 slices and 18 memory blocks.

Index Terms—Code-based Cryptography, Niederreiter Cryptosystem, Constant Weight Coding, FPGA Implementation.

1 Introduction

MORDERN public-key cryptographic systems rely on either the integer factorization or discrete logarithm problem, both of which would be easily solvable on large enough quantumn computers using Shor's algorithm [1]. The upcoming breakthroughs of powerful quantum computers have shown their potential in computing solutions to the problems mentioned above [2], [3]. The cryptographic research community has recognized the urgency of this challenge and begun to settle their security on alternative hard problems in the last years, such as multivariate-quadratic, lattice-based and code-based cryptosystems [4]. We address in this paper the problem of encoding information into binary words of prescribed length and Hamming weight in resource-constrained computing environment, e.g. reconfigurable hardware, embedded microcontrol systems, *etc*. This is of interest, in particular, when one wishes to efficiently implement McEliece's scheme [5], [6] or Niederreiter's scheme [7], the most promising candidates for code-based cryptography.

For Niederreiter/McEliece encryption, one needs to convert any binary stream of plaintext into the form of constant weight words. Constant weight means that the the number of '1' in the binary plaintext must be an constant. Moreover, since Niederreiter encryption scheme is extremely fast, this formatting process should be fast as well to maintain the high throughput of data encryption. The exact solution [8] of constant weight coding requires the computation of large binomial coefficients and has a quadratic complexity though it is optimal as a source coding algorithm. In [9], Sendrier proposed the first solution with a linear complexity using Huffman codes. Afterwards, the author of [9] improves its coding efficiency very close to 1 by exploiting Goloumb's run-length encoding method [10]. His proposal is fairly easy to implement and runs in linear time. The price to pay is the variable length encoding [11]. He also proposed an approximation method in this paper by restricting the value of d to the power of two. This method greatly simplifies the encoding process and improves coding throughput.

Heyse *et al.* [12], [13] followed this up and proposed to adapt the approximation method from [11] for embedded system applications. They implemented their design on AVR microcontrollers and Xilinx FPGAs. However, we find such method is not applicable for large parameter sets of the Niederreiter scheme (See Table 1) [14], [15], [16]. Their design maintains a lookup table of pre-stored data with space complexity of $\mathcal{O}(n)$ to encode input messages into constant weight words. This table is still quite big for small embedded systems making their design less unscalable if n is large. For instance, CFS signature scheme [14] exploits very long Goppa code and it requires to compress the lengthy constant weight signatures into binary stream. Recently proposed LDGM sparse syndrome signature scheme [17] also requires a large constant weight coding during the signature generation. At the time being, we do not have a considerably lightweight yet efficient solution for the constant weight encoding when considering these applications.

The purpose of this work* is to tweak Sendrier's approximation method [11] and hence makes it easy to implement for all possible secure parameters of the Niederreiter cryptosystem proposed in literature while maintaining the efficiency. Our contributions include:

1) We propose a new approximation method of constant weight coding without complicated float point arithmetic and memory footprint. This method allows us to implement a compact yet fast constant weight encoder for resource-constrained computing environment.
2) We improve the coding efficiency from the point view of source coding by fine-tuning the optimal

J. Hu and R. Cheung are with the Department of Electronic Engineering, City University of Hong Kong, Hong Kong e-mail: j.hu@my.cityu.edu.hk; r.cheung@cityu.edu.hk. T. Guneysu is with the Department of Mathematics and Computer Science, University of Bremen, Germany e-mail: tim.gueneysu@uni-bremen.de
Manuscript received XX XX, 2016; revised XX XX, 2016.

*The source code of this project is available under https://github.com/davidhoo1988 /ConstantWeightCoding

TABLE 1: Parameters recommended used in the Niederreiter cryptosystem, referenced partly from [13], [17], [18]

(n, t)	Security Level	Application	Coding System	Public/Secret Key Size (Kbits)	Prestored data for CW coding (Kbits)
$(1024, 38)$	60 bit	Encryption	Goppa code	$239/151.4$	4
$(2048, 27)$	80 bit	Encryption	Goppa code	$507/108.3$	8
$(2690, 57)$	128 bit	Encryption	Goppa code	$913/182.4$	10.5
$(6624, 117)$	256 bit	Encryption	Goppa code	$7,488/2,268.5$	25.9
$(65536, 9)$	80 bit	Encryption/Signature	Goppa code	$9,000/1,019$	256
$(262144, 9)$	80 bit	Encryption/Signature	Goppa code	$40,500/4,525$	$1,024$
$(1048576, 8)$	80 bit	Encryption/Signature	Goppa code	$160,000/20,025$	$4,096$
$(9800, 18)$	80 bit	Signature	LDGM code	936/—	19.1
$(24960, 23)$	120 bit	Signature	LDGM code	4,560/—	58.4
$(46000, 29)$	160 bit	Signature	LDGM code	13,480/—	117.2

precision for computing the value of d, when compared with other approximation methods. The experiments have shown the performance of our new method is better than Heyse's approximate version [12] and even comparable to Sendrier's original proposal [11].

3) We integrate our design with the Niederrieter encryption and obtain a more compact result. We fix a critical security flaw of Heyse *et al.*'s Niederrieter encryptor [13]. Our secure implementation of Niederrieter encryptor can encrypt approximately 1 million plaintexts per second on a Xilinx Virtex-6 FPGA, requiring 183 slices and 18 memory blocks.

We will first revisit Sendrier's proposal of constant weight coding and its approximation variant [11], [12] in Section 2. This motivates us to propose a new approximation method and to fine tune it for an optimal source coding performance, presented in Section 3. Section 4 and Section 5 describe our actual implementations for the proposed constant weight encoder/decoder and Niederrieter encryption unit. We present our results compared with the state of arts in Section 6. Section 7 concludes this paper.

2 Sendrier's methods for constant weight coding

Sendrier presented an algorithm for encoding binary information into words of prescribed length and weight [11]. His encoding algorithm returns a t-tuple $(\delta_1, \delta_2, \ldots, \delta_t)$ in which δ_is are the lengths of the longest strings of consecutive '0's. This method is easy to implement and has linear complexity with a small loss of information efficiency. In this work, we unfolded the recursive encoding and decoding algorithm originated from [11], in Algorithm 1 and Algorithm 2.

We use the same notations from [11] in the above two algorithms to keep consistency. For example, read(B, i) moves forward and reads i bits in the stream B and returns the integer whose binary decomposition has been read, most significant bit first; Write(B, i) moves forward and writes the binary string i into the stream B; encodefd(δ_{index}, d) returns a binary string and decodefd(d, B) returns an integer; These two functions are actually the run-length encoding and decoding methods proposed by Golomb [10], [11]. best_d(n, t) returns an integer such that $1 \leq$ best_d$(n, t) \leq n-t$ and Sendrier suggested to choose it close to the number defined by Eq. (1). In fact best_d(n, t) can take any value in

Algorithm 1: Encode Binary String to Constant Weight Word (Bin2CW)

```
Input: message length $n$, message weight $t$ and a binary stream $B$
Output: a $t$-tuple $\Delta = (\delta_1, \delta_2, \ldots, \delta_t)$
1  $\delta = 0, index = 1$
2  while $t > 0$ do
3      if $n \leq t$ then
4          $t--, n--$
5          $\delta_{index} = \delta$
6          $\delta = 0, index++$
7      else
8          $d =$ best_d$(n, t)$
9          if read$(B, 1) = 1$ then
10             $n -= d$, $\delta += d$
11         else
12             $i =$ decodefd$(d, B)$
13             $\delta_{index} = \delta + i$
14             $n -= (i + 1), t--, \delta = 0, index++$
15 return $(\delta_1, \ldots, \delta_t)$
```

Algorithm 2: Decode Constant Weight Word to Binary String (CW2Bin)

```
Input: $n$, $t$ and a $t$-tuple $\Delta = (\delta_1, \delta_2, \ldots, \delta_t)$
Output: a binary string $B$
1  $index = 1$
2  while $t$ != 0 and $n > t$ do
3      $d =$ best_d$(n, t)$
4      if $\delta_{index} \geq d$ then
5          $n -= d$, $\delta_{index} -= d$
6          write$(B, 1)$
7      else
8          write$(B, 0)$
9          write$(B$, encodefd$(\delta_{index}, d))$
10         $n -= (\delta_{index} + 1), t--, index++$
11 return $B$
```

the range, though the efficiency would be reduced if this value is too far from Eq. (1).

$$d \approx (n - \frac{t-1}{2})(1 - \frac{1}{2^{\frac{1}{t}}}) \tag{1}$$

Sendrier also presented an approximation of the best d where the values of d (given by Eq. (1)) was restricted to power of two [11]. More precisely, d is first computed via Eq. (1) and then round to $2^{\lceil log_2(d) \rceil}$. This method greatly simplifies the functions of encodefd$(\cdot)$ and decodefd$(\cdot)$ and therefore gives a significant advantage in speed while the loss of coding efficiency is very limited. The simplified versions of encoding and decoding with encodefd$(\cdot)$ and decodefd$(\cdot)$ after approximation are described as follows:

$$\text{encodefd}(\delta, d) = base_2(\delta, u) \tag{2}$$

$$\text{decodefd}(d, B) = read(B, u) \tag{3}$$

where $base_2(\delta, u)$ denotes the u least significant bits of the integer δ written in base 2 and $u = \lceil log_2(d) \rceil$. For the above two equations, one should particularly pay attention to the case of $d = 1$, the minimum allowed value of d. In this case we have $u = 0$ and therefore we artificially define that $base_2(\delta, 0) = null$ and $read(B, 0) = 0$ for ensuring our algorithm works for all possible u.

Recently, Heyse *et al.* implemented Niederreiter encryption scheme on embedded micro-controllers in which they used a look up table to compute the value of d for constant weight encoding [12]. Their method is based on the approximation method from [11]. One major contribution of their work is they observe that the last few bits of n can be ignored for constant weight encoding because these bits make little difference to the outcome of d value. Therefore they directly remove them and instead insert into the value of t. This method significantly reduces the size of the look up table. According to our analysis, the look up table is shrunk to roughly $\mathcal{O}(n)$. The problem though is that for security parameters with large n, their method might be inappropriate. On the one hand, the size of look up table increases linearly with n, resulting in somewhat unscalability for some applications, in particular, for signatures schemes requiring large n. On the other hand, the coding efficiency drops dramatically and thus lowers the throughput of the constant weight encoder as n increases. We would describe and analysis this point in the next section.

3 Proposed approximation method of d

3.1 Reduce memory footprint and computational steps

The most crucial step of constant weight encoding & decoding is to compute the value of d, that is Eq. (1) requiring floating-point arithmetic. However, many embedded/hardware systems do not have dedicated floating-point units for such computations. [13] proposed to replace floating-point units by a lookup table for reconfigurable hardware. The problem of their method is that, for some (n, t), the lookup table could be large. For example, $(n = 2^{16}, t = 9)$ requests the size of lookup table should be 256 Kb, which is obviously not a negligible memory overhead for embedded systems.

To solve this problem, we propose to replace such table look up by computing d directly using fixed point arithmetic. We split the computation of d into two parts. The first part — $\theta[t] = (1 - 1/2^{\frac{1}{t}})$ is pre-computed and stored in the fixed-point format. The second part $(n - \frac{t-1}{2}) \cdot \theta$ is then efficiently calculated by a fixed-point unit. In this fashion we notably shrink the size of look-up table from $\mathcal{O}(n)$ to $\mathcal{O}(t)$.

Furthermore, we substitute $(n - \frac{t-1}{2})$ by n based on the following observations:

- $n \gg t$ such that $n - \frac{t-1}{2} \approx n$
- Eventually d must be round to an integer and hence the difference between $(n - \frac{t-1}{2})\theta$ and $n\theta$ is very likely to be ignored.

This replacement allows us to remove the computational steps of $n - (t - 1)/2$ and therefore a faster and simpler implementation of constant coding uisng a single multiplication is achievable.

In summary, our new proposal of the approximation of d is as follows:

$$d \approx n \cdot \theta[t] \tag{4}$$

where $\theta[t] = (1 - 1/2^{\frac{1}{t}})$ is a function of t and pre-computed for embedded/hardware designs. Our new approximation of d is more compact, requiring only one multiplication instead. In the following, we will show this method also achieves reasonable high coding efficiency as a source coding algorithm.

3.2 Represent $\theta[t]$ in fixed-point format

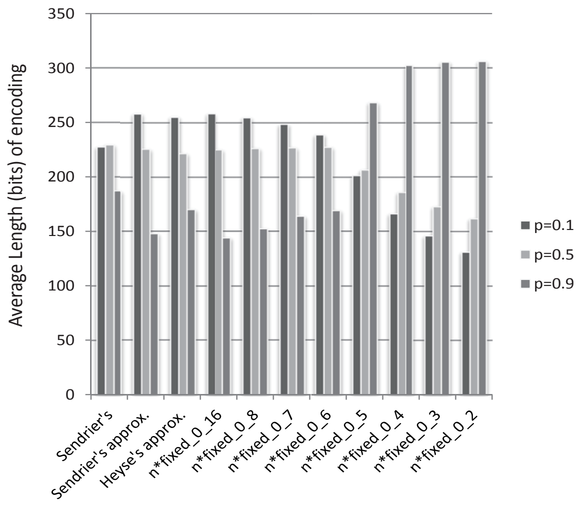
As aforementioned, $\theta[t]$ is an array of fractional numbers and should be stored in fixed-point format. Note that the integer part of $\theta[t]$ must be 0 and therefore we only need to preserve its fractional part. Hereafter, we denote our fixed-point format as fixed_0_i, where i indicates the bits we have used for storing the fractional part of $\theta[t]$.

In practice, we hope to use fixed_0_i with the smallest i while maintaining desirable coding efficiency. Smaller i means lower data storage and a simpler multiplication, which is particularly advisable for resource-constrained computing platforms. Indeed, one of the key issues of this paper is to determine the best fixed_0_i for $\theta[t]$. In the next section, we describe our experiments on exploring the optimal fixed_0_i.

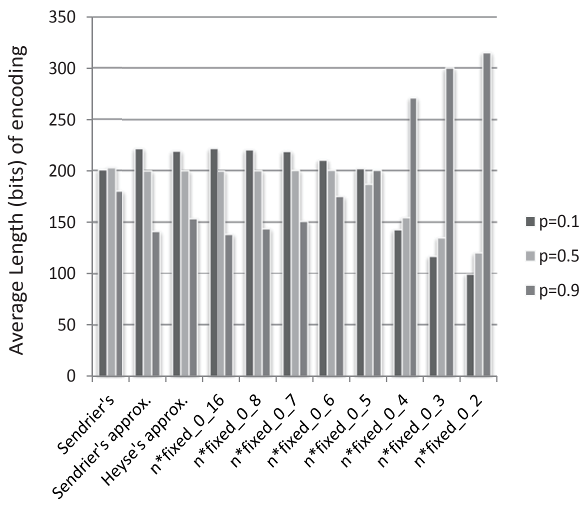
3.3 Find the optimal precision for constant weight encoding

To find the optimal fixed_0_i, we studied how different i could affect the coding efficiency. In our experiments, we investigated all possible precision from fp_0_32 to fp_0_1 to compare with the Sendrier's methods [11] and Heyse's approximate version [12].

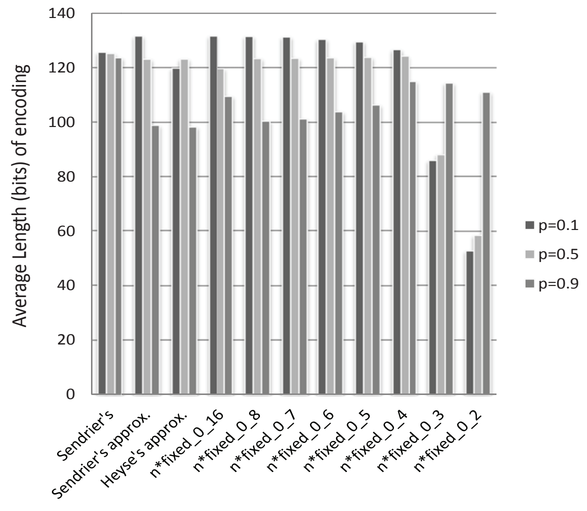
We measured the coding efficiency by calculating the average coding length for a successful encoding because longer coding length indicates a better coding efficiency. Since constant weight coding is variable length coding, we must consider how different plaintexts as input could affect the performance in order to determine which approximation is the best. To be thorough, different types of binary plaintexts, classified by the proportion of '1' contained, should be tested for evaluating the real performance of different encoding approximation methods. In our instances, we measure three particular types to simplify the model: '0' dominated texts ($p = 0.1$, '1' exists with probability of 0.1 in the plaintext), balanced texts ($p = 0.5$, '1' exists with



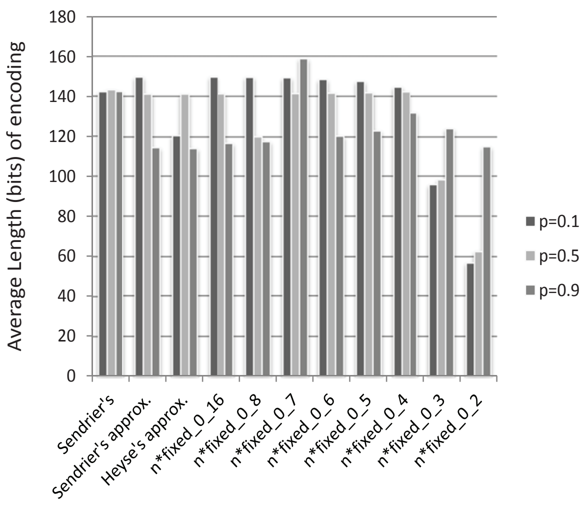
(a) $n = 2^{10}, t = 38$



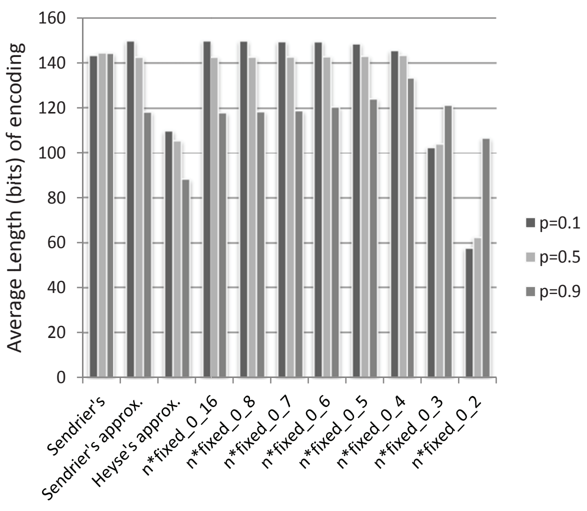
(b) $n = 2^{11}, t = 27$



(c) $n = 2^{16}, t = 9$



(d) $n = 2^{18}, t = 9$



(e) $n = 2^{20}, t = 8$

Fig. 1: The performance of different methods for choosing the optimal d. We have listed five most frequently used sets of (n, t) for the Niederreiter cryptosystem. We have done three experiments for each (n, t) in which the input binary message contains '1' with the probability $p = 0.1$, $p = 0.5$ and $p = 0.9$ respectively. The results of each experiment are obtained by running 10,000 different messages. The X-axis lists different methods including Senderier's primitive [11], Senderier's approximation [11], Heyse's approximation [13] and our n*fixed_0_16 — n*fixed_0_2. The Y-axis represents the average length (bits) of the input message read for a successful constant weight encoding.

probability of 0.5) and '1' dominated texts ($p = 0.9$, '1' exists with probability of 0.9).

Figure 2 describes the coding performances when we adjust the precision of $\theta[t]$. Taken as whole, the $p = 0.1$ group and $p = 0.5$ group have a similar trend of average message length encoded as the arithmetic precision decreases: The message length drops slightly from n*fixed_0_16 — n*fixed_0_2 in consistency. On the contrary, $p = 0.9$ group appears to be quite different where the numbers of bits read for a single constant weight coding first stay stable and then drop with the approximation precision decreasing. The numbers first keep stable because the loss of precision in $\theta[t]$ is comparatively trivial but if the precision drops too low, for instance, with fixed_0_2 representation $\theta[t] = 0$ for $2 \leq t \leq 38$ and hence $d = 0$. It leads to a constant n and small value of i in Algorithm 1 forcing us to read more bits of input stream before the algorithm halts. According to the evaluation criteria mentioned in the last paragraph, we compute the average length of the three types of plaintexts and identify the best approximation of d from our proposal after analyzing the statistics obtained. On the one hand, the n*fixed_0_5 group outperforms at ($n = 2^{10}, t = 38$) and ($n = 2^{11}, t = 27$). On the other hand, the n*fixed_0_4 group beats the others at ($n = 2^{16}, t = 9$), ($n = 2^{18}, t = 9$) and ($n = 2^{20}, t = 8$).

Table 2 compares our proposed methods with the Sendrier's [11], Sendrier's original approximation using power of 2 [11] and Heyse's table lookup approximation [12]. From this table it is seen that our proposal gains better coding efficiency than the original approximation and Heyse's approximation among all five parameter sets used for the Niederreiter scheme. Note that the average number of bits we have to read before producing the constant weight words is upper bound by $log_2\binom{n}{t}$ and the thus the ratio of the average number read and the upper bound measures the coding efficiency [11]. Additionally, our proposal even outperforms the Sendrier's method at two of these parameter sets — ($n = 2^{10}, t = 38$) and ($n = 2^{11}, t = 27$) with 5.05% and 0.95% of improvements, respectively. It is also worth mentioning that for ($n = 2^{16}, t = 9$), ($n = 2^{18}, t = 9$) and ($n = 2^{20}, t = 8$), the performance of our proposal falls slightly behind with 2.37%, 2.08% and 2.22% of loss when compared with the Sendrier's method, it nonetheless outruns Sendrier's approximation and Heyse's approximation. In particular, the performance of Heyse's approximation becomes unfavourable with 23.56% loss at ($n = 2^{20}, t = 8$) and we are pushing the limits of Heyse's method here as the lower bits of n are innegligible and cannot be removed with such large n.

4 Proposed constant weight encoder and decoder

4.1 best_d module

The most critical arithmetic unit is the best_d module which computes the best value of d according to the input n and t. With our proposal of computation of $best\_d$, it consists of three stages performing the following task:

1) **compute $\theta[t]$ via a priority encoder.** As discussed in previous section, format fixed_0_5 is used to represent $\theta[t]$ for ($n = 2^{10}, t = 38$) and ($n = 2^{11}, t = 27$) and fixed_0_4 is used for ($n = 2^{16}, t = 9$), $n = 2^{18}, t = 9$ and $n = 2^{20}, t = 8$. We find by analysis that for some t, the value of $\theta[t]$ are identical. For instance, $\theta[6] = \theta[7] = (0.00011)_2 = 0.09375$ in fixed_0_5 format, shown in Table 3. This observation drives us to exploit priority encoder to simplify the encoding of $\theta[t]$.
2) **compute $n \cdot \theta[t]$ via a fixed point multiplier.** We configured the Xilinx LogiCORE IP to implement high-performance, optimized multipliers for different n and t. The fractional part of the multiplication result is truncated and its integer part is preserved for the next stage to process.
3) **output the value of d and u.** Recall that the value of $n \cdot \theta[t]$ must be round to $d = 2^u$. Here again another priority encoder is used to decode the integer part of $n \cdot \theta[t]$. The detailed decoding process is listed in Table 4.

Figure 2 depicts our best_d unit. This unit works in three stage pipelines. It first computes $\theta[t]$ and then obtains

TABLE 3: Encoding of $\theta[t]$

value of t	$\theta[t] = (0.\theta_1\theta_2\theta_3\theta_4\theta_5)_2$*	$\theta[t] = (0.\theta_1\theta_2\theta_3\theta_4)_2$†
$22 \leq t \leq 38$	00000	N/A
$11 \leq t \leq 21$	00001	
$8 \leq t \leq 10$	00010	0001
$6 \leq t \leq 7$	00011	
$t = 5$	00100	0010
$t = 4$	00101	
$t = 3$	00110	0011
$t = 2$	01001	0100
$t = 1$	10000	1000

*This $\theta[t]$ is represented in fixed_0_5 form, e.g. $\theta[t] = \sum_{i=1}^{5} \theta_i \cdot 2^{-i}$. This format is used in ($n = 2^{10}, t = 38$) and ($n = 2^{11}, t = 27$).
†This $\theta[t]$ is represented in fixed_0_4 form, e.g. $\theta[t] = \sum_{i=1}^{4} \theta_i \cdot 2^{-i}$. This format is used in ($n = 2^{16}, t = 9$), ($n = 2^{18}, t = 9$) and ($n = 2^{20}, t = 8$).

TABLE 4: Decoding of $n \cdot \theta[t]$

integer part of $n \cdot \theta[t]$	value of d	value of u
$n\theta[t] > 2^{18}$	2^{19}	19
$2^{17} < n\theta[t] \leq 2^{18}$	2^{18}	18
$2^{16} < n\theta[t] \leq 2^{17}$	2^{17}	17
$2^{15} < n\theta[t] \leq 2^{16}$	2^{16}	16
$2^{14} < n\theta[t] \leq 2^{15}$	2^{15}	15
$2^{13} < n\theta[t] \leq 2^{14}$	2^{14}	14
$2^{12} < n\theta[t] \leq 2^{13}$	2^{13}	13
$2^{11} < n\theta[t] \leq 2^{12}$	2^{12}	12
$2^{10} < n\theta[t] \leq 2^{11}$	2^{11}	11
$2^{9} < n\theta[t] \leq 2^{10}$	2^{10}	10
$2^{8} < n\theta[t] \leq 2^{9}$	2^{9}	9
$2^{7} < n\theta[t] \leq 2^{8}$	2^{8}	8
$2^{6} < n\theta[t] \leq 2^{7}$	2^{7}	7
$2^{5} < n\theta[t] \leq 2^{6}$	2^{6}	6
$2^{4} < n\theta[t] \leq 2^{5}$	2^{5}	5
$2^{3} < n\theta[t] \leq 2^{4}$	2^{4}	4
$2^{2} < n\theta[t] \leq 2^{3}$	2^{3}	3
$2 < n\theta[t] \leq 2^{2}$	2^{2}	2
$1 < n\theta[t] \leq 2$	2	1
$n\theta[t] \leq 1$	1	0

TABLE 2: The coding performance of the optimal d chosen from our approximation method

n	t	method	number of bits read			coding efficiency	efficiency improved
			maximum	minimum	average		
2^{10}	38	Sendrier's [11]	236	164	214.70	93.19%	—
		Sendrier's approx. [11]	263	124	210.45	91.32%	-1.98%
		Heyse's approx. [13]	261	143	210.73	91.44%	-1.85%
		n*fixed_0_5	311	183	225.54	97.89%	**+5.05%**
2^{11}	27	Sendrier's [11]	208	160	194.46	95.51%	—
		Sendrier's approx. [11]	227	119	187.55	92.11%	-3.56%
		Heyse's approx. [13]	224	127	190.96	93.78%	-1.80%
		n*fixed_0_5	361	132	196.30	96.41%	**+0.95%**
2^{16}	9	Sendrier's [11]	133	113	124.87	99.50%	—
		Sendrier's approx. [11]	133	77	117.80	93.84%	-5.10%
		Heyse's approx. [13]	133	86	116.34	92.68%	-6.83%
		n*fixed_0_4	135	95	121.98	97.20%	**-2.37%**
2^{18}	9	Sendrier's [11]	148	132	142.61	99.38%	—
		Sendrier's approx. [11]	151	91	135.17	94.18%	-5.22%
		Heyse's approx. [13]	300	101	133.21	92.81%	-6.60%
		n*fixed_0_4	154	112	139.64	97.31%	**-2.08%**
2^{20}	8	Sendrier's [11]	149	135	144.00	99.52%	—
		Sendrier's approx. [11]	151	99	136.94	94.64%	-4.90%
		Heyse's approx. [13]	265	17	110.07	76.07%	-23.56%
		n*fixed_0_4	158	109	140.81	97.31%	**-2.22%**

$n \cdot \theta[t]$ using a multiplier. Finally, the value of d would be determined by a priority decoder.

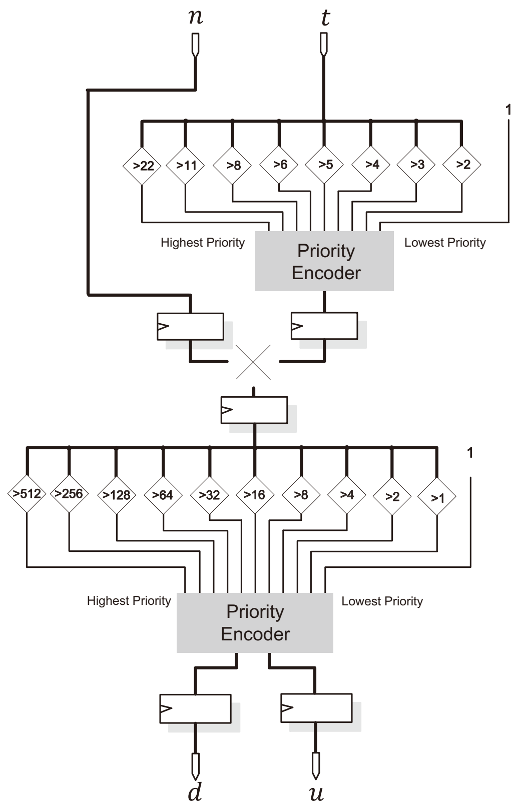


Fig. 2: best_d module for CW encoder and decoder. We list here the detailed configurations of $n = 2^{11}, t = 27$ for demonstrative purpose.

4.2 Bin2CW encoder

Figure 3 depicts the architecture of the proposed constant weight encoder. Input binary message is passed inwards using a non-symmetric 8-to-1 FIFO which imitates the function of read$(B, 1)$. We use a serial-in-parallel-out shift register to perform read$(B, u), 0 \leq u \leq \lceil log_2(\frac{n}{2})\rceil$. The proposed best_d module is exploited here to compute the value of d. The values of n, t, δ are accordingly updated using three registers.

4.3 CW2Bin decoder

Figure 4 depicts the architecture of the proposed constant weight decoder. A m-to-m bit FIFO is used instead to transfer the input t-tuple word by word. This logic is actually the bottle neck in the constant weight decoder when compared with the encoder. We similarly use three registers to update the values of n, t, δ. The major difference is that the shift register here outputs the value of δ bit by bit as step 9 of Algorithm 2 requires.

5 Integrating with the Niederrieter Encryptor

In this section, we demonstrate how the proposed Bin2CW encoder can integrate with the Niederrieter encryptor for data encryption, shown in Algorithm 3.

```
Algorithm 3: Niederreiter Message Encryption, referenced from [18]

  Input: message vector $m$, public key $pk = \{\hat{H}, t\}$
         where $\hat{H}$ is an $n$ by $mt$ matrix
  Output: ciphertext $c$
1 Bob encodes the message $m$ as a binary matrix/vector
  of length $n$ and weight at most $t$.
2 Bob computes the ciphertext as $c = \hat{H}m^T$, $m^T$ is the
  transpose of matrix $m$.
3 return $c$
```

The Bin2CW encoder is used to perform the first step in this algorithm. Recall that Bin2CW encoder returns a t-tuple

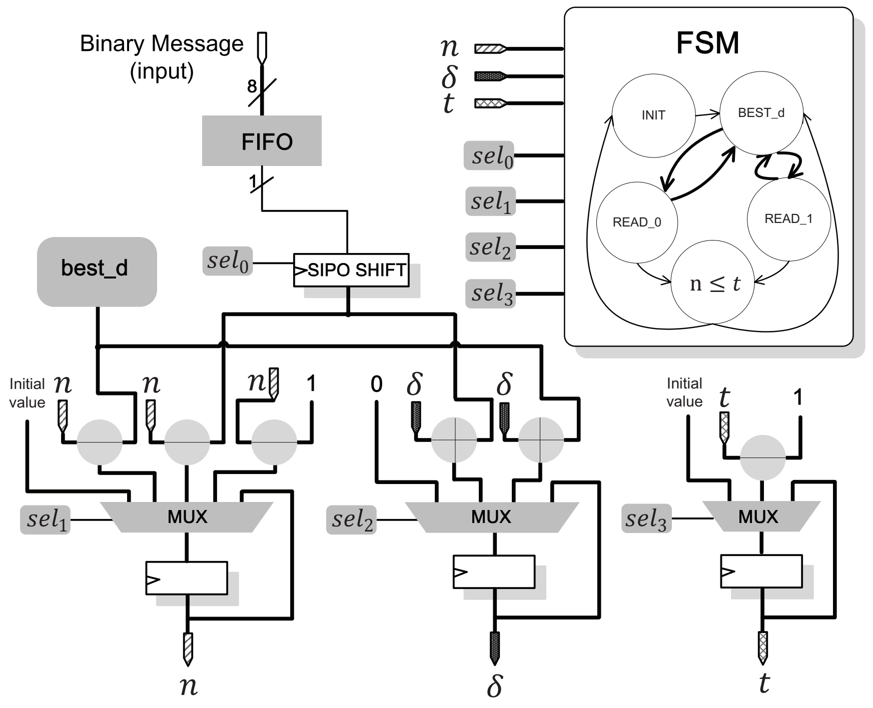


Fig. 3: General architecture of CW encoder

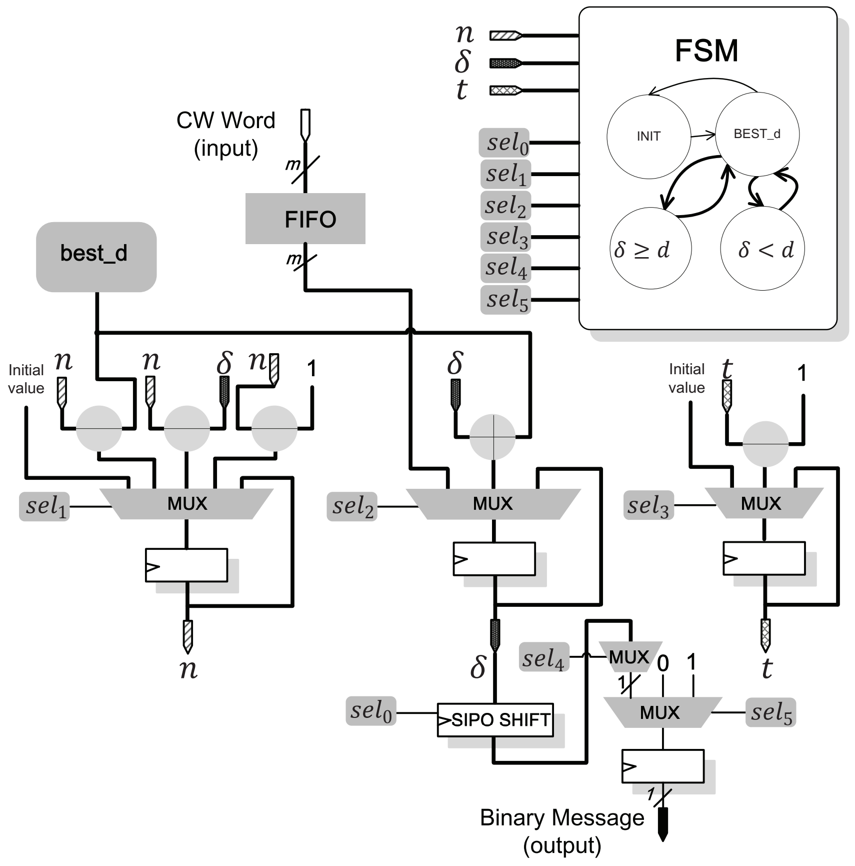


Fig. 4: General architecture of CW decoder

of integers $(\delta_1, \ldots, \delta_t)$, which records the distance between consecutive '1's in the string. However such t-tuple cannot be directly transferred to compute the ciphertext. We believe that the way Heyse *et al.* [13] encrypts $c = \hat{H}m^T$ with $m = (\delta_1, \ldots, \delta_t)$ is incorrect due to two reasons:

1) It is very likely that $\delta_i = \delta_j$ where $i \neq j$ such that the

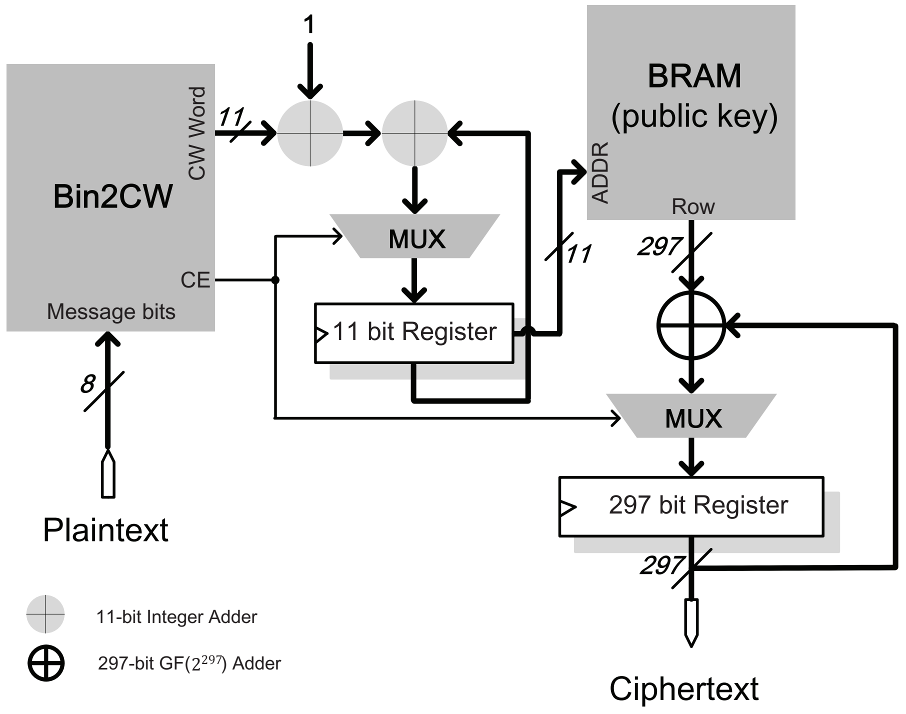


Fig. 5: Block diagram of the Niederrieter encryptor.

number of errors are less than t and it is assumed to be insecure from cryptoanalysis points of view.

2) $(\delta_1, \ldots, \delta_t)$ returns the integer ranging from 0 to $n-t$ but the constant weight word exactly ranges from 0 to n. That is to say, the last t rows of the public key $\hat{H}^T$ are never used.

To fix this flaw from [13], we propose to return the 'real' constant weight binary words of length n and Hamming weight t. Suppose the constant weight are represented by $(i_1, \ldots, i_t)$, the coordinates of the '1's in ascending order, then we compute $i_1 = \delta_1$, $i_2 = \delta_2 + i_1 + 1$, ..., $i_t = \delta_t + i_{t-1} + 1$ as the input of the second step, Algorithm 3.

Figure 5 depicts our revised version of Niederreiter encryption unit on the basis of [13]. The public key $\hat{H}^T$ is stored in an internal BRAM and row-wise addressed by the output of the 11 bit register. Two 11-bit integer adders are embedded to convert $(\delta_1, \ldots, \delta_t)$ to $(i_1, \ldots, i_t)$ which are eventually stored in the 11 bit register. The vector-matrix multiplication in step 2, Algorithm 3 is equivalent to a XOR operation of selected rows of $\hat{H}^T$, denoted here as a $GF(2^{297})$ adder in this figure. It is also worth mentioning that the vector-matrix multiplication work concurrently with CW encoding: Whenever a valid i_k has been computed, it is transferred to the $GF(2^{297})$ adder summing up the selected rows. After the last i_t has been computed, the last indexed row of $\hat{H}^T$ also has been added to the sum. This sum, stored in the 297-bit register, is now available as the ciphertext.

6 Results and comparisons

We captured our constant weight coding architecture in the Verilog language and prototyped our design on Xilinx Virtex-6 FPGA (Table 5). To the best of our knowledge, the only compact implementations of constant weight coding have been proposed by Heyse *et al.* [13]. Their light weight architecture is generally identical to ours except the design of best_d module. Their best_d module works in two pipeline stages: In the first stage it retrieves the value of u by table lookup. Then in the second stage it outputs d according to the value of u using a simple decoder. Comparatively, our best_d module has three stages of pipeline and thus it leads to a lower throughput but our architectures are smaller and improve the area-time tradeoff of the constant weight coding implementations proposed by Heyse *et al.* [13], shown in Table 5. In particular, we use only one $18Kb$ memory block for all parameter sets of our experiments.

We also observe that in our designs, the memory footprint does not increase and the high clock frequency also maintains as the parameters grow. This is because the main difference among encoders or decoders with different parameters n and t is the operand size of multiplier embedded in the best_d module, which grows logarithmically from $10bit \times 5bit$ to $20bit \times 4bit$. On the other hand, the memory overhead of Heyse's implementations grow linearly with n and might be problematic when n is large as aforementioned. To show this, we re-implemented Heyse's work for $(n = 2^{16}, t = 9)$, $(n = 2^{18}, t = 9)$ and $(n = 2^{20}, t = 8)$. The experimental results verifies this point. Additionally, another negative side effect of heavy memory overhead is that the working frequency of circuits drops rapidly as shown in Table 5. For small parameters (a) and (b), the lookup table in Heyse's design could be made of distributed memory (LUT) and therefore has little impact on frequency. However for large parameters (c), (d) and (e), such look up table can no longer be instantized as LUTs because Xilinx Virtex-6

TABLE 5: Compact implementations of CW encoder and decoder on Xilinx Virtex-6 FPGA

(a) $n = 2^{10}, t = 38$

	Algorithm	Platform	Area [slices]	Memory blocks [36 + 18 Kb RAM]	Frequency [MHz]	Throughout [Mbps]	Coding efficiency
This work	Bin2CW, new approx.	Xilinx xc6vlx240t	74	0+1	330	160.2	97.89%
	CW2Bin, new approx.		79	0+1	330	148.1	
Heyse *et al.* [13]	Bin2CW, Heyse's approx.	Xilinx xc6vlx240t	110	0+1	310	178.8	91.44%
	CW2Bin, Heyse's approx.		88	0+1	310	162.1	

(b) $n = 2^{11}, t = 27$

	Algorithm	Platform	Area [slices]	Memory blocks [36 + 18Kb RAM]	Frequency [MHz]	Throughout [Mbps]	Coding efficiency
This work	Bin2CW, new approx.	Xilinx xc6vlx240t	91	0+1	350	187.2	96.41%
	CW2Bin, new approx.		95	0+1	340	168.6	
Heyse *et al.* [13]	Bin2CW, Heyse's approx.	Xilinx xc6vlx240t	118	0+1	340	208.4	93.78%
	CW2Bin, Heyse's approx.		110	0+1	340	164.6	
Sendrier [11]*	Bin2CW, original	Intel Pentium 4	—	—	2400	17.3	95.51%
	Bin2CW, approximate					33.0	92.11%

*Sendrier implemented a different but very close parameter set $n = 2^{11}, t = 30$. We also put it here for reference.

(c) $n = 2^{16}, t = 9$

	Algorithm	Platform	Area [slices]	Memory blocks [36 + 18 Kb RAM]	Frequency [MHz]	Throughout [Mbps]	Coding efficiency
This work	Bin2CW, new approx.	Xilinx xc6vlx240t	103	0+1	440	316.1	97.20%
	CW2Bin, new approx.		109	0+1	310	212.9	
Heyse *et al.* [13]	Bin2CW, Heyse's approx.	Xilinx xc6vlx240t	90	10+3	240	182.3	92.68%
	CW2Bin, Heyse's approx.		90	10+3	230	170.3	
Sendrier [11]	Bin2CW, original	Intel Pentium 4	—	—	2400	18.3	99.50%
	Bin2CW, approximate					22.0	93.84%

(d) $n = 2^{18}, t = 9$

	Algorithm	Platform	Area [slices]	Memory blocks [36 + 18Kb RAM]	Frequency [MHz]	Throughout [Mbps]	Coding efficiency
This work	Bin2CW, new approx.	Xilinx xc6vlx240t	138	0+1	410	295.5	97.31%
	CW2Bin, new approx.		118	0+1	320	219.5	
Heyse *et al.* [13]	Bin2CW, Heyse's approx.	Xilinx xc6vlx240t	94	40+3	170	106.0	92.81%
	CW2Bin, Heyse's approx.		93	40+3	180	104.9	

(e) $n = 2^{20}, t = 8$

	Algorithm	Platform	Area [slices]	Memory blocks [36 + 18Kb RAM]	Frequency [MHz]	Throughout [Mbps]	Coding efficiency
This work	Bin2CW, new approx.	Xilinx xc6vlx240t	156	0+1	370	284.9	97.31%
	CW2Bin, new approx.		122	0+1	300	222.7	
Heyse *et al.* [13]	Bin2CW, Heyse's approx.	Xilinx xc6vlx240t	124	160+3	130	96.6	76.07%
	CW2Bin, Heyse's approx.		125	160+3	130	98.8	

distributed memory generator only allows maximum data depth of 6,5536. We instead use block memory outside the FPGAs to construct the table and this accordingly leads to slower speed due to relatively far and complicated routing. This block memory is the real bottleneck of Heyse's work as n grows.

We finally implemented the Niederreiter encryptor, a cryptographic application where constant weight coding is used exactly as mentioned in Section 5. Table 6 compares our work with the state of art [13]. Our new implementation is the most compact one with better area-time tradeoffs. We use the same block memory as [13] did where $16 \times 36\text{Kb} + 1 \times 18\text{Kb}$ RAMs are utilized to store the public key matrix $\hat{H}$, and one 18Kb RAM for the 8-to-1 FIFO within the constant weight encoder.

7 Conclusion

We proposed a new method of determining the optimal value d in constant weight coding. This method enabled a more compact yet efficient implementation of constant weight encoder and decoder for resource-constrained computing systems. Afterwards, we exploited this new encoder

TABLE 6: FPGA Implementation results of Niederreiter encryption with $n = 2048, t = 27$ compared with [13] after PAR

Aspect (Virtex6-VLX240)	Niederreiter [13]	**This work**
Slices	315	183
LUTs	926	505
FFs	875	498
BRAMs	17	18*
Frequency	300 MHz	340 MHz
CW Encode e = Bin2CW(m)	$\approx$ 200 cycles	349.1 cycles
Encrypt $c = e \cdot \hat{H}$	$\approx$ 200 cycles	352.1 cycles

*We used 16×36Kb RAMs and 2×16Kb RAMs

to implement the Niederreiter encryptor on a Xilinx device. For the time being, our design is the most compact one for any of the code-based encryption schemes.

Acknowledgments

The authors would like to thank...

References

[1] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM journal on computing*, vol. 26, no. 5, pp. 1484–1509, 1997.

[2] L. M. Vandersypen, M. Steffen, G. Breyta, C. S. Yannoni, M. H. Sherwood, and I. L. Chuang, "Experimental realization of shor's quantum factoring algorithm using nuclear magnetic resonance," *Nature*, vol. 414, no. 6866, pp. 883–887, 2001.

[3] N. Xu, J. Zhu, D. Lu, X. Zhou, X. Peng, and J. Du, "Quantum factorization of 143 on a dipolar-coupling nmr system," *arXiv preprint arXiv:1111.3726*, 2011.

[4] D. J. Bernstein, "Introduction to post-quantum cryptography," in *Post-Quantum Cryptography*. Springer, 2009, pp. 1–14.

[5] R. J. McEliece, "A public-key cryptosystem based on algebraic coding theory," *DSN progress report*, vol. 42, no. 44, pp. 114–116, 1978.

[6] N. Sendrier, "Code-based public-key cryptography," in *Post-Quantum Cryptography Summer School*, 2014.

[7] H. Niederreiter, "Knapsack-type cryptosystems and algebraic coding theory," *Pproblems Of Control and Information Theory-Problemy Upravleniya I Thorii Informatsii*, vol. 15, no. 2, pp. 159–166, 1986.

[8] T. M. Cover, "Enumerative source encoding," *Information Theory, IEEE Transactions on*, vol. 19, no. 1, pp. 73–77, 1973.

[9] N. Sendrier, "Efficient generation of binary words of given weight," in *Cryptography and Coding*. Springer, 1995, pp. 184–187.

[10] G. Goloumb, "Run length encoding," *Information Theory, IEEE Transactions on*, vol. 12, pp. 399–401, 1966.

[11] N. Sendrier, "Encoding information into constant weight words," in *Information Theory, 2005. ISIT 2005. Proceedings. International Symposium on*. IEEE, 2005, pp. 435–438.

[12] S. Heyse, "Low-reiter: Niederreiter encryption scheme for embedded microcontrollers," in *Post-Quantum Cryptography*. Springer, 2010, pp. 165–181.

[13] S. Heyse and T. Güneysu, "Towards one cycle per bit asymmetric encryption: code-based cryptography on reconfigurable hardware," in *Cryptographic Hardware and Embedded Systems–CHES 2012*. Springer, 2012, pp. 340–355.

[14] N. T. Courtois, M. Finiasz, and N. Sendrier, "How to achieve a Mceliece-based digital signature scheme," in *Advances in Cryptology-ASIACRYPT 2001*. Springer, 2001, pp. 157–174.

[15] G. Landais and N. Sendrier, "CFS Software Implementation." *IACR Cryptology ePrint Archive*, vol. 2012, p. 132, 2012.

[16] M. Finiasz, "Parallel-CFS," in *Selected areas in cryptography*. Springer, 2011, pp. 159–170.

[17] M. Baldi, M. Bianchi, F. Chiaraluce, J. Rosenthal, and D. Schipani, "Using ldgm codes and sparse syndromes to achieve digital signatures," in *Post-quantum cryptography*. Springer, 2013, pp. 1–15.

[18] J. Hu and R. C. Cheung, "An application specific instruction set processor (ASIP) for the Niederreiter cryptosystem," Cryptology ePrint Archive, Report 2015/1172, 2015, http://eprint.iacr.org/2015/1172.pdf.